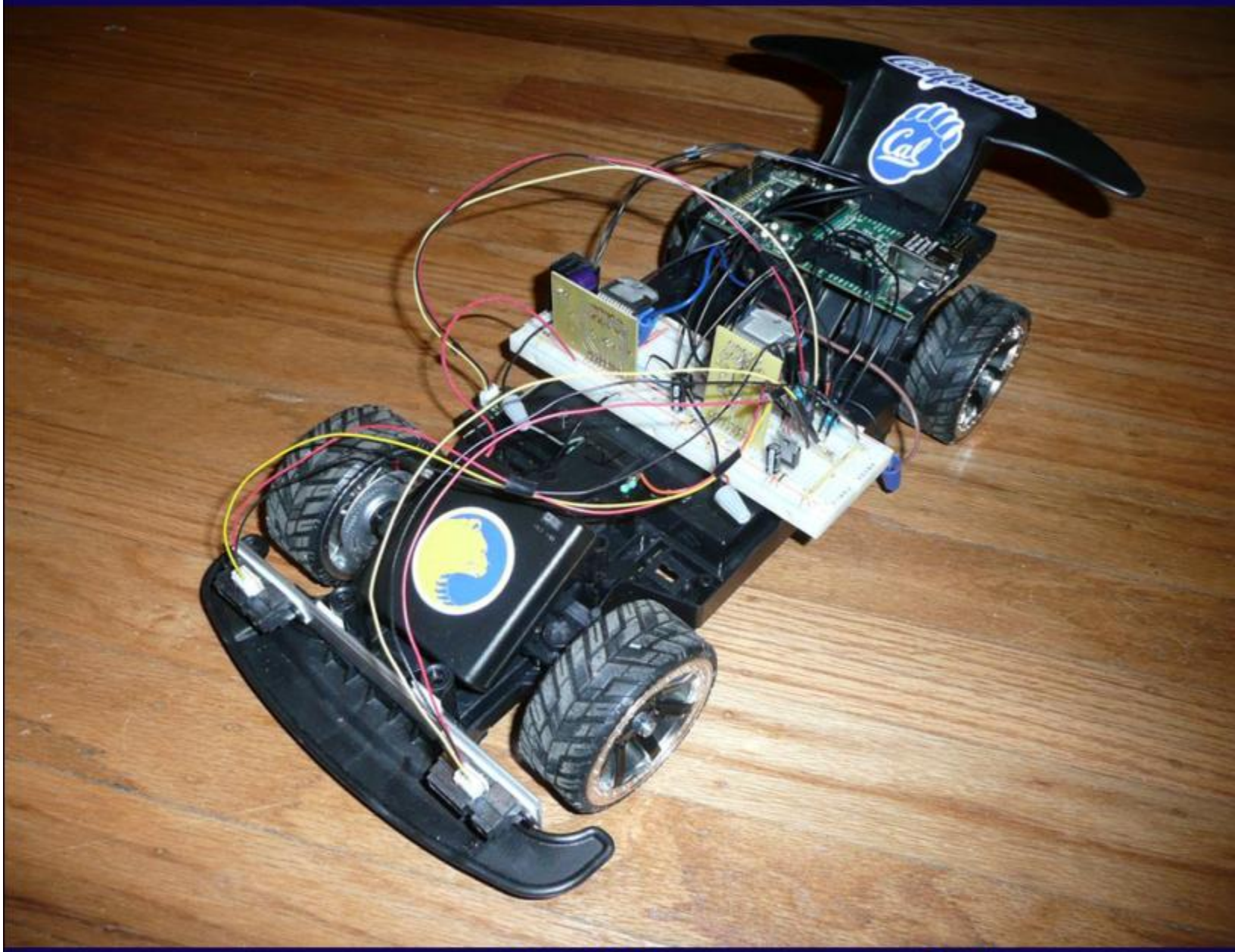
Cal Autonomous Robot C.A.R.



University of California at Berkeley
Department of Mechanical Engineering

Ben Chu Mikhail Podust Stephen Tu

Overview and Acknowledgements

The car was constructed as a final project for the class ME135: *Design of Microprossor-Based Mechanical Systems* during the Spring 2009 semester at the University of California, Berkeley. We would like to thank George Anwar, Jae Kim, and Tom Clark for their guidance and support in helping us make this project a reality.

Original Proposal and Expected Tasks

The following proposal was presented February of 2009 to George Anwar, Jae Kim, and the students of ME135:

Objective

Our group proposes to design and build an autonomous vehicle to follow a lead car. Additionally, the chasing car will replicate the exact path the lead car takes while maintaining a constant distance from it.

Applications

This system has potential for mass deployment in both military and civilian applications. For instance, military convoys of dozens of trucks can be guided by a single driver in the lead vehicle, reducing the amount of personnel exposed to danger. Furthermore, driverless vehicles have many advantages over vehicles with drivers. Automated vehicles can be more reliable and more accurate than a human, which is prone to fatigue and error. If the project is successful, other features, like lane changing for automated highways can be integrated into the system.

Software Tasks

Controller

The code will be written in LabVIEW or C. We will be implementing a basic PID controller and measure the error of the system as the difference between the chase car's current position and the centerline of the lead car. The error will be fed to the control algorithm, which will process the input and create output signals for the actuators controlling the knobs on the remote control. The continuous measurement of the error will provide a real-

time task for the software, while the simultaneous control of the actuators introduces multitasking. The gains of the PID controller will be tuned during testing of our prototype.

GUI

The LabVIEW front panel will provide a human-machine interface to allow the user to adjust the distance at which the vehicle will follow the lead car. The GUI will also display a graph of the global x and y coordinates of the car. The front panel will also allow for a manual override to stop the vehicle in case of emergencies. The feedback gains can also be adjusted to control the speed of the response, and the steady state error.

Hardware

RC Car

An RC car will be purchased from a toy store or hobby shop, and will serve as the autonomous vehicle. We intend to mount all of the hardware on this car. Depending on the vehicle, custom modifications may have to be made to accommodate the extra hardware. As such, we intend to machine and attach a rack to the car to house the luminary board and its accompanying power supply.

Feedback Sensors

The lead car will shoot an infrared signal back to the autonomous vehicle as a means to measure distance. Two sensors are needed to track the longitudinal distance to the lead car, the lateral distance from the centerline of the lead car, and the yaw angle with respect to the centerline of the car. The data gathered by these sensors will be processed by our microcontroller and corresponding output signals will be sent to our actuators controlling the remote control.

Controller: Luminary Board

Due to our severe size and weight limits, a small microcontroller is needs to be mounted on the RC car. Consequently, we will use the Luminary Board as our microcontroller because other alternatives, such as the CompactRIO, are too large and heavy for our application. Furthermore, for this system to work, our final design will be subject to two conditions. The first is that the microcontroller must be powered by batteries on board the car, not a USB cable connected to a computer. The second condition is the requirement that the car can transmit wireless signals to the actuators controlling the remote control.

Actuators

Since RC cars already come with a convenient way to control motion, we will use the out-of-the box remote control to direct the follow car's motion. However, we intent to use motors controlled by output signals from the Luminary Board to automatically move the levers.

Summary

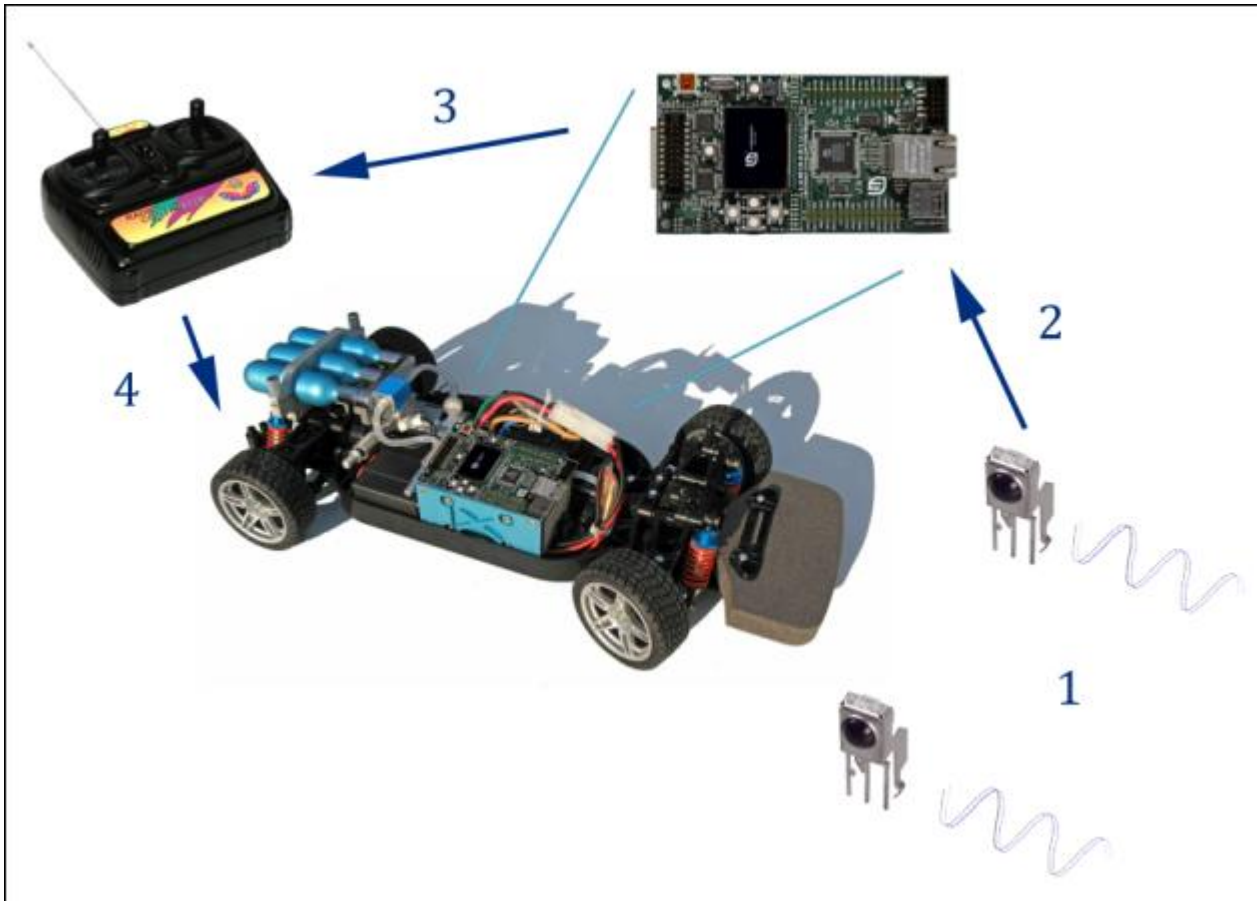


Figure 1: Conceptual Overview of the Proposed Design (Lead Vehicle Not Shown)

1. Front mounted, forward facing IR sensors collect distance measurements.
2. Data is sent to the luminary board where it is processed. Location of the chase vehicle with respect to the lead car (error) is ascertained.
3. Based on the calculated error, the PID control algorithm on the luminary outputs commands to actuators (not shown) that are directly mounted the remote control.
4. As in standard RC car operation the remote controls forward motion and steering on the vehicle.

