Project RoboCar

Team Members: Jhalak Gurung, Kabin Nam, Justin Park, James Smith, Allen Zeng
Graduate Advisor: Juan Duarte

Introduction

Project RoboCar is our knowledge-culminating final project for the EE16 series. The project tasked a team of 5 members to build a mobile robot on 3 wheels (2 drivable) that moved around according to some input command. It uses the MSP430 Launchpad as its guts with some circuitry for driving the motor and sensing through a microphone. In our version of the project, the microphone was used to recognize 4 different voice commands, which would then make the robot move forward (fast or slow), left or right. The project incorporated elements from circuit design, signal processing, classification, and control theory.

Circuit

The first part of the process was to create a circuit that could record the audio of the command the user was giving the robot. The circuit needs to convert the output of the microphone to a form that is compatible with the MSP430 microcontroller. This is called the front end circuit, and its output is fed into one of the inputs on the MSP430.

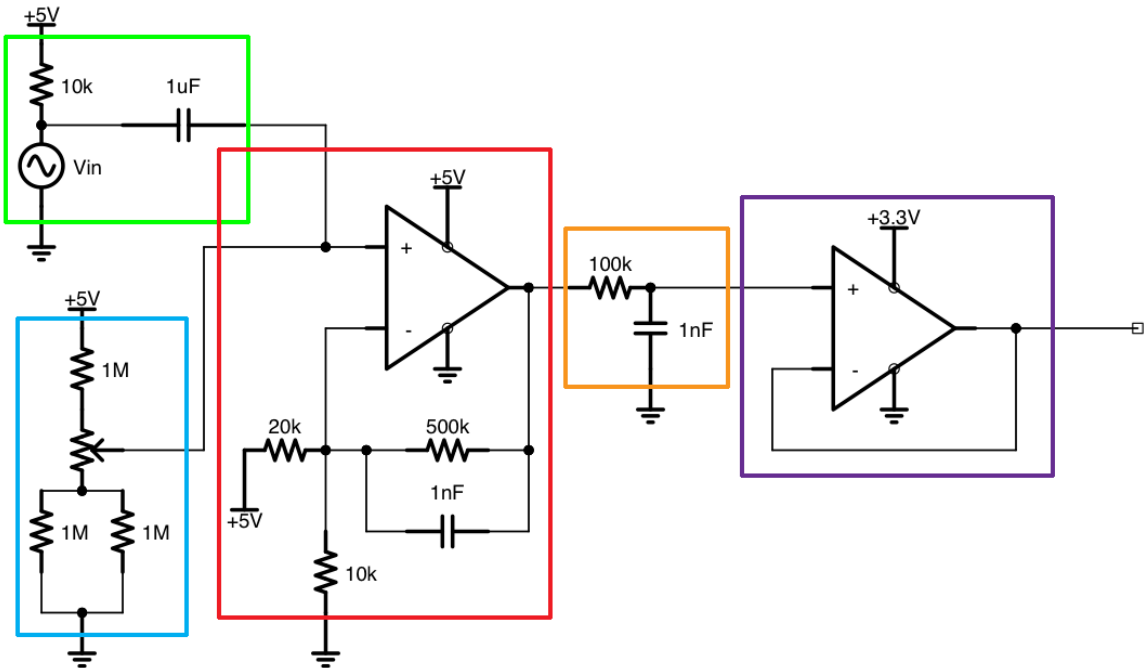


Figure 1: Front end circuit diagram

Green Stage

This stage of the front end circuit conditions the audio signal that comes out of the microphone. A pull-up resistor is added to the 5V DC line in order to create a bias voltage for the microphone. The

capacitor is also in-line in order to allow changes in voltage through into the op-amp input, hence the microphone. Our particular microphone operated in the 40mV range peak-to-peak.

