Grayson Gerlich, Hannah Eisenhofer, Nathaniel Moretz

Grading TA: Zaid Bin Tariq

LITEC Lab 5: Accelerometer Report

Section 2 Side B

Professor: Russell Kraft

Introduction

The goal of Lab 5 was to implement I2C devices and control algorithms to on the car. Specifically, the accelerometer was a new I2C device for this lab, and had slightly different functionality to the other devices used in the past. In this lab we also implemented the LCD display and 12 button keypad all operating on the same I2C bus. The RF control unit from Lab 4 returned as well. We developed code to take user input on the keypad, and set thresholds and control gains for the control system. We also developed code to accelerometer outputs to control both car speed and direction, allowing the car to drive downhill and stop upon reaching a flat area.

One of the main focuses of this system was implementing a variety of control variables. We used 3 separate gain values to determine car function: kS for steering, kDX for x axis control, and kDY for y axis control. The first two were set by keypad at the start of operation, while the later could be adjusted at any time using the potentiometer on board. Multiple gains were tested in the development of the system. At high steering gains, the car steered directly down the hill but had a highly jerky response. At high kDX gains, the car accelerated much faster than it turned, leading to the car making sweeping arcs on the ramp instead of linear maneuvers.

A final key component of Lab 5 was calibration and accelerometer data acquisition. The accelerometer we used was very, very prone to noise, and the “0” point would skip about greatly if bumped. Two steps were taken to adjust for this during car functionality. The first was to average 8 accelerometer readings every time accelerometer data was taken. Because a relatively high steering gain was used to get the car to turn 90 degrees on the ramp quickly, small amounts of noise resulted in large jitters in the steering servo, so noise reduction was crucial. The second step was to calibrate an x0 and y0 offset on a level surface before operation. This prevented the accelerometer’s often offset 0 points from affecting the car’s ability to see level ground.

Data and Analysis

Because our group had done the gondola lab before this one, we had an idea of where our gains needed to be. Our final gains for Lab 6 had been a kP of 1 and a kD of 18, so we began in that ballpark for our kS and kDX. Our keypad was designed to allow us to easily select one of 4 different options for both values, as well as a custom value if needed, and we set the keypad to take kS values of 1, 2, 3 or 4. The selections for kDx were originally, 5, 10, 15, and 20, but through testing we narrowed this to 3, 6, 9 and 12.

After extensive trial runs we determined that a kS of 3 and a kDX of 9 was optimal to get the car down the ramp in the straightest line with the smoothest motion (by visual examination). Then, we tailored the kDY with the potentiometer to ensure the car didn’t reverse too fast to sense a horizontal state, and that it would sense level ground at the bottom of the ramp. This was crucial because our manual reverse mode needed to detect a Y value between +- 10 for 400 ms to engage forward drive. With the amount of noise in the sensor, at full kDY drive power, the car could easily drive through this threshold without realizing it.

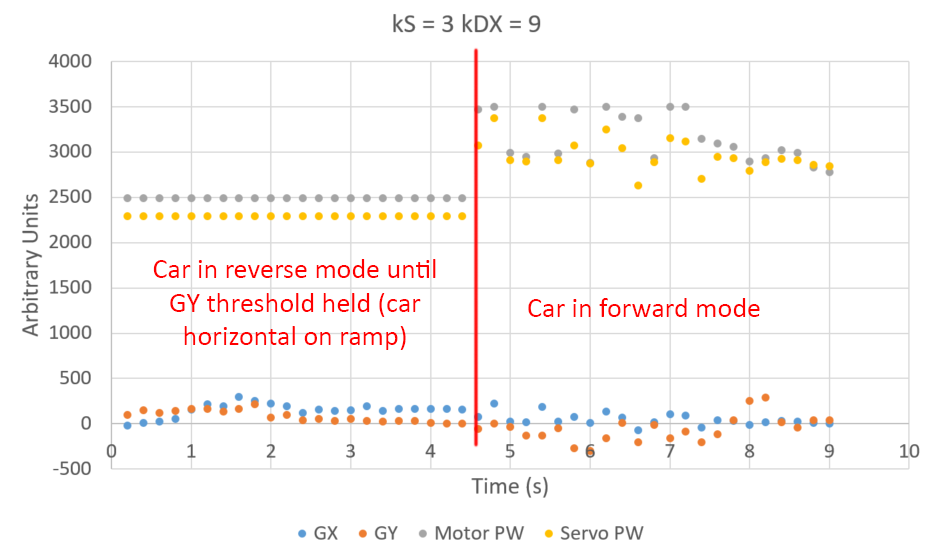


Figure 1: Normal car operation with optimal settings. Car reverses at max motor PW allowed by the potentiometer, and turns fully to the left. Once the Y value (in red) sits close to 0 for enough time, the car engages forward mode and drives until X and Y values approach 0, where it stops when motor PW hits 2760.

