

ME 461 Laboratory #7 (2 Week Lab)

Balancing and Steering Control of the Segbot

Goals

1. Organize your code from previous labs so that all sensor feedback is sampled in order and the most current signals from the sensors are used in the control law equations.
2. Double check the polarity of feedback signals and motor command output so it matches the model assumptions.
3. Implement Balance Control of the Segbot. Manually tune offsets and balance gains.
4. Implement a driving and steering control for the Segbot.

Exercise 1: Setting up Lab 7

To begin this lab, you will be retrieving your code from Lab 6 (which already integrated code from Lab 5) and code from Lab 4. We will be moving and reorganizing some code. For instance, you **will not be using a CPU Timer Interrupt function for your code**. This is because we want a very specific flow of code from various interrupt service routines. Pay attention to this as you go through this lab.

To start, create a new project using the LAB6starter and rename the project to 'Lab7'. Also, rename the main C file to 'Lab7_main.c'. Open both of your Lab6_main.c and Lab7_main.c files. Copy the entirety of the Lab6_main.c file and paste it over the top of the Lab7_main.c file's contents. Now build your Lab 7 project and it should run identically to where you left off at the end of Lab 6.

I would like you to organize your code by completing the following steps. At any point it might be helpful to debug your code to ensure you have all function and variable definitions set up at the top of your main file.

1. Since Lab 6 merged all your Lab 5 code, your Lab 7 code should be sampling the IMU (MPU-9250) every 1 millisecond. Double check this is happening. In CPU timer 0, you should see the SPI transmission code. The IMU values (linear acceleration and rate gyro values) are received and scaled appropriately in the `SPIB_isr()` function.
2. You will see below that we will move the SPI transmission from the CPU timer 0 interrupt to ADCA's interrupt service routine (don't do this yet). So, we need to set up ADCA! To do so, you need to copy the initializations performed in Lab 4's `main()` function along with the global variables used for ADCA and its filter. Also, make sure the ADCA interrupt function has been enabled and setup in the PIEVector Table. Copy the ADCA interrupt service routine function from

Lab 4 to Lab 7. This interrupt should be reading the joystick results, converting them to a voltage, then filtering them. Ensure these voltages are stored in global float variables so you can print them and for possible use in your final project.

3. Note in your ADCA setup, we are using EPWM5 to trigger ADCA channels 2 and 3 (joystick). So, copy your EPWM5 initializations, which is also found in Lab 4's `main()` function. We wish to sample the channels every **1 millisecond**. Remember that at the end of Lab 4, you sampled the microphone every 0.1 ms, so **make sure to change EPWM5's period back to 1 ms**.

Many of the remaining steps in Exercise 1 have already been done in previous labs or provided for you in the Lab6starter project. We are going over them again just as a refresher and remind you what all is in your code.

4. Ensure your code has EPWM2 set up, along with the `setEPWM2A()` and `setEPWM2B()` functions.
5. Ensure your code calls `setupSPIB()` in the `main()` function to initialize SPIB and its pins. Double check the SPIB interrupt function, `SPIB_isr()`, has been enabled and setup in the PIEVector Table.
6. Ensure your code calls `init_eQEPs()` in the `main()` function to initialize the wheel encoders.

Now that all the initializations are finished, these next steps will walk you through coding the order in which the different feedback signals are sampled. You have already setup ADCA channels 2 and 3 to be triggered by EPWM5 every 1 ms. After the initialization code in `main()`, the code will start the sequence of sampling the needed feedback signals. First in the sequence is sampling ADCA A2 and A3, which are connected to the joystick. Once ADCA A2 and A3 have finished converting, the `ADCA_isr()` function is called. Inside this interrupt function, you will add code to start the second step in the sequence, the SPI transmission to the IMU. Once that is done, the `SPIB_isr()` interrupt function will be called by the hardware. In this function, you will receive the IMU readings and then read the wheel encoders. For just this exercise, you will then drive the motors using your previous `setEPWM2X()` functions. In Exercise 2, you will code the third and final step in the sequence: triggering the software interrupt, or `SWI_isr()`, inside `SPIB_isr()`. It is in the `SWI_isr()` where the control law for the Segbot is calculated and commanded to the motors