Testing

A prototype will be built, which will provide a testing ground for the software. Since we will be sensing the distance between sensors, they will have to be carefully calibrated to provide meaningful data. We will then fine tune the feedback gains of the PID controller to make sure the entire system is stable. Our last concern is the speed at which the Luminary

Board can process and output data. We will have to be efficient with our computer code and precise with our timing to make sure the follow car is properly controlled. If the microcontroller cannot process data fast enough, the result could be that the second car can only follow the lead car in a narrow range of distances.

Implementation Plan

Materials

- Purchase powerful RC car and batteries - toy store, hobby shop ($50).
- Obtain aluminum for the electronics rack- Alco metals, scrap at machine shop.
- Obtain IR/RF/ultrasound signal generator and sensors and gyroscope: angle.
- Obtain accelerometer, and quadrature encoder: speed.
- Obtain motors to actuate our remote or PWM to motors.

Modeling Vehicle Dynamics

- Send commands to motors accurately using Labview+Luminary and PWM or actuators, with little time delay. (Need 2 digital inputs)
- Amplify signals or remove noise before it reaches the motor.
- How does a signal from PWM and remote translate into acceleration, and angle?
- Use derivatives to find velocity and angular velocity.
- Figure out a relationship that fits the observed behavior (hopefully linear).
- How does angle from remote translate into acceleration, velocity and angle?
- Convert rotational motion into linear motion, using a rack and pinion system.

Sensors

Acquire data from the IR sensors accurately using Labview+Luminary, at a relatively high sampling rate. (Need 3 analog inputs)

- Amplify signals and remove noise from the sensors.
- How does the distance between IR/RF sensors translates into a signal?
- Figure out a relationship that fits the observed behavior (hopefully linear).
- Find a geometric relation between the distances of sensors and the distance and angle between vehicles.
- Use derivatives to find relative velocity and relative angular velocity.
- Figure out how to obtain the velocity of the car using a quadrature encoder or sensors. (Need 1 digital input).
- Find out how the signal from the accelerometer translates into velocity and the signal from the gyroscope translates into angle. (Need 1 analog input)

Mechanical Design and Fabrication

- Model and machine parts.
- Assemble parts together.

Electrical Design and Prototyping

- Model circuit diagram of the system, which includes the analog inputs and outputs, amplifiers and filters.
- Prototype the circuit on a breadboard for testing.
- Figure out power requirements of the entire circuit and attach batteries to externally power the circuit.
- Figure out wireless capabilities, if possible.

Longitudinal Control Testing

- Implement a PID algorithm for achieving a set relative distance with a stationary lead car.
- Tune the gains until the response is relatively quick, with a small steady state error.
- The next step is to generalize the PID control for a moving or accelerating lead car that only travels forward in a straight line.
- Next, generalize to a car that can also reverse direction.

Lateral Control Testing

- Implement a PID algorithm for achieving a set angular distance with a stationary lead car.
- Tune the gains until the response is relatively quick, with a small steady state error.
- The next step is to generalize the PID control for a moving or accelerating lead car that is turning.
- The final step is to combine longitudinal and lateral control and test the response performance for a car that travels without restraint, while optimizing the code for speed.

LabView Interface

- Determine operator interface and data logging for the controller (GUI).
- Debug, include error handling (code).

Achieved Tasks

Overview

The majority of the tasks listed in the original proposal were achieved. The longitudinal control was really effective, due to controlling the drive motor with the on-board Pulse Width Modulation (PWM) signal. One of the few exceptions was proportional turning. The steering control was not as effective, mostly in part due to poor hardware. The steering was controlled by a DC motor, instead of a servo motor, which would have been able to control the steering angle. With the DC motor, steering control consisted of either a full turn to the left or right.

Plant

The final design used a 1:10 scale toy remote controlled car. It had two DC motors to control steering and driving. It came with a transmitter and the on board circuitry was replaced with custom circuitry and the microcontroller. The entire car is powered by 12 AA batteries supplying a maximum of 18V to the car. The batteries are mounted in two locations. Four are located at the front and the other eight are conveniently stowed underneath the car (where the batteries were intended to be stored originally).

The performance of the motor is limited by the physical considerations of the system; the motor cannot achieve an instantaneous change in power, which limits the speed of the controller response. However, to prevent the motor from burning out, the duty cycle of the PWM sent to the motor was limited (in software) at the highest experimentally determined value the motor could take without incurring damage. A consequence is that during aggressive maneuver when a large motor output is required by the control algorithm, the car's performance will saturate at the set threshold. This was a necessary tradeoff because burning out the motor is a real possibility (as we experienced with our first car), but a good control algorithm should be able to prevent large values in the required duty cycles, provided the lead car is within a reasonable distance from the object.

Sensors

Sensors commonly used in hobby robotics (the Sharp GP2D12 infrared rangefinder series) were used. The sensor has an approximate sampling period of 38 ms, with an error of ±10ms, which limits its precision. Each sensor also has a range of approximately 10-50 cm. Outside that range the sensors are unable to provide meaningful data. Three sensors were used to output data about the location of the tracked object and various obstacles. The sensor outputs a voltage, which has a nonlinear relationship with distance. Data points were collected and a mathematical model was generated. By inspection, we saw that the relationship was an inverse function of the form $V(x) = 1/ax + C$, where "a" and "C" are constants. After tuning the constants, we settled on the function $V(x) = 1/0.03x - 4$, which fit the data pretty well. We could have used an nth-order least squares regression to get a more precise fit of the data, but that would have required computing high order polynomials every iteration while the car is running. This would require quite a few floating point operations per iteration, which we would like to minimize. It is also not necessary, since the data we collected was only accurate up to about two significant figures (due to the tick marks in the measuring tape we used).

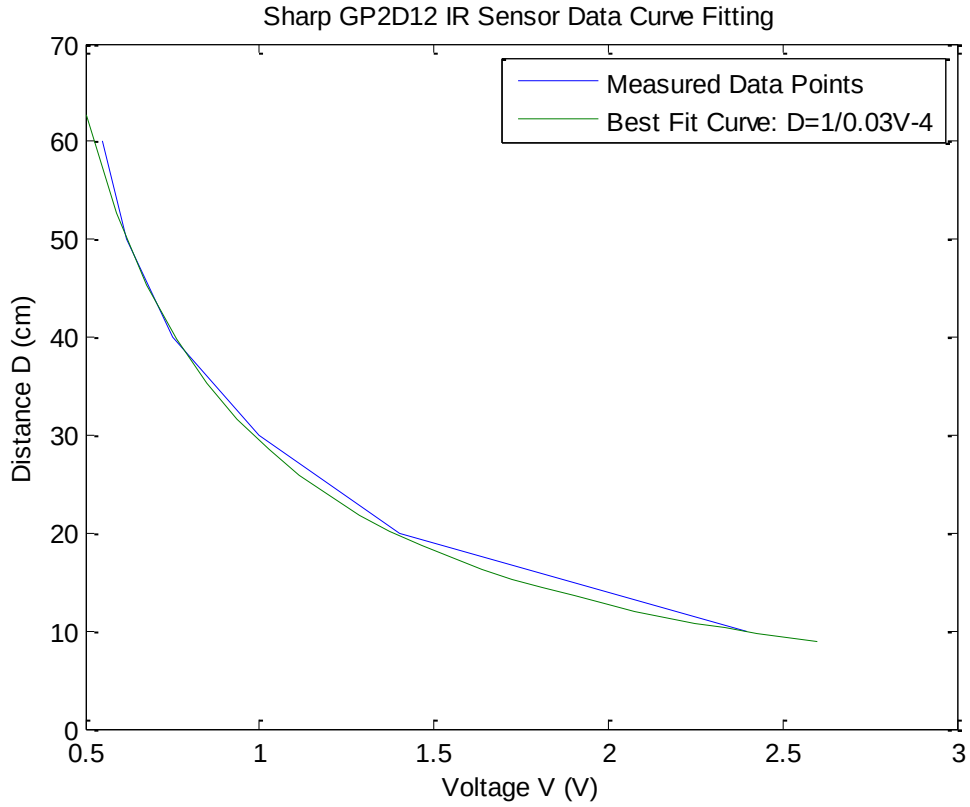


Figure 2: MATLAB Plot of the Sensor Curve Fitting

Two sensors mounted on front of the car were used to sense the distance to the object on the left and right sides of the car. The average of the two sensor readings gave the longitudinal distance from the car to the object it is tracking. A third sensor was mounted on the side as a safety measure to avoid collisions on the right side. When the third sensor was not there, and the car was approaching a long wall, it kept driving until it reached the desired distance. But if the angle of approach was too steep, there was no way to maneuver out of the position, leaving the car trapped. With the sensor in place, the situation was avoided, as the car will be forced to steer away from the wall. Ideally, a fourth sensor on the left side would have been implemented, but due to the lack of ADC ports on the luminary board (there were only four; one was used to measure voltage across the power supply, and the remaining three were used for the sensors), it was not possible without some extension to the board. The right side sensor was thereby placed to demonstrate a proof of concept application. It would have been trivial to extend the software to accommodate a left side sensor.

Figure 3 - Sharp GP2D12 IR Sensor

Steering and Drive Motor Control

Upon investigation of the RC cars that were available in the appropriate budget range, we realized that less expensive cars only had discrete turning, meaning there was only "steering" and "no steering." Only more expensive high quality hobby cars were sophisticated enough to have servo motors that allowed for fully proportional steering. Therefore, instead of using a standard PID algorithm to control the steering, we implemented a custom steering algorithm that, given the available sensor readings, made "best guesses" as to the angular orientation of the leading object, and followed those guesses through until it became clear (through sensor data) that the guess was no longer valid, in which case it made a new guess and repeated. The drive remained controlled by a standard PID algorithm.

