EECS 16B Hands-on Lab Report FA21

By Samuel Clark(3035783100) and Yunhao Cao(3035915063)

Prelude

In Fall 2021 semester, we as a group finished the 16B lab project: SIXT33N small car. In this project, we applied multiple concepts learned in EECS 16B class to this project, including using unity gain buffers to supply half rail voltages for the mic board, using capacitors as a buffer for circuit safety, applying low-pass filters to filter out noise that does not constitute voice commands, and using principal component analysis to both store and classify voice commands. We found ourselves amazed by the opportunities to put hands-on practice with what we learned in class and while we are concluding on what we learned during the lab sessions, want to also express our gratitude towards the staff who all made this possible!

Part I. General Lab Questions

1. The oscilloscope is probably the most important and frequently used equipment throughout all lab weeks. It can be used to view circuit waveforms(to measure peak-to-peak voltage, average voltage(center of waveform) as well as static voltages(max voltage).
2.
    a. It is called a "decoupling" capacitor, it helps to reduce noise on the power supply (possibly imposed by other subparts of the circuit) and stabilizes the voltage fluctuations (caused by large resistance increase or sudden introduction of power loss, etc.).
    b. Think of the decoupling capacitor as a "short" for AC circuit (impedance is non-zero) but as an open circuit for DC circuit.
3.
    a. We built the band-pass filter by combining a high-pass filter, a unity gain buffer, and a low-pass filter in that order. Specifically, our high pass filter was a CR filter, and our low-pass was an RC filter. The unity gain buffer prevented current from flowing back through and charging capacitors in the CR filter with voltage in the RC filter.
    b. The unity gain buffer was introduced to eliminate the effect of load of the LED that follows the filters. To mitigate the effect of a low-pass filter after the high-pass filter, we used a relatively low impedance low-pass filter after the high-pass filter instead of using an unity gain buffer.
    c. The cutoff frequencies were 994.7183943243459 Hz for high pass and 3978.8735772973837 Hz for low pass, making the low pass cutoff about four times the high pass cutoff. It makes sense that the high pass cutoff is lower than the low pass, because this ensures that only values between the high cutoff and low cutoff are accepted, which in our case is approximately between 1000 and 4000 Hz.

Part II. Front End Circuits

Summary: In this lab, we used the low pass filter from the color-organ as a starting point for our left and right motor encoders. We added a 5V and a 3.3V voltage regulator to turn the 9V from our batteries to 5V, and then the 5V to 3.3V, building these by calculating how to position voltage dividers, resistors, and capacitors. With this complete, we started work on our mic board, tuning it by adjusting the gain and then modifying our low-pass filter to work properly with it and capture the correct range of frequencies that corresponded with the vocal commands we would later select. Finally, we moved on to the motor controllers, using BJTs and diodes to properly power our motors (both left and right, separately) and control them with a single switch. Thus, we had done most of the circuitry needed for our car, with sound as an input and motor movement as an output, and were ready to find out what inputs to take and what precise motor outputs to produce.