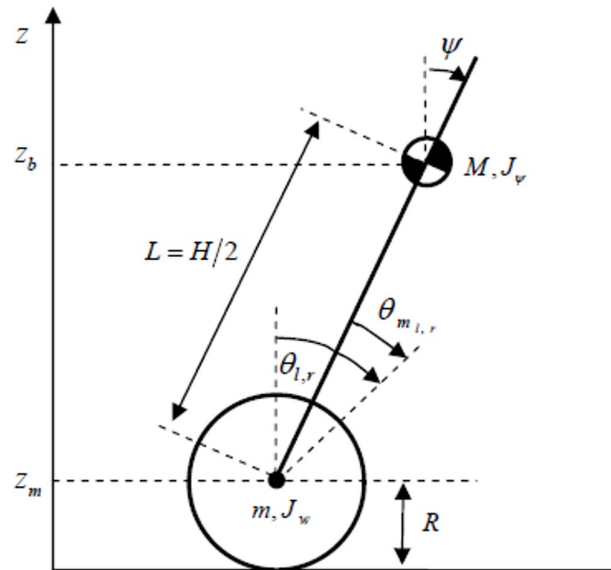
To put it succinctly, our code will flow via the following interrupt functions in order: `ADCA_isr()` → `SPIB_isr()` → `SWI_isr()`. Complete the following steps to begin coding this process:

1. Inside `ADCA_isr()`, after you have read, converted, and filtered the joystick values, start the SPI transmission and reception of the three accelerometer and three gyro readings of the IMU (the MPU-9250). This is identical to the transmission code currently in the CPU timer interrupt 0. Make sure to start the SPIB transmission *only* here in the ADC interrupt function and **not** in the CPU

timer interrupt 0 function (so comment code out where necessary). After the correct number of 16-bit words have been written to SPITXBUF, the `ADCA_isr()` function can acknowledge the interrupt and then exit.

2. When the SPI is finished transmitting (which also means it is done receiving), the `SPIB_isr()` interrupt function will be called by the hardware. Inside the `SPIB_isr()` interrupt function, pull the MPU-9250's Slave Select high and read the 6 IMU values. So that your code works with the next exercise's given code, call your three accelerometer and gyro readings `float accelx`, `float accely`, `float accelz`, `float gyrox`, `float gyroy`, and `float gyroz`. Just like in Lab 5 and Lab 6, convert to units of g for the accelerations and units of degrees/s for the gyro rates.
3. Comment out all the control calculations from Lab 6 for now. It is highly recommended to comment this out so you can copy and paste sections of it later in the lab. Leave the LabVIEW code uncommented. This code should be in whichever CPU timer interrupt function you worked on in Lab 6, probably the timer 1 or timer 2 interrupt.
4. Still inside the SPIB interrupt function, call the `readEncLeft()` and `readEncRight()` functions to sense the left and right motor angles. Store the values in two global float variables.
5. Ensure `UARTPrint` is set equal to 1 every 200 times in the SPIB interrupt so that the `while(1)` loop in `main()` is told to print every 200 ms. You can search (CTRL+F) for `UARTPrint` in your file make certain no other code is setting it equal to 1. Set the `serial_printf()` function in `main()`'s `while(1)` loop to **print the joystick voltages, the accelerometer z value, the gyro x value, and both motor angles.**
6. For this section, simply make sure that the polarities of your motor feedback and controller output are the same as the below figure (which is the same polarities that we set for Lab 6, but let's check to be sure). You will need to add calls to your `setEPWM2A()` and `setEPWM2B()` functions inside the SPIB interrupt function so that you can drive the motors open-loop as you did initially in Lab 6. Create, if not already created, global `float uLeft` and `uRight` variables and simply pass them to the correct `setEPWM2X()` functions. Add negatives where necessary to set the correct direction of the feedback and output commands. You will also check for the correct sign on the angles returned from read encoder functions. Remember, the best place to add a negative for the output is on the "u" value passed to the `setEPWM2X()` functions. Applying a positive control effort should drive the car forward (caster wheel is the back) *and* your angle variables should be increasing.

Show your TA your working code and correct polarities. Also note that x gyroscope value has the correct positive direction as the below figure, but the z acceleration value is in the opposite direction. We will take care of that negative in the Kalman filter code in Exercise 2.



- After you have check your working code with your TA, comment out the `setEPWMX()` function calls you just made. We will call this later and don't want to be calling these functions in two different places. You can also comment out the `readEncLeft()` and `readEncRight()` function calls.