Blue Stage

This stage essentially adds a DC bias and creates a high-pass filter out of both the green and blue stages that is applied to the input. We use large 1M Ω resistors to make sure our frequency cutoff is at most 10 Hz. In the end, this allows all of the input signals to be centered around 1.6V and the potentiometer allows for easy fine-tuning to prevent any clipping of loud input.

Red Stage

This stage amplifies and level shifts the input signal. This op-amp is set up as a differential amplifier. This helps boost our small voice input signal to a larger voltage. Since the MSP supports a voltage range of 0 to 3.3V, we set the gain of the op-amp to around 75 in order to boost the mic signal near the threshold of the MSP. In this case, by setting the bias voltage to around 1.65, we can set the amplitude from 20mV to 1.5V. This allows us to produce a gain signal from 0.15V to 3.15V.

Since we needed a bias voltage of 1.65V, we started off with a voltage divider on the op-amp inverting input with a 10k and 20k resistor.

$$R_b = \frac{20k \ * \ 10k}{30k} = 6.\underline{66}k$$

In order to achieve our 75 gain, we simply multiplied:

$$R_t = AR_b = 75 * 6.\underline{66}k \ \approx 500k$$

A capacitor is then added in parallel in order to eliminate any AC ripple in the inverting input so that our gain stays at a solid 75.

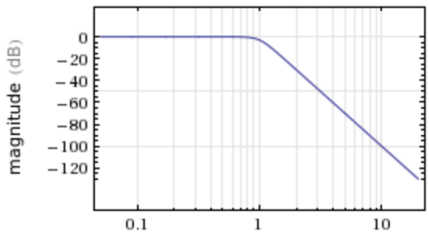
Orange Stage

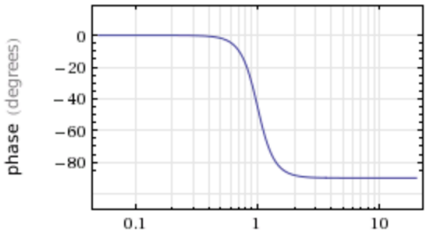
Simple low pass filter limited to around the frequency response of the human voice.

$$H(jw) = \frac{V_o}{Vi} = \frac{Z_C}{R \ + \ Z_C} = \frac{1}{1 + jwRC}$$

Let $w_c = \frac{1}{RC} = 10^6$

$$H(jw) = \frac{1}{1 + \frac{hw}{w_c}}$$





frequency frequency

Figure 2: Low pass filter bode plots

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(100x10^3\Omega)(1x10^{-9}F)} \approx 1591\ Hz$$

Purple Stage

This stage is a buffer op-amp to keep all signals within the 0 to 3.3V range to safely feed into the MSP430.

PCA Classification

Given a recording from the front end circuit, we need a way to determine if the recording represents one of the commands that we want to give to the robot. The recording is fed into a classification system that will determine the command word that it is closest to, and whether or not the similarities were close enough to constitute a match. Audio samples of the command words were transformed into the Fourier domain for analysis. The audio classifier was produced through a k-means clustering of PCA vectors extracted from the training word frequencies.

The two most significant factors in getting good word classifications were the number of syllables and the frequency content of each word. Words with different syllables would have peaks in different places, so when we calculate signal similarity using a dot product, we would get a larger total output if the syllables lined up. Two words with a different set of frequencies were good for the same reason. The words we ended up using were “stop”, “left turn”, “to the right”, and “go go go”.