1. The buffer keeps the rest of the mic output circuit from affecting the micboard output voltage, so it avoids the rest of the circuit "loading" the output of the microphone.
2. We connect the OS1 with a 100k resistor because we cannot make the signal node equal to 1.65V, the resistor has to be 100k here because it would reduce load to the signal (for it to be big), while the decoupling capacitor in the front with this resistor forms a high-pass filter, so we have to choose it to be small enough so that the cutoff frequency for high-pass filter is only at 1.59Hz, the best option thus is, 100K.
3. We want to provide a virtual ground for the amplifying op-amp so it does not amplify the middle 1.65V.
4. Because we only have access to a voltage range that is non-negative, but the mic produces both positive and negative voltages, so the op-amp would not be able to produce the amplification.
5. We use the oscilloscope to measure the frequency response of a system. The response of the microphone system looks like a soundwave centered around 1.65V and maxed at 3.3V.
6. The cutoff frequency is the frequency in which the filters would produce an output magnitude of $\sqrt{2}$ of the original signal, it acts as the starting point where the filter really starts cutting off the magnitude of the input signal by a log(decrease in magnitude) to log(change in frequency) scale.
7. The human voice is in the low frequencies range and we want to get rid of the high frequency noise while listening to voice commands.
8. A PWM signal is a pulse-width modulation signal. It is a method of repeatedly switching on and off the voltage to produce an analog-like signal. But instead of controlling what the actual output voltage is, a PWM signal controls the percentage of times when the output voltage is at high and percentage of times when the output voltage is at low, and this allows very simple hardwares to be able to control the average power delivered through the output node. The duty cycle describes the percentage of "on" times for the PWM signal. And in our specific project, the higher duty cycle, the faster the motor on our car runs.
9. There are two reasons
    a. The motors we use do not operate at the voltage that our launchpad operates in(3.3V)
    b. Even if the motors were to operate, the launchpad would not have been able to support the magnitude of current that the motors draw. (Launchpad only supplies 100mA, while motors require a lot more)

10. The BJT acts like a transistor that switches ON and OFF as the output pin of the launchpad changes voltage, allowing the motor to flow current through the BJT and eventually reach to ground while the output pin is pulled up to HIGH, also disallowing the motor to flow its current to ground while the output pin is pulled down to LOW. The base resistor of the base terminal determines how large the current is flowing into the base terminal and determines the current that can flow between Collector and Emitter terminal. So the smaller the base register, the faster the motors will run.
11. Motors can act as HUGE inductors so when the power is suddenly disallowed from running through the BJT, a huge voltage will be built up on the Collector terminal of the BJT and that could cause serious damage such as breakdown of the BJT circuit.

Part III. System ID

Summary: This section of the lab required that we install sensors and encoders to the basic model of the car which we had just created. We began by placing in the encoder/photointerrupter, building voltage dividers as necessary to reduce the voltage of a high signal from an encoder. We then tested to ensure our encoders worked, and got ticks only when the wheel was on. Then, we allowed our system to be run on battery power by placing wires from two nine volt batteries to the input as opposed to running the power from the USB connected to the launchpad. We then collected data by running our car on the ground and seeing what tick values it produced for each encoder, generating a set of raw data which we can use to track the velocity of the wheels, and then a set of fine data we can use to perform a least-squares regression. Calculating with this regression generated an operating point: a velocity value we want both of the wheels to be moving at to get our car to move straight. This value was used in part IV as the v^* for open loop control.