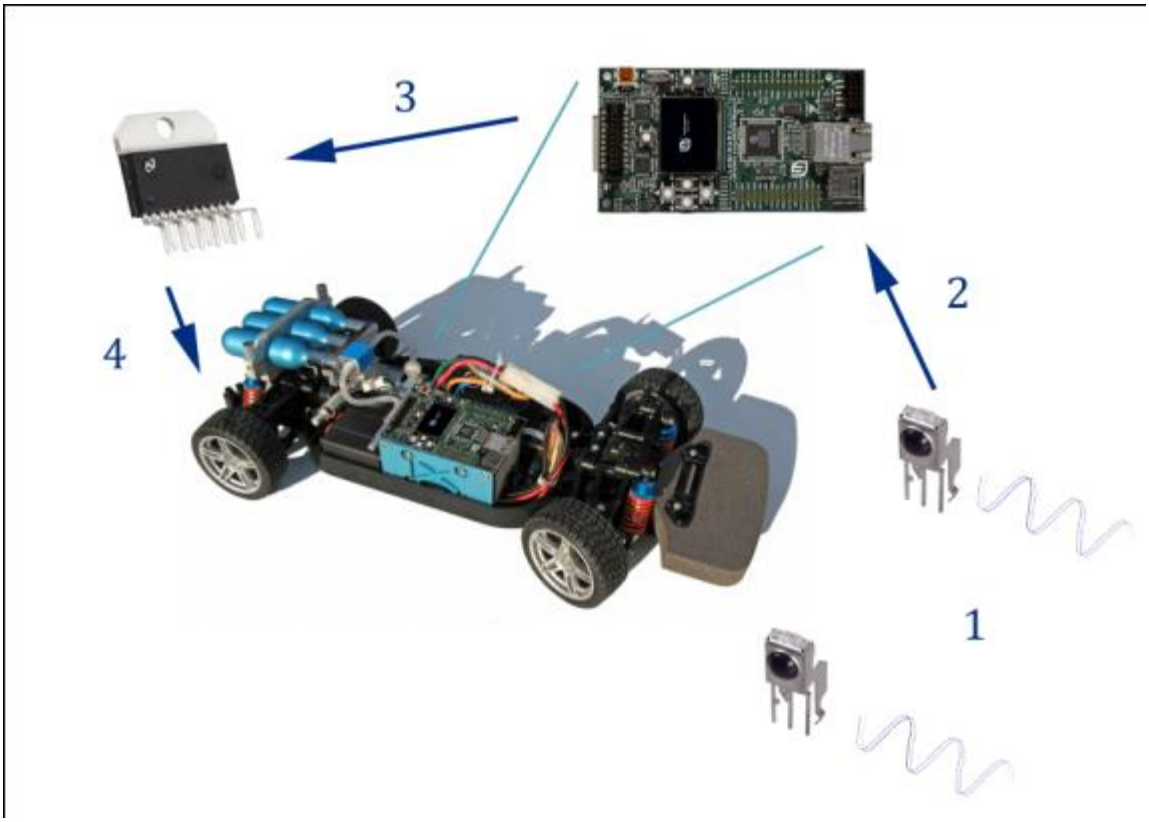


Figure 2: Conceptual Overview of the Actual Design (Lead Vehicle Not Shown)

1. Two front mounted, forward facing IR sensors collect distance measurements.
2. Both sensors output a voltage to the luminary board that corresponds to a perceived distance to the chase vehicle. The distance between the chase car and the lead vehicle is the average between the two sensor readings and that value is used to compute the error between actual and desired position away from the lead car. Additionally, the difference between the two readings is used to trigger steering if it exceeds a threshold value.
3. The luminary board processes incoming data and outputs a PWM signal to the LMD 18200 H-Bridge chip provided by the mechanical engineering department. The duty cycle of the output signal depends on the error and the PID controller algorithm. Steering and forward drive are controlled by two separate H-Bridges.
4. The H-Bridges take a given PWM signal with a certain duty cycle, direction and brake values as inputs[1] and output a usable voltage with which it is possible to drive a motor. The entire system is self contained and is fully autonomous.

The basic underlying principle of the steering control algorithm was that if the difference between the perceived distances of each sensor exceeded a certain threshold, then steer to correct it (taking the sign into account to steer the correct direction). See Figure for a more complete description. However, this causes the steering to be done in discrete steps instead of in a continuous motion. The implication is that the car will tend to stray away from the center of the target and move to the edge of the target. Another implication is that all turns are essentially identical. That is, whether or not the target is oriented at a shallow angle or at a steep angle, as long as if the orientation exceeded a threshold, the same turn would be made.

[1] We used a 7414 digital inverter as an output buffer for all the PWM and digital signals sent to the H-bridge.