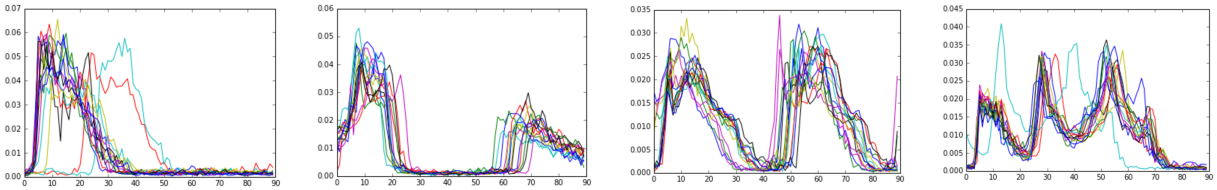


Figure 3: Recordings of the final 4 command words

Although the last two words, “to the right” and “go go go” had the same number of syllables, they had a different enough frequency range to be classified uniquely, although not as easily as the other two words. We found that it was difficult to predict word classification in general, even for words that seemed to have very different sounds. Two examples of words we had to throw out were “freedom” and “america”. It would seem that based on our criteria for what makes a word easy to classify that these would be good candidates, but we found that the system had a hard time distinguishing them.

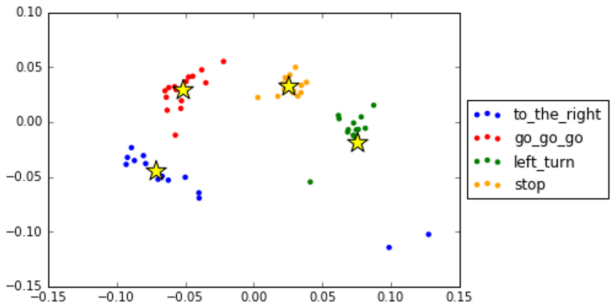


Figure 4: Final word classification clustering

To make the classifications more robust, we added a 3rd PCA vector to better differentiate “to the right” and “go go go” in case they were too indistinguishable using 2 dimensions.

Control

Once we have determined which command we want to give to the robot (via classification), we needed to create a control system that would allow for various movement types and also support a feedback mechanism that would allow the robot to drive strait. There are systemic biases in each motor of the robot, generally an artifact of tolerances during the manufacturing process, that will not allow the robot to drive straight if we just drive it open loop. Outputs are produced from the MSP430 to the motors on to robot to get the desired movement.

To find the open loop model, we first defined the state of the system as a vector $\vec{x}$:

$$\vec{x_w} = \begin{bmatrix} d_w \\ v_w \end{bmatrix}$$

Where w is either L or R depending on if it is the left or right wheel, respectively. Therefore, we can define the system as the following equations:

$$\vec{x}[t+1] = A\vec{x}[t] + Bu[t]$$
$$\vec{y}[t] = C\vec{x}[t]$$
$$A = \begin{bmatrix} 1 & T_s \\ 0 & 1 \end{bmatrix} \; B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \; C = I$$

The vector $\vec{u}$ represents the change in velocity that we want to apply to the wheel. Multiplying this equation out, we end up with the following system of equations:

$$d_w[t+1] = d_w[t] + T_s v_w[t] + b_1 u[t]$$
$$v_w[t+1] = v_w[t] + b_2 u[t]$$

This can be formulated as a least squares problem to solve for B:

$$u[t]B = \vec{x}[t+1] - A\vec{x}[t]$$

We then loaded the data for $\vec{u}$ and $\vec{x}$ taken from the `dynamics_data.ino` sketch. We used iPython to solve for the values of the B matrix for each wheel.

In order to get the car to drive straight, we need to add some kind of feedback mechanism. This state feedback design can be seen in Figure 2.

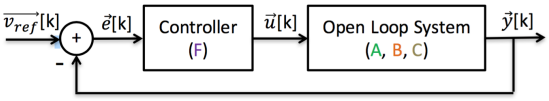


Figure 5: Closed loop state feedback

Where $\overline{w}$ is the **other** wheel:

$$\vec{v_{ref}} = \begin{bmatrix} d_{\overline{w}} \\ v_d \end{bmatrix}$$

Now we can redefine $\vec{u}$ and create a new state update equation:

$$\vec{u}[t] = F\left(\vec{v_{ref}}[t] - \vec{y}[t]\right) = F\left(\vec{v_{ref}}[t] - C\vec{x}[t]\right)$$
$$\vec{x}[t+1] = A\vec{x} + BF\left(\vec{v_{ref}}[t] - C\vec{x}[t]\right) = (A - BFC)\vec{x} + BF\vec{v_{ref}}[t]$$
$$A_{CL} = (A - BFC)$$
$$B_{CL} = BF$$