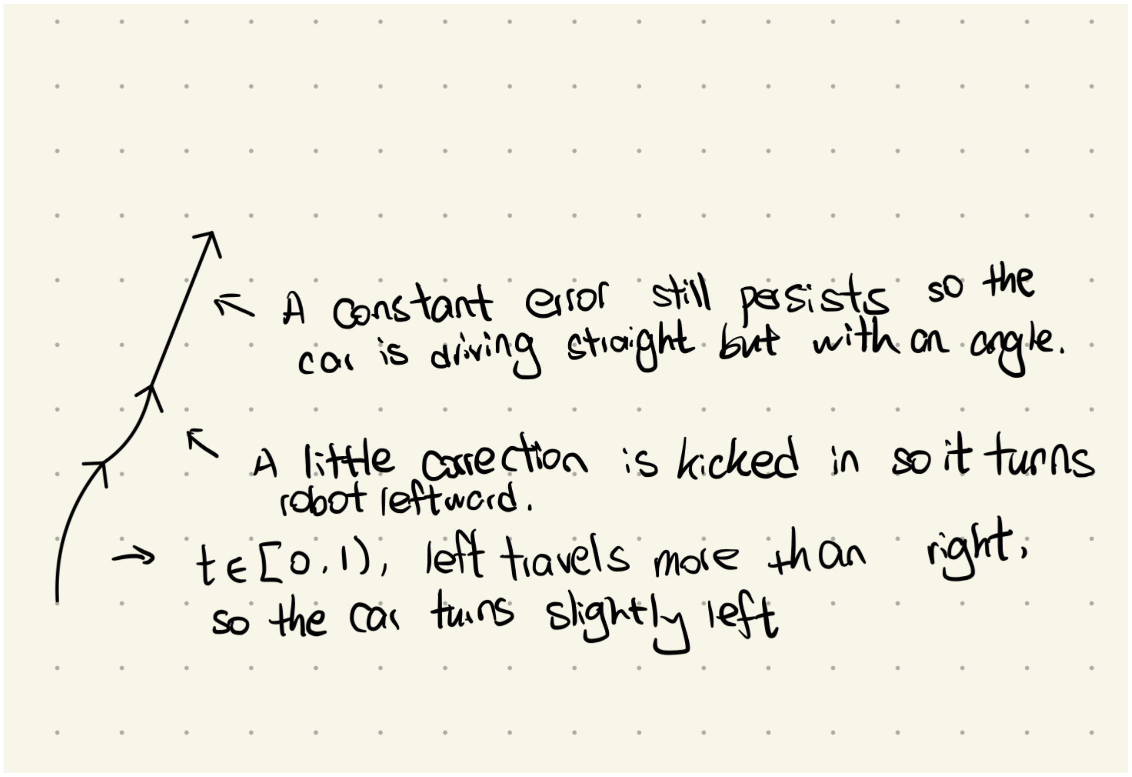
1. Encoders work by having a wheel with spokes and empty spaces in between, and a light shining through the empty space to a sensor on the other side. Whenever a spoke passes by due to the wheel turning, the light is interrupted, and one tick is counted by the encoder. Thus, the encoder counts the number of spokes that pass, thus keeping track of the wheel as it turns and knowing what position it is in.
2. We used a small region of PWM where the PWM and velocities are quite consistent. This was very important because the region where the PWM and velocities are not consistent is probably where the motor is not powered enough to start rotating. And since we will only be using the midpoint of our selected range, the other regions that we didn't select wouldn't affect our parameters found by least squares since we won't be likely to operate in those PWM ranges later anyways.
3. Theta and Beta parameters give us a way to solve for a 2D line. In theory theta parameter means increase in encoder ticks per second per PWM increase and -Beta means encoder ticks per second when output is 0V .
4. Because the motors are not completely perfect so we want to use separate theta and beta values to try to account for those imperfectness.

5. It is safer so that when we want to use a delta PWM to correct our error in encoder readings we wouldn't be using a PWM value that is outside of our selected region.
6.
    a. We only thought about working in our selected PWM range, but to account for the real physics model of both motors, we can use polynomials to interpolate(or simply connect the data points to form a line-section graph) our PWM to velocity behavior.
    b. Instead of selecting an area that looks stable and then compute jolt, use jolt as the starting point of the interpolated function on the x axis and interpolate the function, any velocity lower than jolt should be discarded out of our selection.

Part IV. Controls

Summary: In this lab, we implemented open-loop control and then used our open-loop measurements to find the proper closed-loop control for straight movement. We start with a simple open-loop control equation, and then run a simulation on how that would cause our car to behave. We then calculated our jolt values, and established open-loop control. With this established, we then worked on closed loop control, finding the f_left and f_right values by which to modify our open loop controls by running our car with them and seeing what made it travel the straightest. Lastly, we added delta_ss as a steady-state error correction, which basically allows for the car to travel straight in the same direction as it started moving, not just straight eventually in some direction.