Figure 1 demonstrates the optimal run. The car backs up 90 degrees left, turns right 90 degrees and ends directly below where it started. Figure 2 is a clear demonstration of what happens when kDY is set too high.

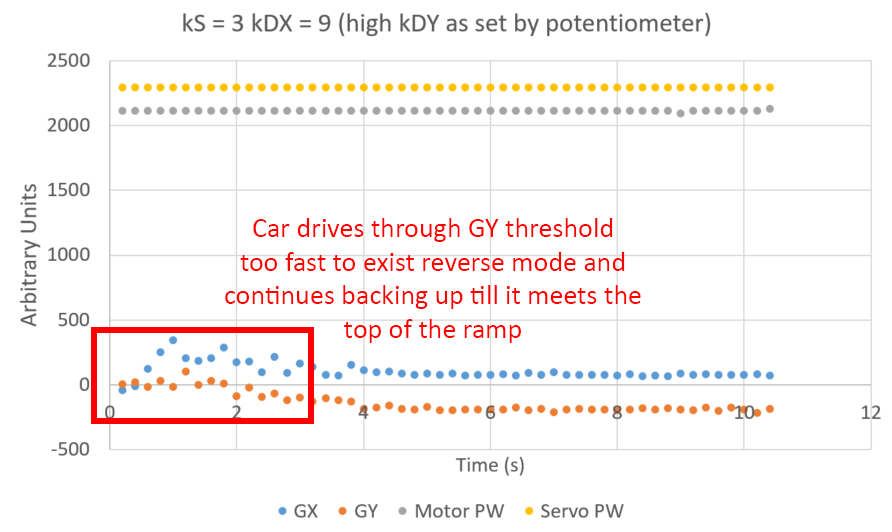


Figure 2: Normal car kS and kDX gains, but with the potentiometer set to about 2/3s maximum kDY. The car accelerates in reverse so quickly it fails to recognize the horizontal position, and the flatline in both Y and X values for the last several seconds is due to the car stalled in reverse as it attempts to crest the ramp.

Figure 2 clearly shows the negative effects of setting a gain incorrectly. The following two figures demonstrate the effects of adjusting the kDX and kS gains beyond their normal settings.

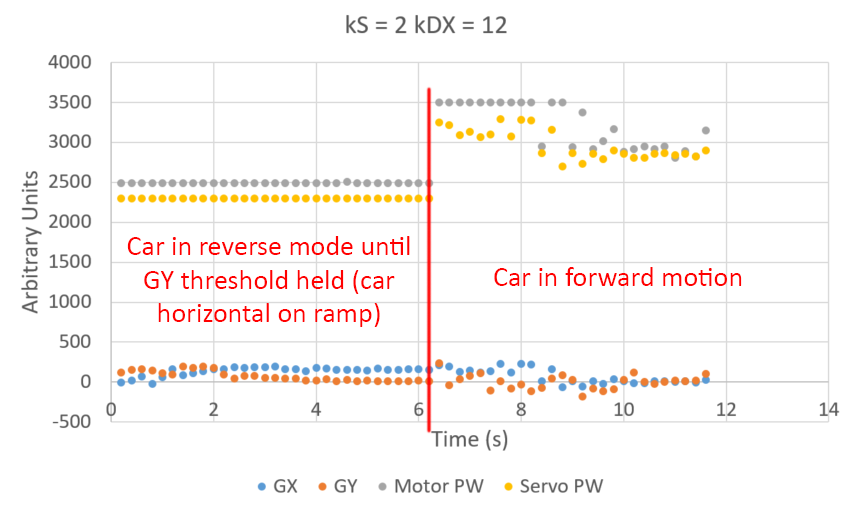


Figure 3: Car operation with a low steering gain and a high kDX. While the car finds horizontal correctly thanks to the kDY setting, the car made a broad, sweeping arc across the ramp. The flat values at about 3500 for the motor PW show the car accelerating at maximum speed, while the steering responds more slowly.