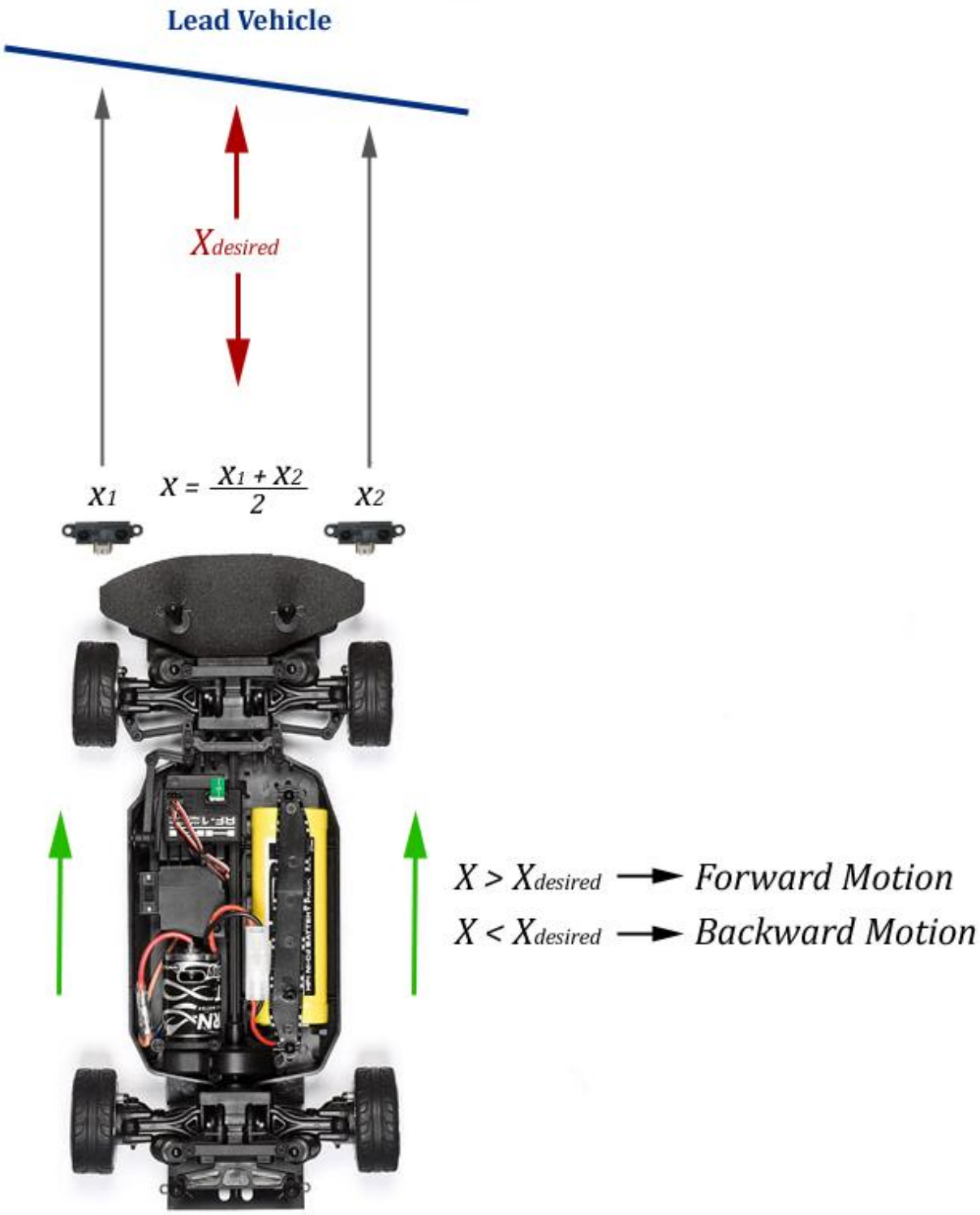


Figure 5: Drive Overview

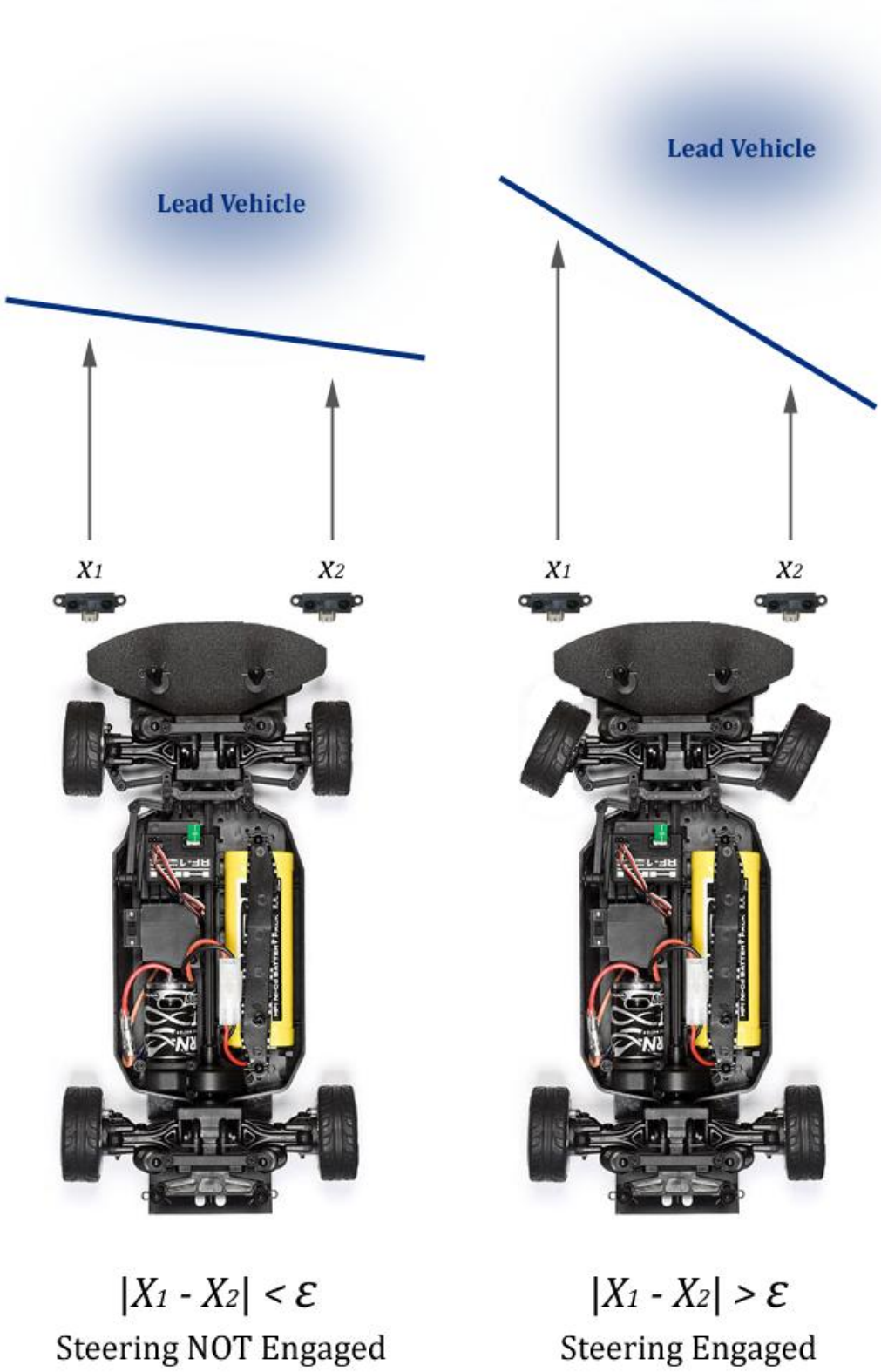


Figure 6: Steering Overview

Control Algorithm

The car features a basic PID (proportional-integral-derivative) controller. In order to tune our control gains, the car was first tested with a basic proportional control. This had a pretty good response to a step input. Then derivative control was added, which dampened out some of the oscillations. The integral control caused oscillations, so a small scaling factor decreases the integral contribution. However, to increase the steering performance, a faster response was needed, so some integral action was required. The integral gain also helped when there is an outside disturbance, when for example, the car attempts to scale an incline. A PD controller alone will have trouble scaling the hill, because the motor's torque is not enough to overcome the force of gravity. However, if an integral gain is used, it will integrate the errors until it the motor's torque becomes so large that it can scale the hill.

However, the controller was sensitive to fast acceleration and decelerations of the lead car. This could be accounted for because the microcontroller is a discrete time system with digital inputs and outputs, as opposed to a continuous, analog system of the simulation. The analog to digital conversion in acquiring sensor data causes a loss of precision. The choice of a sampling time is also important. If the sampling time is chosen to be too small, the derivative control would be too sensitive and vulnerable to noise. If the sampling time is too large, the integral action will be less accurate. The biggest problem was the derivative control, which caused the car to oscillate slightly at steady state, due to the derivative control being sensitive to sensor noise. Thus the sampling time was tuned for the smallest value that does not cause steady state oscillations. Then a scaling factor was added to the integral control.

Graphical User Interface

Our GUI was much less sophisticated due to the small resolution of the OLED and its memory intensity. However, much of the data we intended to put on the screen was unnecessary for our purposes, because we can run LabVIEW on debug mode and use its front panel. However, we wrote a basic OLED code for data that is of interest to the user, which includes indications for low battery and losing sight of the object. However, this feature also included a human machine interface for debugging without a computer. It also added an element of multitasking, because it ran a separate timed loop at a different sampling time than the PID controller time loop.

Error Handling and Safety Features

Various error handling situations were incorporated to prevent instability. There is the feature where the car shuts off if the sensors lose sight of the object. When this feature was first implemented, it caused the integral control to continue to integrate during the stopping time. As a result when the car re-detected an object, it ran at full speed. Thus we made a simple change in the software that disables the integration when the sensors do not detect an object.

Differences from Expected Tasks

Most of the other tasks that were not achieved were trivial and deemed unnecessary. Dynamical modeling of the RC car was not required. Much of the sensors in the original report were not necessary as the IR sensors provided enough data.

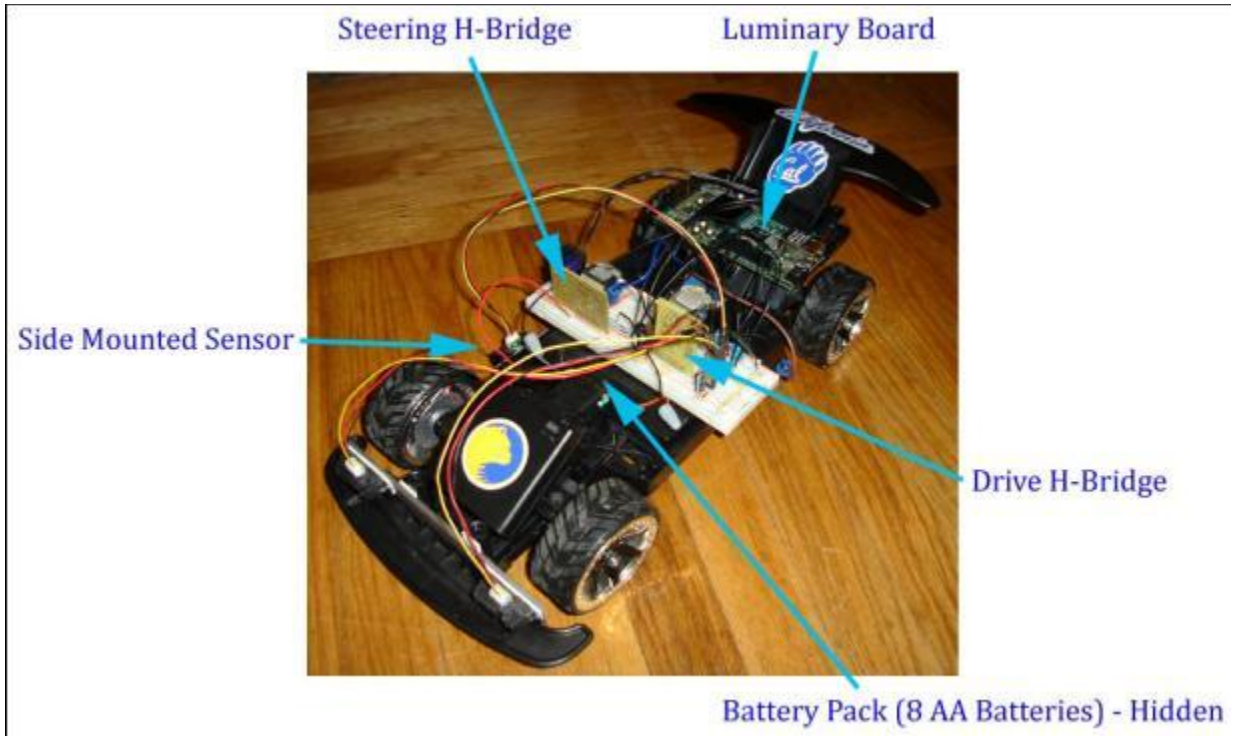


Figure 7: Top View

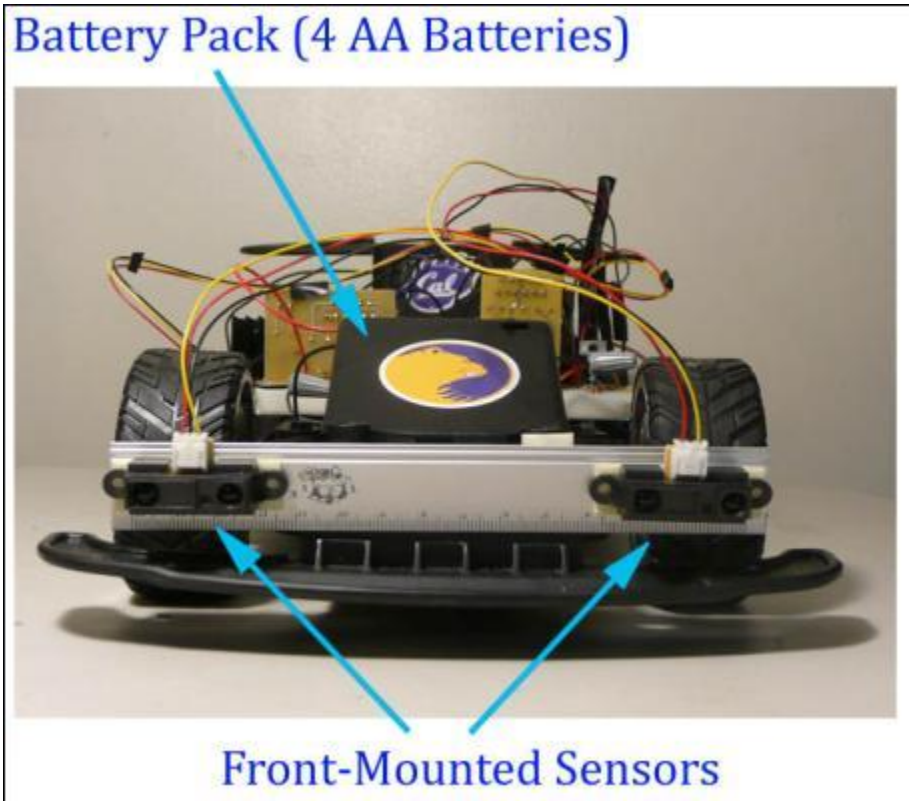


Figure 8: Front View of the CAL Autonomous Robot

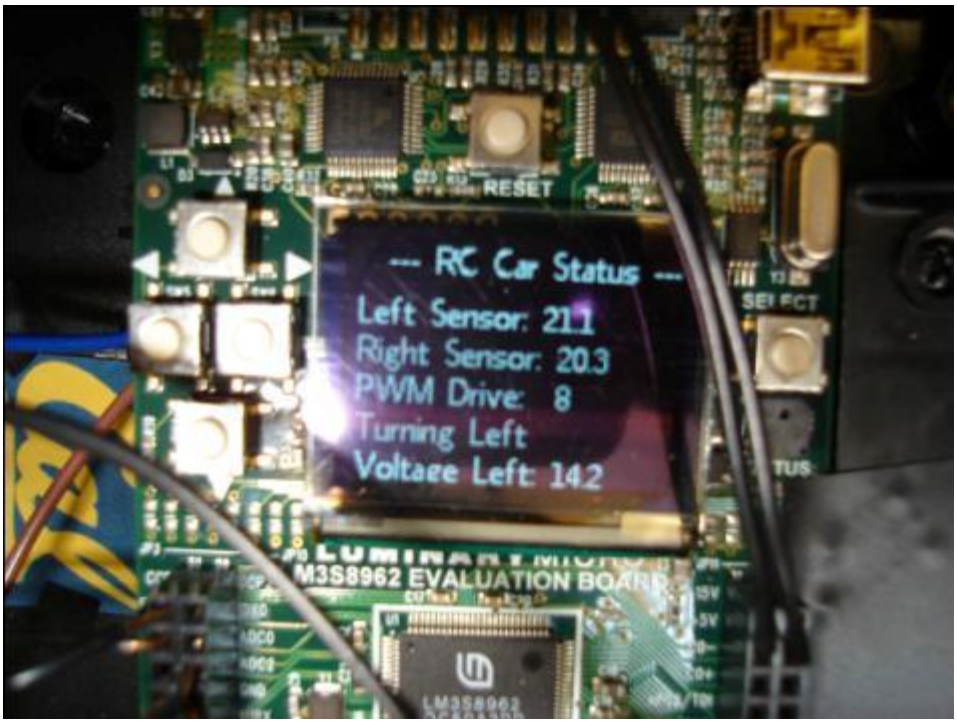


Figure 9: Graphic User Interface on the Luminary Board

Expected Vs. Actual Challenges

Going Cordless

At first we thought that making the car completely autonomous was going to be the most difficult task. There are two wires that connect to the microcontroller. There is a USB cable for its external power supply and another USB cable to interface with our computers to receive compiled binaries and to output real time data for debugging purposes. However, we eventually learned that you can flash the code onto the board, and the board can be externally powered using a 5V voltage regulator.