1. We have u_L = (v^* + beta_left) / theta_left and u_R = u_L = (v^* + beta_right) / theta_right for open loop control, with u = [u_L, u_R]. Beta is a constant offset to velocity measured in ticks/timestep that accounts for things like static friction, and theta is a "sensitivity factor" in ticks/(timestep * duty cycle), representing the slope of the velocity versus u graph.
2. Open loop control fails because it cannot adjust its inputs during operation, instead relying upon predetermined inputs. Thus, if it is off at all, the car cannot adjust itself back on track to be straight, and will instead follow the same trajectory that it would have, since its inputs are not adjusted. Closed-loop control allows for inputs to be adjusted during operation, so it is necessary to correct any deviations and keep the car traveling straight through these adjustments.
3. Closed loop control has the equations:
   u_L = (v_star - f_left * delta + beta_left) / theta_left and
   u_R = (v_star + f_right * delta + beta_right) / theta_right
   Where v_star is the desired velocity, the theta values are the PWM inputs that make up the "sensitivity factor" as described in question 1, the beta values are the constant offsets to velocities in ticks/timestep for things like static fraction, the delta value is the control variable that measures the difference in the distance travelled between the left and right wheels, and f_left and f_right are the feedback terms used to adjust each wheel.
4. The system eigenvalue is 1 - f_left - f_right, meaning that the system is stable when $f_L + f_R < 2$ and $f_L + f_R > 0$. The second condition will never fail, because we must choose positive f values.
5. You know if the system eigenvalue is positive or negative in practice based on which direction the car turns--if the car begins to turn right if positive, and left if negative. So, when the car is turning right it is positive and when it is turning left it is negative.

6. Setting both f values to 0 simply produces the equation for open-loop control, making there be no adjustment to the inputs. Non-zero f values act as a coefficient to the delta value, allowing us to use data from operation to adjust our inputs.
7. If we were to use negative values of f then we are actually amplifying the error. If we were to use negative values of f we would have to put negative signs in front of the thetas.
8. Zero delta ss means the trajectory would be straight while non-zero delta ss means in the end of the trajectory the trajectory would look like it is curved. So when we add the appropriate delta ss value into the code it would correct for constant errors.
9. 
A constant error still pesists so the car is driving straight but with an angle.
A little correction is kicked in so it turns robot leftword.
t ∈ [0,1), left travels more than right, so the car turns slightly left

Part V. SVD/PCA

Summary: In this lab, we recorded a compendium of sound files to get our launchpad to "recognize" certain words, collecting data points from them, removing outliers, and processing that data to find an algorithm that can recognize certain sounds. To do this, we selected a length, pre-length, and threshold (see question 1 below) to correctly capture the data points in a particular category, and then applied Principal Component Analysis to find a new basis for our data points, and projected onto this basis so that we can separate our data into clusters corresponding to each sound. We then used these clusters to find centroids for each cluster, using this to estimate if a given input was "close enough" to a centroid to be identified as that sound input. In the end, we tested these classifiers by speaking our commands into the mic board and seeing if they were correctly identified, going back to our data and PCA analysis if any errors were found to ensure that our commands reached at least 80% accuracy in recognition.