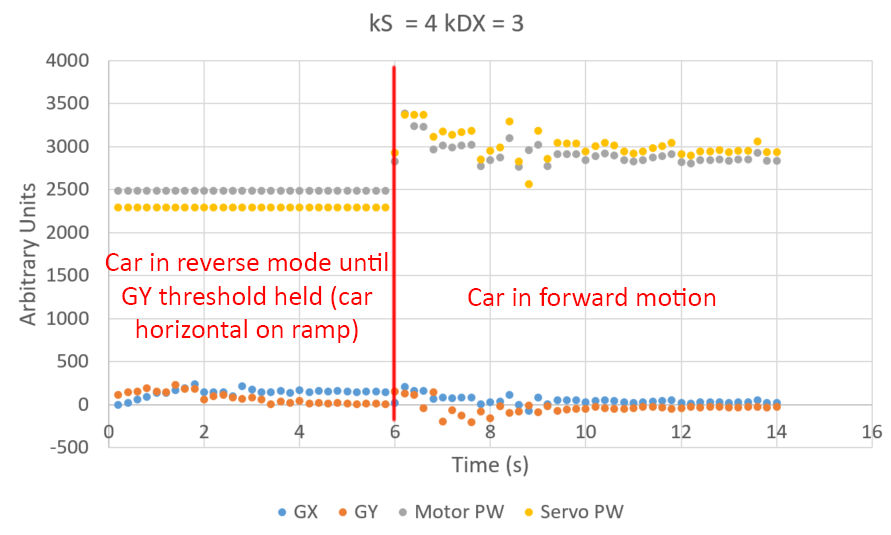


Figure 4: Car operation with a high kS and low kDX. Unlike the previous figure, once the car in this figure enters forward mode, the servo responds more strongly than the drive motor, making the car face down the ramp much earlier while the motor responds slowly. Because kDY is set low, most of the motor’s proportional control comes from kDX, so in this example the car basically coasts to the bottom of the ramp. The 200ms sample frequency dampens the high frequency noise present in this experiment, which was very present visually.

The final figure presented is a direct comparison of the effects of different control values on motor and servo control during the car’s forward motion.

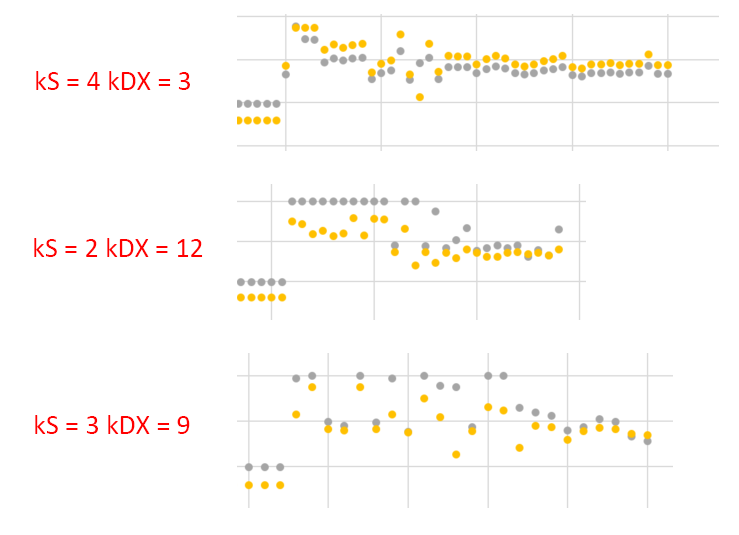


Figure 5: Side by side comparison of different gain settings. The first two are the edge cases, while the final plot shows the optimized case. In the high kS example, the servo (in yellow) clearly responds to the Y axis change, and the motor responds in kind without a strong kDX. In the high kDX example, the motor peaks out due to the tilt of the car, while the servo doesn’t steer hard enough. In the final example, the data points are more spread out because the two gains are working with and against each other to create an average trend which equates to clean motion.

These figures demonstrate the functionality of the car under most working scenarios, as well as the effects of the various gains.