Wireless Communication vs. Hardware On Board

We originally proposed to control the car by actuating the joysticks on the remote control that came with the car, which we thought would be easier, and would have an added advantage of being able to control the car wirelessly from a distance. However, that option was deemed too complex, as the joysticks only pressed a pushbutton that controls the motor. When the button was pressed, the motor will only run at a set speed, which made variable velocities impossible. Furthermore, we found no simple and compact way to allow wireless communication between the Luminary Board and the actuators that would control the remote. These issues were quickly resolved when we interfaced the luminary board directly with the motors using H-Bridges. By using this method, we eliminated both the need for wireless communication and the inconvenience of using this remote control.

Power Supply

A problem with the voltage regulators was that the output voltage was not consistent when we were running the car from the same voltage source. Thus, we learned that we needed capacitors on the output to smooth out the noise and to store charge.

Although AA batteries are rated at 1.5V, the usual output voltage of each battery falls below that value after prolonged use. Consequently, the output voltage of our power supply varies with time. This poses a considerable problem for our PID controller. Since the H-Bridges are powered by a time varying power supply, a given PWM duty cycle might not produce the same average voltage across the motor at two different times. Consequently, if the constants of the PID controller are tuned when the batteries are at full charge, then by the time the batteries are drained, the PID controller cannot adequately control the car.

We tried to resolve this issue by placing a 12V voltage regulator between the power supply and the H-Bridges. Our hope was that this would eliminate the time varying property of our system. Unfortunately, our regulators allowed only 1A of current to pass through. While our system uses approximately 0.4 A at steady state, large current spikes (which provide a large amount of torque to the motors) are required for them to overcome their inertia when starting from rest. A result that we consistently observed using voltage regulators was that only one of the systems (either drive or steering) would work. Voltage regulators that are rated for higher current do exist. However, we were not able to obtain them.

A rejected alternative solution was to make the constants of the PID controller a function of the battery pack voltage. We implemented a method of measuring battery voltage. In fact, Figure 7 shows that we display the voltage remaining on the OLED display. However, the accuracy of the displayed value is questionable due to noise. Furthermore, the relationship that exists between time, battery voltages, and the PID controller constants is mostly likely a complicated one that must be empirically determined through rigorous testing. The time constraints of this project prevented us from finding that relationship. Consequently, this idea was abandoned.

Sensors

Our first choice of a sensor was an IR emitter and detector combo that utilizes phototransistors. However, we noticed that the transistors were very susceptible to external IR emission sources from everyday objects, like lamps and sunlight, so we settled on the Sharp IR rangefinder series, using the 2D12 model that suits our distance detection needs. In addition to be relatively immune to external noise, it also only detects the object straight ahead, without measuring the interference from the surroundings. This also has the advantage of being able to track any target, without having to mount IR emitters on the object.

The sensor had noise, which was not the ideal case of continuous and instantaneous data. However, we noticed that the sensors often dropped its output voltage to zero every 10 seconds or so. We were able to mitigate some of the noise using a 1st order low pass filter. Using RC circuits, each low pass filter had a time constant of approximately 50 ms. The time constant was tuned to balance out the noise and also to have a quick response from the car. This is because we also need fine resolution in the distance feedback data to have a faster controller. Also the sensors would read a voltage close to zero when an object is out of range. Thus we set a minimum voltage threshold to saturate the distance reading (recall that our voltage relationship is inversely proportional).

Sensor noise was often enough to trigger the steering, which sometimes caused instability because improper sensor readings forced the car to turn away from the lead car. However, at larger distances the noise becomes significantly greater. One improvement that was made was to vary the threshold (linearly) with the distance to the target car. The intuition behind this was that, the closer the lead vehicle is, the more sensitive the following car should be to orientation (partially due to the fact that the sensors perform more accurately at closer distances). Likewise, the farther away the lead vehicle is, the less sensitive the following car should be. This improvement, while seemingly trivial, actually helped quite a bit with the steering. At the very least, it made it possible to follow the lead car at farther distances without straying off course by turning too much (mostly due to sensor noise, which seems to be magnified at longer distances).

We also decreased our following distance to 20 cm, which allow the sensors to work in a regime closer to the object, which increases sensor accuracy. This was done by tuning the gains for a much more aggressive controller. Correspondingly, we increased K_d to dampen out the oscillations. This fixed our problem by making our car very responsive to changes in the lead car's position, ensuring that the car remains within 40 cm of the target for most of the time.

Because of the limitations of steering, the car will eventually stray down to the edges of the object it is tracking, which makes it more vulnerable to losing sight of the object altogether. Therefore we outfitted the lead car with long surface to allow for some room for error. However, this did not fix the problem, but merely delayed instability. Theoretically, we would need an infinitely long surface because the car would stray continuously over time. Complicating this issue is that sensor noise would affect the steering accuracy. We decided instead to make the surface on the lead car convex relative to the car, so that the car will correct itself. As the car strays toward the edge, the curvature will become greater, until it is large enough to set off our steering tolerance and cause the car to move back toward the center. This simple modification to the surface is able to prevent instability due to the car straying to the edge.

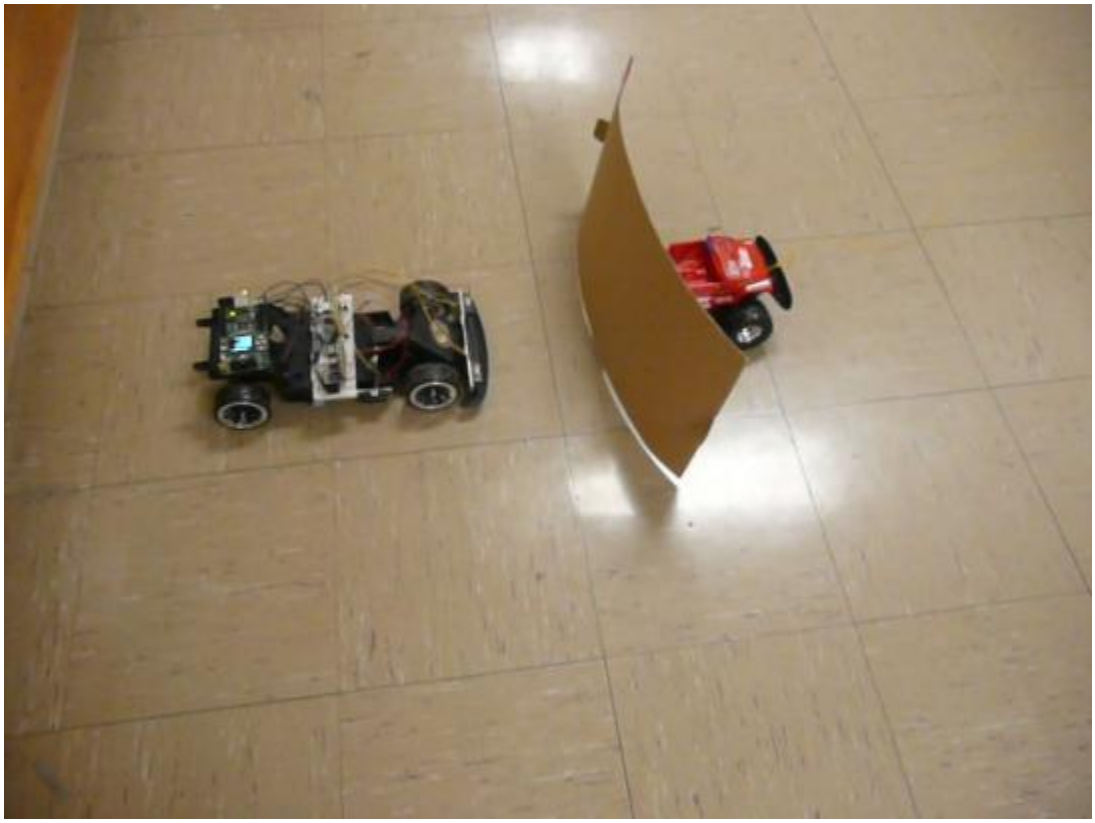


Figure 10: Chase and Lead Car (With Curved Surface Area)

Hindsight

Most of the difficulties arose from the hardware. But once we bought the hardware, we were committed to working problems around it. Our first car was the most inexpensive car on sale but it was poorly built and eventually broke. We spent a lot of time working out the problems, when a more rugged car would have saved us time. But when it eventually broke, we were relieved because we could buy a new car. This time around we had some experience, so we looked around for cars with proportional steering. But those were too expensive, so we bought a nicer toy car that would suit our power and size needs. If we had a chance to do this project again, we would spend more time researching RC cars and make a more informed choice the first time around rather than trying to get a hopeless car working.