1. The **length** is the window of how long a piece of data is, which means we capture the sound that occurs over some length of time. The **pre-length** is the amount of samples between the start of the command and passing the threshold, and the **threshold** is the value which, once crossed, marks the start of a speech command. In other words, anything below the threshold is not significant for our data collection, and crossing the threshold marks the point of importance. In our final lab, we ended up with a length of 80, a pre-length of 5, and a threshold of 0.5.
2. We must process our data before running SVD/PCA because we need to ensure our data is normalized and all of our variables have the same standard deviation. This will ensure that they are weighted equally, and that we can use these weights to create our PCA vectors. We can't have scenarios where a slightly louder word or otherwise "larger" value skews the PCA results.
3. This is based on the fact that our data is in rows, not columns. The PCA algorithm (note 17) dictates that we use the V^T matrix of vectors if our data is in rows. We've stacked the data *vertically*, with each row as a data vector, so V^T will be the value that will store the principal components.
4. SVD/PCA helps compress our data in such a way that we can have a "low rank" approximation of our data that preserves its structure. It must be "low rank" because our launchpad has low processing power and could not store all of the data, thus the SVD/PCA compression allows us to fit our code onto the launchpad without consuming all of the memory. Essentially, SVD/PCA preserves everything that is relevant about the data but takes up a much smaller portion of the system memory, which is perfect for a small computer like the launchpad.
5. To identify what word a recorded data point corresponds to, we must simply check its distance from the centroid of a calculated cluster for that word, after of course applying SVD/PCA to translate our recorded data point to a new basis. If it is within a certain distance to the centroid, it will register as an iteration of that word.
6. For a line, we'd need only one, since this is a one-dimensional shape. For the circle, we'd definitely need two PCA vectors, because there are two important dimensions. With the cylinder, we could probably make do with only two PCA vectors--since the height is small, the only main components are those of the base, which is two dimensional. If using only two PCA vectors doesn't work out, we'll need a third to account for the height component, but it's very likely that using two will produce good results.

Part VI. Advanced Controls

Summary: In this portion, we modified our closed-loop controls to allow for turning while also working on the real-time word classification system that built upon our SVD/PCA work from the last section. We calculated what adjustments we needed to make during our closed loop control to cause a turn of a certain degree value, and then implemented this in our code. Through our testing, we uncovered mechanical errors by testing our car and making adjustments by instructing our car to turn slightly in the one direction when it was really turning the other direction as it "thought" it was going straight due to encoder values. The most annoying part of the lab was implementing classify.ino, since this entailed us changing data points taken from the mic board into the PCA basis and using that to see how close the points were to our centroids corresponding to commands. Getting our words to be recognized properly required a lot of work and going back to part V to reexamine our values.

1. To allow the car to turn, we needed to change exactly what the closed-loop control was adjusting. Like before, we start with the equations:

   u_L = (v_star - f_left * delta + beta_left) / theta_left and

   u_R = (v_star + f_right * delta + beta_right) / theta_right

   But, we need to modify these equations by some value delta_ref, adding a negative value for a right turn and a positive value for a left turn. This is to "trick" the car into thinking it has turned one way and in "adjusting" it is actually turning the opposite way. For going straight, our delta_ref is zero--we've already allowed for straight travel with our closed loop control. We know that delta_ref = ((v_star/5)*n*CAR_WIDTH)/TURN_RADIUS, where n is the sampling interval time, for left turns and negative of that for left turns, so our final equations are simple: the same as above but with delta = delta + delta_ref + STRAIGHT_CORRECTION (see question 2).
2. Delta_ss defines "straight" as when the encoders on each wheel are moving at the same rate, thus synchronizing the motors. It corrects the car so that when it reaches this point of the encoders moving at the same rate, it is travelling in the same direction as it started. This theoretically should be all we need to get the car to go straight, but STRAIGHT_CORRECTION needs to be applied to fix mechanical issues. Even if delta_ss has made the encoder measurements to be the same, axis wobble or mismatched wheel sizes can still cause a tilt, which STRAIGHT_CORRECTION must be employed to correct. In other words, delta_ss matches the encoder measurements, and STRAIGHT_CORRECTION accounts for mechanical errors by causing a "turn" in the opposite direction of how the car turns when delta_ss is applied.
3. The EUCLIDEAN_THRESHOLD represents the maximum distance a data point can be from a centroid to be recognized as the command associated with a centroid, where distances that are too high don't correspond with any command. In the final lab, our Euclidean Threshold is 0.3. Meanwhile, the LOUDNESS_THRESHOLD is essentially the quietest value for which a command is recognized. Below this value, it is assumed that the sound picked up is just background noise. Our loudness threshold in the final lab was 600, which was rather high but worked because of how loud we talk.