Pseudocode

Variable/function initializations

Main

Inits

While (infinite)

While slide switch == switched

Set motors to neutral

Turn BILED off

If (first time in loop)

Get input from user to confirm calibration

Calibrate

Call set\_gains()

Get direction (right or left) from user

Update first time in loop variable

If motor = forward

Biled green

If motor = reverse

Biled red

If motor = stopped

Biled off

If (accel flag)

If gy within +- 10 for 5 cycles

Levelflag = 0

If levelflag != 0

Set steering in user selected direction

Full reverse

If levelflag = 0

Call set\_servo\_PWM

Call set\_drive\_PWM

Reset flag

Reset count

If (print flag)

Clear LCD

Print to LCD

Print to serial

Reset flag

Read\_AD\_input function

Assign local variable

SFR inits

Wait for conversion to complete

If (cast variable == 0)

Return ADC value

Else

Return ADC variable scaled to some casting gradient

PCA\_ISR function

If (CF)

Increment both counts

If accel count > 80

Update accel flag

accel count = 0

If print count > 200 ms

Update print flag

Print count = 0

PCA0 offset

Pause function

Wait 120 ms

Wait function

Wait 1000 ms

Calibration function

Declare local variables

Reset averages

For 64 counts

I2c read

Wait for confirmation

Read all 4 bytes

Add data

Average data

Read\_accels function

Declare local variables

Reset averages

For 8 counts

I2c read

Wait for confirmation

Read all 4 bytes

Add data

Average data

Subtract calibration data

Set\_servo\_PWM function

Get error value

Adjust PW with ks and gx

Check for PW exceeding PW\_MAX

Set PW

Set\_drive\_PWM function

Read potentiometer value

Scale PW value according to gy

Scale PW value according to gx value (sideways tilt = forward drive)

Check for PW exceeding PW\_MAX

Set PW

Set\_gains function

Declare local variables

Get user input in 5 cases (using standard infinite while and read\_keypad):

1 = ks of 1

2 = ks of 2

3 = ks of 3

4 = ks of 4

5 = custom ks

Get user custom ks

Get user input in 5 cases (using standard infinite while and read\_keypad):

1 = kdx of 3

2 = kdx of 6

3 = kdx of 9

4 = kdx of 12

5 = custom ks

Get user custom ks

Conclusions

Lab 5 was a success in many respects. Not only did the car function essentially exactly as required, our testing process was relatively painless and produced consistent results. This allowed us to debug with relative ease and reduced the amount of time we took figuring out minor errors. We learned the functionality of the accelerometer, and developed a testing regime for gains which worked masterfully. This being said, we did face several large challenges.

First was calibration. Initially we coded the calibration to use the get\_accels function to minimize lines of code we typed. However, when we implemented the x0 and y0 offset in get\_accels, we suddenly couldn’t drive the car. We eventually realized that during calibration, we were immediately subtracting the x0 and y0 we had just calculated from the values we needed every time get\_accels ran. We solved this by simply hardcoding the 64 trial average for calibration.

Then, detecting level ground became an issue. One of the major challenges of both lab 5 and lab 6 was memory allocation. We had room for one more unsigned char, and needed this char to be both a flag for backwards motion and a running counter for detecting flag ground. Because the car started in reverse, we set the value of “levelflag” to 1 initially, and incremented it whenever multiple +- 10 values were detected in the Y axis, resetting the counter to 1 if the values moved out of that range. Once we detected 5 sequential values in the range, we set levelflag to 0 to set the car in forward motion.

The final challenge we faced was with the drive motor. We had forgotten to include code to set the motor in neutral at the start of operation, and so the car would often lag for 5-10 seconds at the start of each run before doing anything. We eventually determined this to be the issue while hardcoding full reverse for the manual reverse mode.

Debugging was often a matter of including print statements or other code flags to indicate where a problem was, and zeroing down on it till it could be solved. In our experience, errors in LITEC are almost always software problems, but we always checked hardware first just to be sure. This was a case in point when the BILED wouldn’t function. We made a few code changes, and before we stressed over other settings, we just replaced the BILED and it functioned fine.

Future improvements to the hardware could include: a way to scale kS down as the car straightens out (perhaps by X value detection) such that the car experiences less jitters as it faces down the ramp. We could also include an integral term to avoid the car stalling at low kDY values. This didn’t become an issue for us, but in early trials, if kDY was set too low, the drive motor simply could not reverse with enough force to get the car to level out.

Hardware implementation:

Nathaniel: 40% Grayson: 30% Hannah: 30%

Software implementation:

Nathaniel: 30% Grayson: 30% Hannah: 40%

Data analysis (if relevant):

Nathaniel: 30% Grayson: 40% Hannah: 30%

Report development & editing:

Nathaniel: 30% Grayson: 40% Hannah: 30%

The following signatures indicate awareness that the above statements are understood and accurate.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_