We want to be able to place the eigenvalues of A_{CL}, so we want $|A - \lambda I|$ to look like $(\lambda - \lambda_1)(\lambda - \lambda_2) = \lambda^2 + (-\lambda_1 - \lambda_2)\lambda + \lambda_1\lambda_2$.

$$\begin{vmatrix} 1 - b_1 f_1 - \lambda & T_d - b_1 f_2 \\ -b_2 f_1 & 1 - b_2 f_2 - \lambda \end{vmatrix} =$$
$$\lambda^2 + (b_1 f_1 + b_2 f_2 - 2)\lambda + (b_2 f_1 T_d - b_1 f_1 - b_2 f_2 + 1)$$

Equating coefficients:

$$b_1 f_1 + b_2 f_2 - 2 = -\lambda_1 - \lambda_2$$
$$b_2 f_1 T_d - b_1 f_1 - b_2 f_2 + 1 = \lambda_1 \lambda_2$$

We can formulate this as a matrix equation:

$$MF = \vec{d}$$
$$M = \begin{bmatrix} b_1 & b_2 \\ b_2 T_d - b_1 & -b_2 \end{bmatrix} \vec{d} = \begin{bmatrix} -\lambda_1 - \lambda_2 + 2 \\ \lambda_1 \lambda_2 - 1 \end{bmatrix}$$
$$F = M^{-1}\vec{d}$$

This allows us to find our controller gains F for any eigenvalues we want. We placed the eigenvalues through a process of experimentation. We found that eigenvalues that were too close to 1 would allow some of systematic biases of the wheels to still influence the car, while eigenvalues that were too small would begin to exhibit oscillatory behavior. We ended up choosing $\lambda_1 = \lambda_2 = 0.92$

The modification to the controller to allow turns was to introduce a new gain value that would be set to 0 or 1 to enable the motion of each wheel. This shows up in the control graph as a new node that the output gets sent through, that will multiply the entire output by the gain. If the gain is 0, the wheel will not spin, thus allowing turning in that direction. These gains are changed as we decide to turn in each direction.

Conclusions

One thing we learned in particular is that tuning the eigenvalues for the closed loop feedback was more of an art than a science. It took us quite a while to even understand what a valid range for the eigenvalues should be. We knew that our controller was working because we could see it trying to correct itself when we would squeeze a wheel, but we were seeing very bad oscillations for a long time. The problem that we were having is that for the most part we were making too large of changes to the eigenvalues between test runs.

Much like placing the eigenvalues, sometimes it was a bit of an art to determine which words would classify well. We optimized this by making it a very streamlined process to record a new word and determine how well it would classify. Being able to iterate quickly on trying new words was a critical feature in the success of our project.

A change we made to our iPython notebooks that ended up saving us a lot of time is to write a cell that prints out the values that we would need to bring into the MSP430 code as valid C code. Then we no longer had to manually copy over several values, or read them off to one another. We could just grab the entire cell output and copy it to the right place in the MSP430 code.

When developing the front end circuit, we found that we could pretty easily debug each part of the circuit individually and see expected results, but when we would put the various stages together, we ran into problems. It was critical for us to be able to understand how and why all of these parts fit together, and to know what kind of values we expected at the junctions between each stage. Once we figured that out and could dial these values in, the front end circuit came together quickly.

Advice we would have for teams creating this project in the future is to fully understand what you are working on. Have an idea about what you expect to see, and meticulously keep track of what you have tried so far. Only change one thing at a time in between tests, and have an efficient workflow. Don't be afraid to challenge each team member's ideas, as it will lead to better understanding for everyone and prevent bad assumptions.