Part VII. Integration

Summary: In this section, we combined our functioning car that we perfected in the Closed Loop Control part with the sound recognition software created in the SVD/PCA section. We used our code from prior labs to tie each sound with a different driving command: straight long, left, right, and straight short. The code would essentially listen to input on a loop, doing nothing if it did not detect a sound wave or if it detected a sound that was too soft or above the Euclidean threshold (see part VI, c) and completing the connected drive function if it heard one of the four commands. In theory this would be very simple, but in practice, we ran into a problem when our software had trouble recognizing the correct sound commands. To fix this, we went back and re-recorded some sounds, and also adjusted our thresholds and mic board gain to get the car in working order. By the end, we were able to make our car perform all four driving functions when prompted with the relevant command.

1.
    a. Sam: Throughout the SIXT33N project, we refreshed ourselves on the general tenets of circuit building while adding in elements learned in 16B, from filters to feedback control to PCA. After introductory labs that focused on revisiting EECS 16A content, we began integrating new content with the use of filters, learning how to choose and position resistors and capacitors to create different filters and cutoff frequencies. We also learned how to use system identification, using outputs to calculate general values for an equation. We also learned how to use change of basis to allow our robot to differentiate between sounds, using SVD and PCA to find optimal vectors on which to "plot" out sound files such that they are unique enough for the launchpad to detect a difference. And of course, we learned an awful lot about debugging, and how to use the oscilloscope to determine what problems our circuit faced so that we could apply our knowledge of circuitry to fix them. My favorite part was applying closed-loop control to get the car to drive in a straight line, because it felt like the culmination of many of the previous labs and it was exciting to see our car function as intended, travelling with such precision that it deviated by less than a single floor tile in its journey. My least favorite part was recording (and rerecording) the audio files for the commands, in hopes that our robot could recognize their differences accurately. We tried many workarounds (such as modifying the centroids, changing the loudness and euclidean thresholds and saying the commands differently), but none seemed to work properly.
    b. Yunhao: In the SIXT33N project, I was able to get hands-on experience with what we have learned in 16B. We applied our understanding of basic DC circuits learned in 16A, combined with knowledge about filters, and op-amps to build the operating circuit of the project car. Because of my previous experience in my high school robotics team, I was able to connect the idea of P coefficient in PID control to the eigenvalue of the A matrix. We also learned to use principal components to compress our voice data and used centroids on the project plane and distance to those centroids to classify voice inputs. My favorite part of this project was the time when we used advanced controls to make the car automatically correct for difference in left / right wheel encoder readings. My least favorite part was when we recorded our phrases and used PCA to store them and identify them in the control board. I think it would be better if we use raspberry pi in the next semester so we would have enough space to store the entire voice vector and use cross-correlation (OMP) to determine matching of the phrases. I think it would give practice to the speed-up OMP part and the result would be much easier to find where the problem is.
2. The most difficult bug we encountered, in terms of time, was when we tried to reposition the low pass filter to fit more components on our breadboard. Since the low pass filter required many precisely placed pieces, when we made a single mistake in transferring these pieces to a new location, we disrupted connections and caused the entire circuit to fail. Using the oscilloscope to probe for outputs and find exactly where we erred took the better part of two hours, but we eventually found a place where our actual voltage at a certain node didn't match the intended

voltage, and then retraced the steps of circuit construction to find the way to change our connections and fix it. From this process, we learned the power of the oscilloscope in debugging, and also that a methodical approach of checking components systematically and one-at-a-time for errors is a surefire way to narrow down what is causing the problem and how one can fix it.

3. Lab was one of the most interesting parts of the class, and was a nice medium in which to practically apply the skills we've learned. It was the right balance of difficulty and forgiveness, matching challenging problems with helpful GSIs ready to step in with constructive advice if necessary.

Part VIII. Collaborators and Sources:

Sam Clark did parts V, VI, and VII, while Yunhao Cao did parts II and III. Both partners collaborated on parts I and IV. In terms of sources, only the lab notes, lab files, and class notes were used.