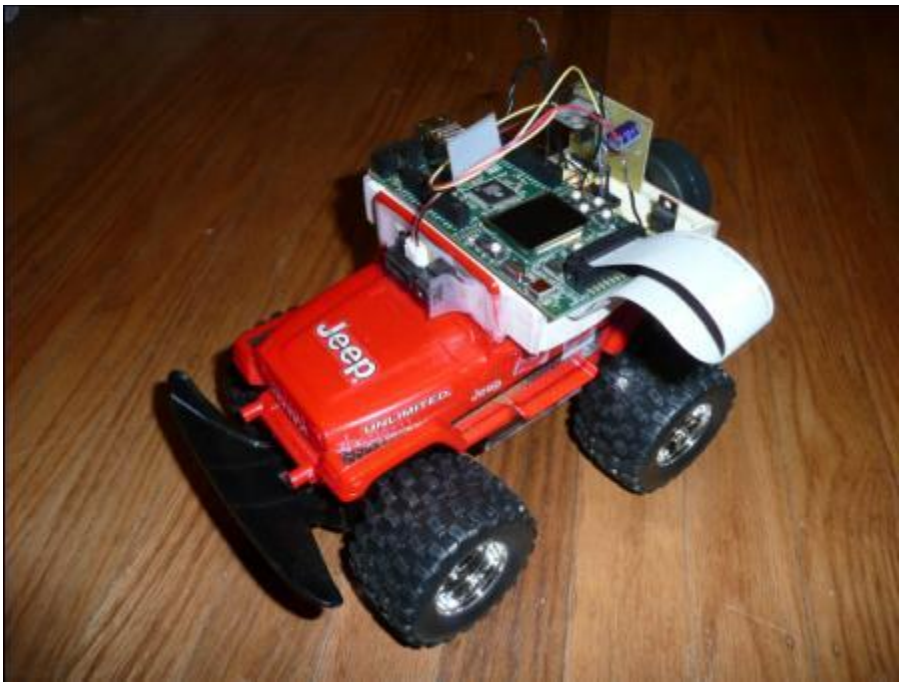


Figure 11: Failed Version, also the Lead Car in Figure 10.

Another thing we would change is to get a better microcontroller. The Luminary board has shared ports which are hard to use since they interfere with one another. Also, this limits the amount of total ports that are available to us, since we are using the LED screen, which shares a port. The amount of ADC ports is limited to four, which limits the amount of analog inputs we can use in our project. We managed to use all four of the ADC ports (two front sensors, right side collision sensor and battery voltage sensor). Our steering algorithm would work more effectively if we had collision avoidance at the left side as well.

Our sensors could also be upgraded, because they were affecting the turning ability. Turning instability is a function of object velocity and turning radius. If we had better sensors, the car could follow at a faster rate while turning, and be more immune to fast accelerations. We were only able to obtain hobby sensors, and their performance was enough to have a working car. However, if we wanted to improve our car for better performance we must find sensors with a faster sampling rate.

I think a big disappointment was our car's mediocre performance on the day of the presentation. Since we had less than a day's notice to switch the demo location to Hesse Hall, we did not have an opportunity to test our car. We assumed that Hesse Hall's floor surface would be comparable to that of Etcheverry Hall, where we had tuned our gains. On the day of the presentation, we noticed that the car's response was not as fast as it was the day before. During the presentation, when we really ramped up the speed and decreased the radius of curvature, the car lost sight of the box and ran off in another direction. On further inspection after the demo, we realized the floor tiling in Hesse was a lot smoother and dustier, which quickly covered our car's tires in a layer of dust. Thus, the car did not have adequate traction to run the aggressive response we needed to have accurate steering. Had we been notified that we were presenting in Hesse Hall with more than a day in advance, we would have tuned the gains in the actual room, or swept the dust off the floor.

Appendix

Circuit Diagram

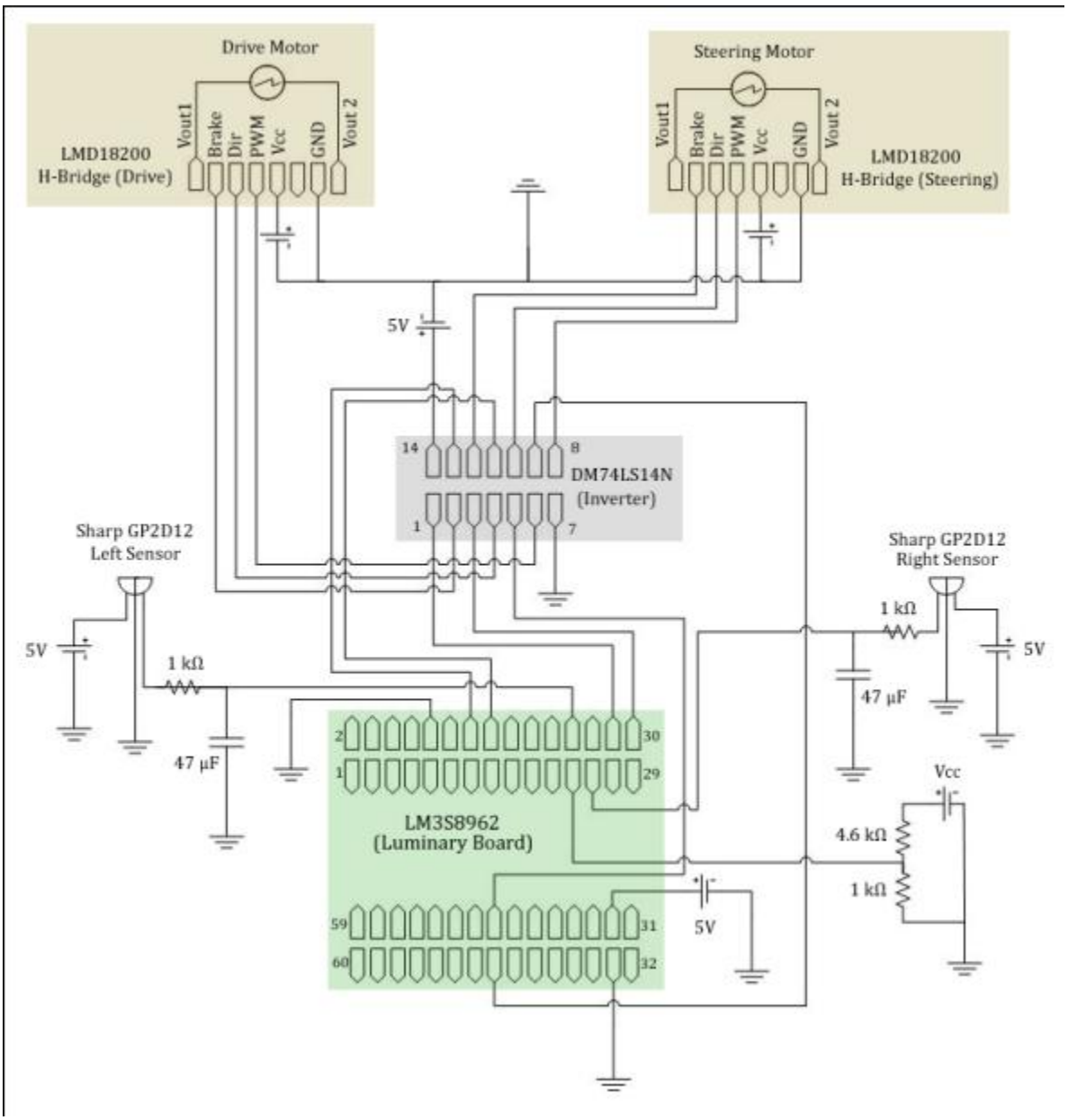


Figure 12: Hardware Circuit Diagram

Note: Vcc is the power supply voltage. The 5V power source was achieved by regulating down Vcc using a 5V voltage regulator.

LabVIEW Code

All the code written for the microcontroller was programmed using the LabVIEW dataflow programming language. To deploy the code onto the luminary board, the LabVIEW code was translated into C code using the LabVIEW ARM microcontroller extension, and then compiled and flashed onto the board using a JTAG emulator. The advantages of this were ease of development (rapid prototyping), and being able to use the LabVIEW debugger to step through code, which is very intuitive. At the top level of the LabVIEW code were two parallel while loops, one which handled the controller logic, and the other which handled updating the LED display. Code for both is shown below, with the control loop shown first.

Control Loop VIs:

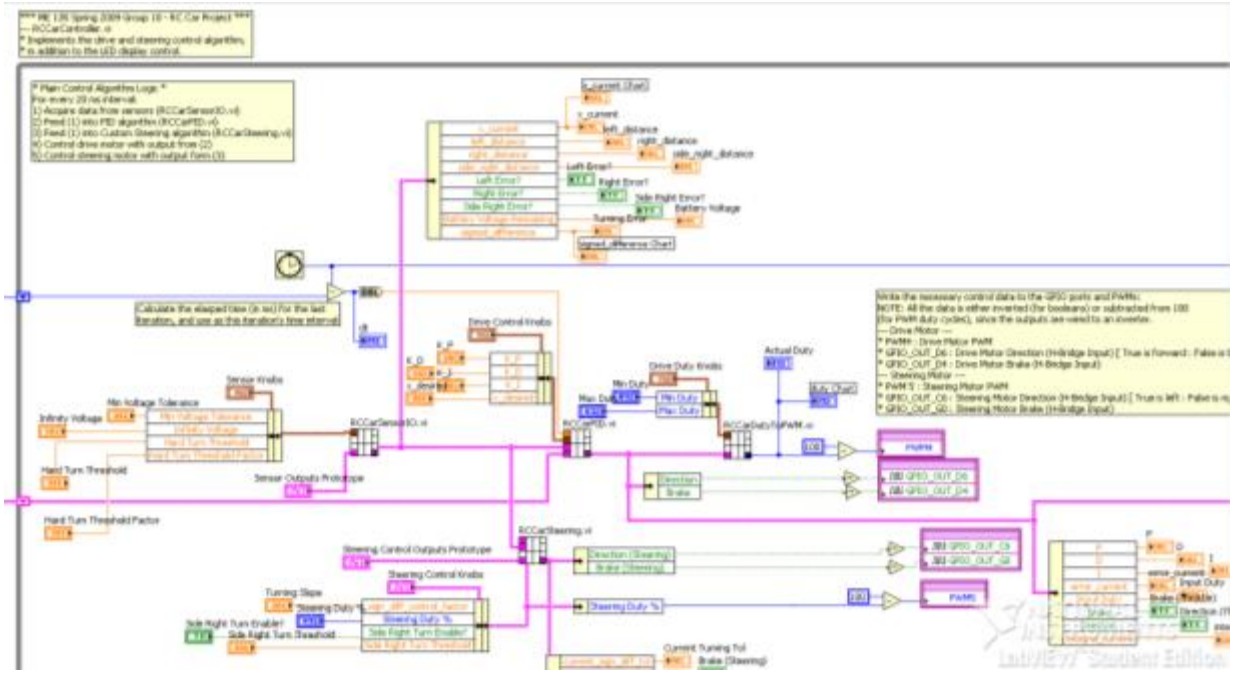


Figure 3 - RCCarController.vi [1st Parallel Loop]

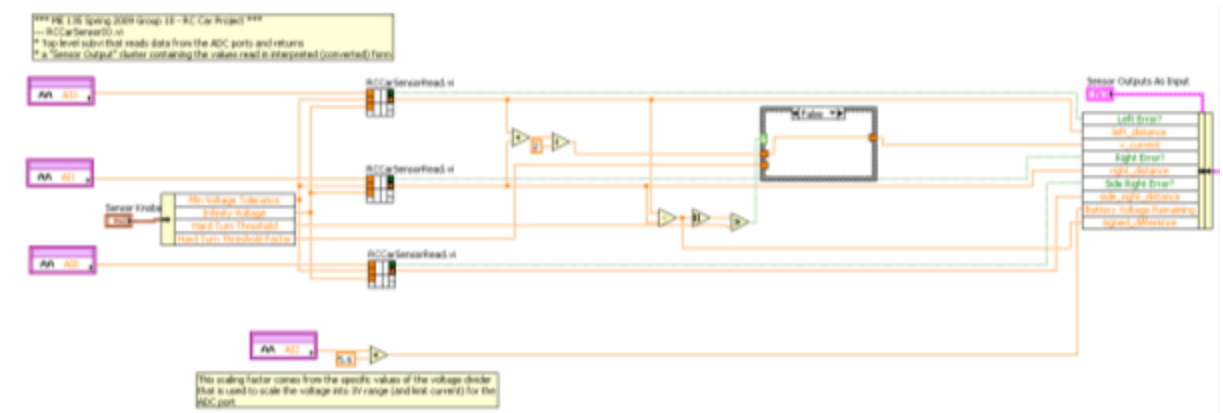


Figure 4 - RCCarSensorIO.vi

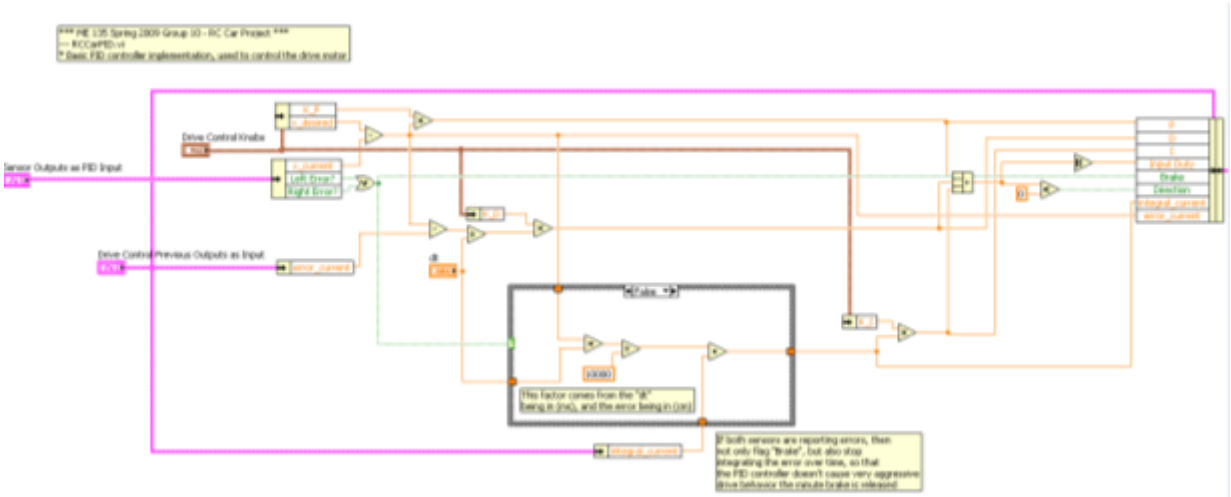


Figure 5 - RCCarPID.vi

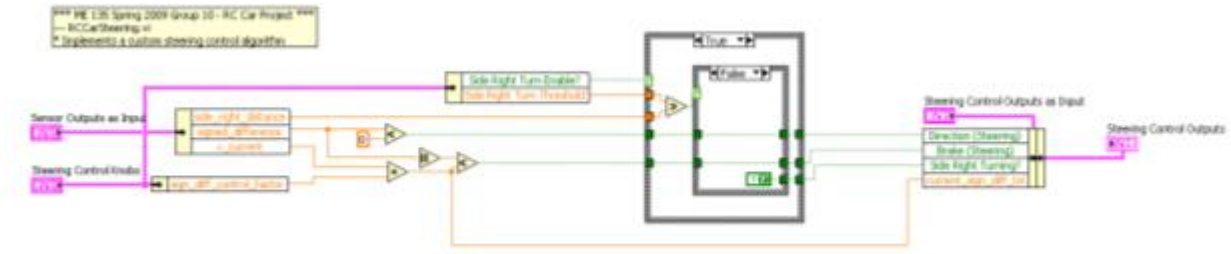


Figure 6 - RCCarSteering.vi

LED Loop VI:

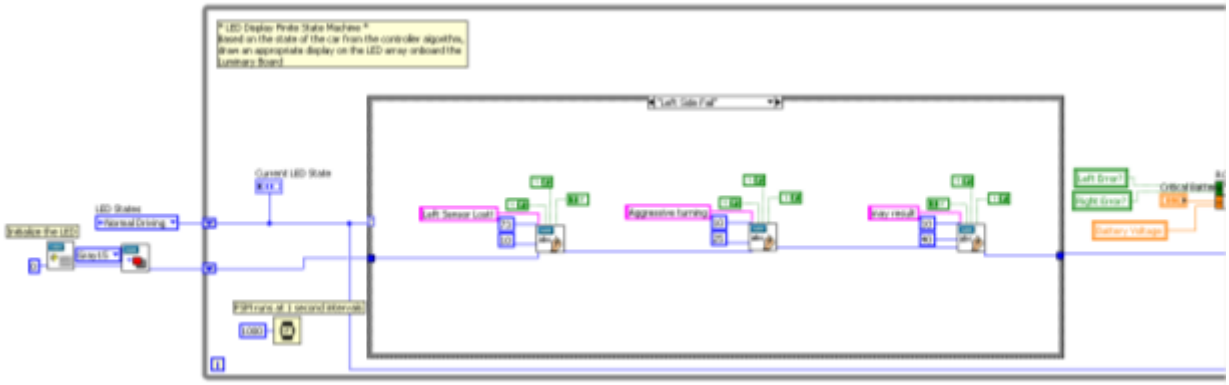


Figure 7 - RCCarController.vi [2nd Parallel Loop]

Pseudo Code

Below is a high level view of the code logic that was implemented on the microcontroller. As stated above, at the top level there were two separate threads (parallel while loops) running. One thread ran the control algorithm (for both drive and steering), and the other thread continuously updated the OLED display with the current state of the vehicle. First, here is the pseudo-code that governs the control algorithm thread:

Procedure 1 RCCarController()

```
1. Set previousState := initial conditions of vehicle
2. loop
3.       Set data := SensorAcquire()
4.       Set [driveOutputs,previousState]:= PIDController(data,previousState)
5.       Set steeringOutputs := SteeringController(data)
6.       outputsToPWM(driveMotor, driveOutputs)
7.       outputsToPWM(steeringMotor, steeringOutputs)
8.       Wait dt(= 20ms)
9. end loop
```

Figure 8 - RCCarController Pseudo Code. Corresponds to Figure 3

Procedure 2 SensorAcquire()

Output: data - data structure containing distances and errors for each sensor

```
1.  Set data := {}
2.  for each  sensor in [ frontLeft, frontRight, sideRight ]  do
3.      /* Read voltage from ADC port */
4.      Set voltage := ADCRead(sensor)
5.      if  voltage ≥ minVoltageTolerance  then
6.          /* Pre-determined voltage curve */
7.          Set distance := 1/(0.03×voltage) − 4
8.          Set sensorError to false
9.      else
10.         /* Sensor does not see anything */
11.         Set distance := InfiniteDistanceValue
12.         Set sensorError to true
13.     end if
14.     Map (distance,sensorError)-tuple to sensor and add to data
15. end for
16. Set data.currentPosition := average(data.leftDistance, data.rightDistance)
17. Set data.angle := data.leftDistance − data.rightDistance
18. return  data
```

Figure 9 - SensorAcquire Pseudo Code. Corresponds to Figure 4

Procedure 3 PIDController(data,previousState)

Input: data - data structure containing sensor data

Input: previousState - data structure containing values from the previous iteration of this algorithm

Output: driveOutputs - data structure containing the appropriate values to send to the drive motor PWM and H-Bridge

Output: previousState - outputs the values of this current iteration, to be used in next iteration as previous values

1. /* This is a standard implementation of a PID controller */
2. Set driveOutputs := {}
3. Set currentError := desiredPosition − data.currentPosition
4. Set currentDerivative := $\frac{\text{currentError} - \text{previousState.error}}{dt(=20ms)}$
5. **if** data.leftSensorError or data.rightSensorError **then**
6. /* Do not integrate error when sensor data is invalid */
7. Set currentIntegral := 0
8. Set driveOutputs.brake to true
9. **else**
10. Set currentIntegral := currentError$\times dt(=20ms)$+ previousState.integral
11. Set driveOutputs.brake to false
12. **end if**
13. Set duty := $K_P \times$ currentError + $K_D \times$ currentDerivative + $K_I \times$ currentIntegral
14. Set driveOutputs.direction := (duty < 0) ? forward : backward
15. Set driveOutputs.inputDuty := |duty|
16. Set previousState.error := currentError
17. Set previousState.integral := currentIntegral
18. **return** [driveOutputs,previousState]

Figure 20 - PIDController Pseudo Code. Corresponds to Figure 5

Procedure 4 SteeringController(data)

Input: data - data structure containing sensor data
Output: steeringOutputs - data structure containing the appropriate values to send to the steering motor PWM and H-Bridge

```
1.  /* Non-standard control logic due to lack of proportional steering hardware */
2.  Set steeringOutputs := {}
3.  Set steeringOutputs.brake := false
4.  if data.sideRightDistance < hardRightTurnTolerance then
5.      /* The car sees a wall on the right, turn left away from it */
6.      Set steeringOutputs.direction := left
7.  else if |data.angle| > turnControlFactor × data.currentPosition then
8.      /* The car sees a significant angular difference in the front, so turn in the appropriate
        direction */
9.      Set steeringOutputs.direction := ( data.angle < 0 ) ? left : right
10. else
11.     /* Any difference in sensor readings will be attributed to noise in this condition, so
        do not turn the car */
12.     Set steeringOutputs.brake := true
13. end if
14. return steeringOutputs
```

Figure 21 - SteeringController Pseudo Code. Corresponds to Figure

Procedure 5 outputsToPWM(motor,motorOutputs)

Input: motor - specifies the motor to control
Input: motorOutputs - data structure containing the data to write to the motor PWM and H-Bridge

```
1.  if motor is driveMotor then
2.      /* Limit the maximum duty cycle sent to the drive motor so it does not burn out
        */
3.      PWMDutyCycle(motor, max(motorOutputs.inputDuty,MaximumAllowedDutyCycle))
4.  else
5.      /* This is a discrete step-response, so no need to vary duty cycle */
6.      PWMDutyCycle(motor,ConstantSteeringDutyCycle)
7.  end if
8.  /* Write the brake and steering data out to the appropriate H-Bridge IO ports */
9.  HBridgeBrakeOut(motor,motorOutputs.brake)
10. HBridgeDirectionOut(motor,motorOutputs.direction)
```

Figure 22 - outputsToPWM Pseudo Code. Corresponds to logic found in Figure 3

And here is the logic for the LED loop:

Procedure 6 LEDDraw()

```
1.  /* Prepare the LED to display */
2.  LEDInit()
3.  Set prevState := initial state of vehicle
4.  loop
5.      Set currentState := Poll current state data from RCCarController()
6.      Draw current state on LED screen, using predefined templates
7.      if currentState does not equal prevState then
8.          Clear LED Screen
9.          Set prevState := currentState
10.     end if
11.     /* LED loop runs at a much slower rate than the control algorithm */
12.     Wait dt(= 1000ms)
13. end loop
```

Figure 23 - LEDDraw Pseudo Code. Corresponds to Figure 7

Concurrency/Multitasking

One of the goals of this project was to develop a fully multitasking program. Use of the LabVIEW dataflow programming model greatly simplified this process, as it clearly (pictorially) identifies execution dependencies and allows the developer to try and take advantage of this and run as much code in parallel as possible. In theory, one can think of each VI as drawing out a dependency graph, where a wire from the output of one block to the input of another becomes a directed edge in this graph. A simple parallel execution path would then identify all nodes in this dependency graph that do not have any incoming edges (i.e. through a topological sorting), and execute all those instructions in parallel. This makes most LabVIEW VI inherently parallel (including the ones associated with this project). For instance, our RCCarController.vi is set up so that once the data acquisition from the sensors is done, then both the drive control algorithm and the steering control algorithm can run synchronously. Neither algorithm depends on the output of the other, so this is theoretically possible without the need of any additional concurrency structures (such as semaphores). Whether or not LabVIEW actually implements its code generation in this fashion is beyond our control however.

One feature of the code that definitely runs synchronously is the LED display. The way that parallel while loops were used in RCCarController.vi forced the LabVIEW code generator to dispatch the two loops on separate threads (or however LabVIEW implements parallel execution). We were able to witness this multitasking when operating the vehicle. Data would continuously update on the LED screen at the same time as the car was busy acquiring new sensor data and running its control algorithms. This was necessary in our implementation, since refreshing the LED screen at the same frequency as the control algorithms ran would cause excessive overhead per iteration, and might not guarantee that each iteration actually finish within its set interval length.

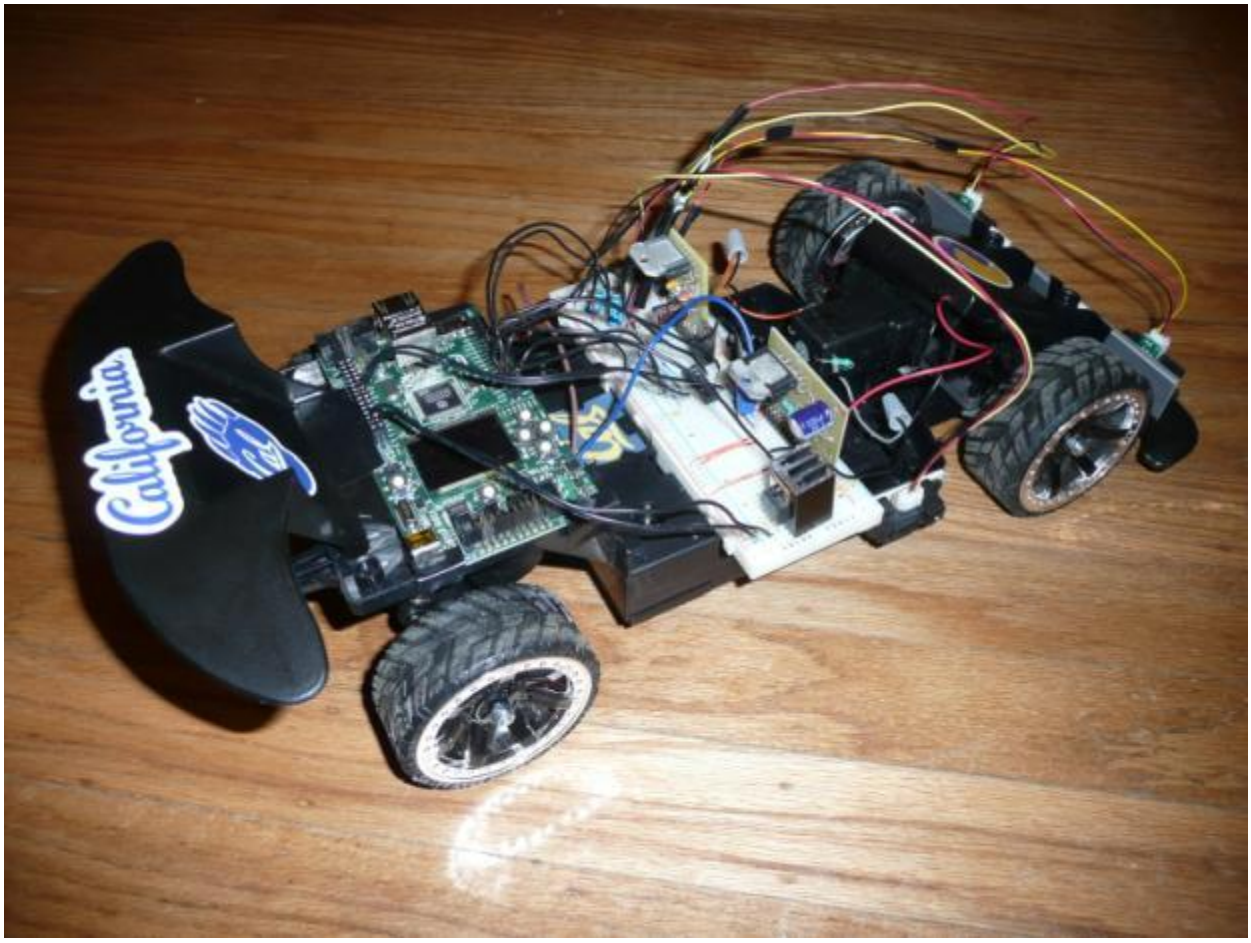


Figure 10: Side View of the CAL Autonomous Robot. Video can be found on http://www.youtube.com/user/calautonomousrobot

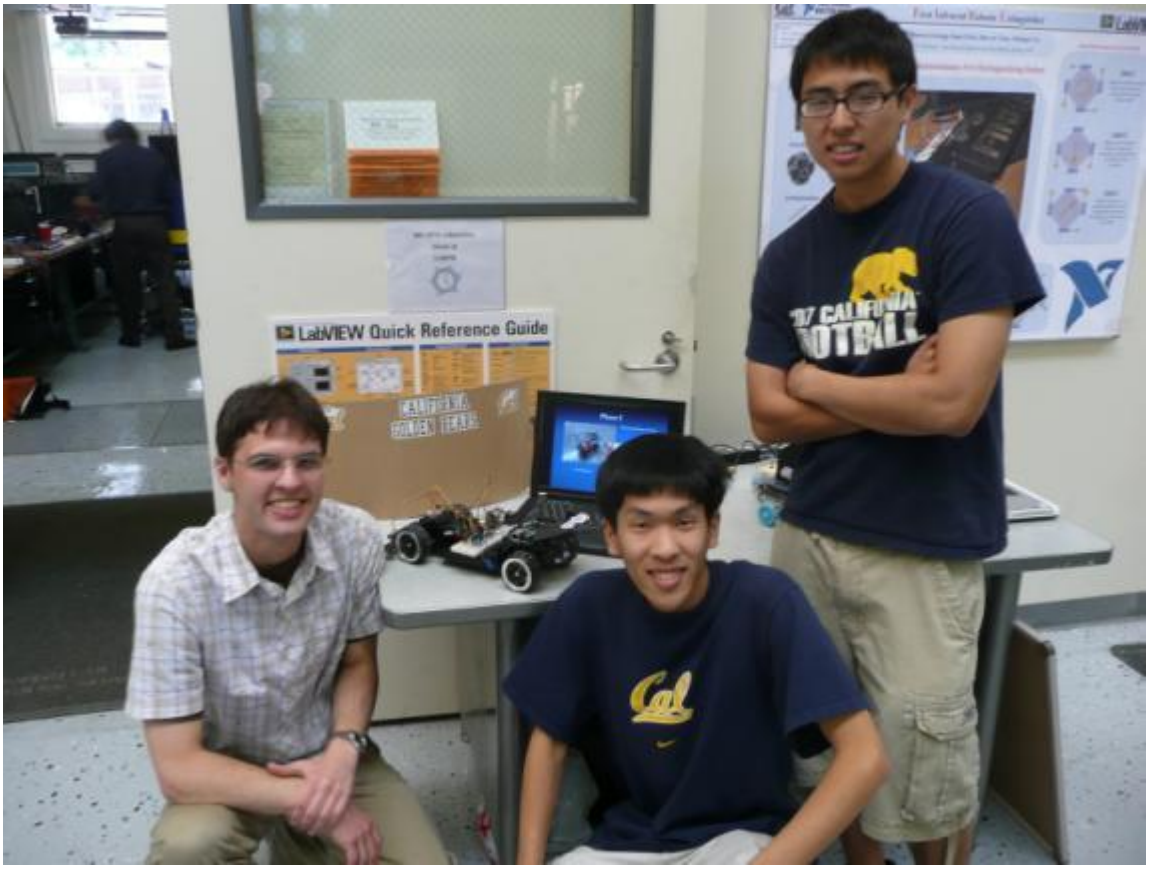


Figure 11: Project Team (Mikhail Podust, Ben Chu, Stephen Tu)