Exercise 2: The Kalman Filter

Now that you have your program sampling all needed feedback signals every 1 ms and the direction of the feedback and motor torque commands are correct, you are ready to implement the balance control for the Segbot. But first, a very important piece of the control algorithm for balancing the Segbot is a Kalman filter.

In order to balance the Segbot, we need some kind of measurement for how tilted the bot currently is. We are receiving two sensor readings from the IMU that will measure this tilt angle. One measurement that does this is the accelerometer's value in the Z direction as it is measuring gravity in units of g. In addition, we could integrate the rate gyro's value (remember, this has units of degrees/second) in the X direction to give us the tilt angle. The accelerometer value is great, but its responsiveness is slow. The rate gyro measurement is very responsive; however, it drifts over time due to integration of the signal noise. A Kalman filter fuses these two sensor readings together to give an accurate "enough" and responsive value measuring the tilt angle of the Segbot.

The Kalman filter will be discussed more in lecture, but since it's an advanced topic outside of the scope for this class, I am giving you the code that implements the Kalman filter equations. Perform the following steps:

1. First, copy all the given global variables towards the top of your main file where the rest of your global variables are defined:

```
// Needed global Variables
float accelx_offset = 0;
float accely_offset = 0;
float accelz_offset = 0;
float gyro_x_offset = 0;
float gyro_y_offset = 0;
float gyro_z_offset = 0;
float accelzBalancePoint = -.76;
int16 IMU_data[9];
uint16_t temp=0;
int16_t doneCal = 0;
float tilt_value = 0;
float tilt_array[4] = {0, 0, 0, 0};
float gyro_value = 0;
float gyro_array[4] = {0, 0, 0, 0};
float LeftWheel = 0;
float RightWheel = 0;
float LeftWheelArray[4] = {0,0,0,0};
float RightWheelArray[4] = {0,0,0,0};
// Kalman Filter vars
float T = 0.001; //sample rate, 1ms
float Q = 0.01; // made global to enable changing in runtime
float R = 25000;//50000;
float kalman_tilt = 0;
float kalman_P = 22.365;
int16_t AverageIndex = -1;
float pred_P = 0;
float kalman_K = 0;
int16_t calibration_state = 0;
int32_t calibration_count = 0;
```

- Copy the below code and paste it in your SPIB interrupt function right after your code has read the IMU values and scaled them to their correct units (and BEFORE the `if ((SpiNumCalls % 200)...` line):

```
//Code to be copied into SPIB_isr interrupt function after the IMU measurements have been collected.
if(calibration_state == 0){
    calibration_count++;
    if (calibration_count == 2000) {
        calibration_state = 1;
        calibration_count = 0;
    }
} else if(calibration_state == 1) {
    accelx_offset+=accelx;
    accely_offset+=accely;
    accelz_offset+=accelz;
    gyro_x_offset+=gyrox;
    gyro_y_offset+=gyroy;
    gyro_z_offset+=gyroz;
    calibration_count++;
    if (calibration_count == 2000) {
        calibration_state = 2;
        accelx_offset/=2000.0;
        accely_offset/=2000.0;
        accelz_offset/=2000.0;
        gyro_x_offset/=2000.0;
        gyro_y_offset/=2000.0;
        gyro_z_offset/=2000.0;
        calibration_count = 0;
        doneCal = 1;
    }
} else if(calibration_state == 2) {
    accelx -=(accelx_offset);
    accely -=(accely_offset);
    accelz -=(accelz_offset-accelzBalancePoint);
    gyro_x -= gyro_x_offset;
    gyro_y -= gyro_y_offset;
    gyro_z -= gyro_z_offset;

    /*-----Kalman Filtering code start-----*/
    float tiltrate = (gyrox*PI)/180.0; // rad/s
    float pred_tilt, z, y, S;

    // Prediction Step
    pred_tilt = kalman_tilt + T*tiltrate;
    pred_P = kalman_P + Q;

    // Update Step
    z = -accelz; // Note the negative here due to the polarity of AccelZ
    y = z - pred_tilt;
    S = pred_P + R;
    kalman_K = pred_P/S;
    kalman_tilt = pred_tilt + kalman_K*y;
    kalman_P = (1 - kalman_K)*pred_P;

    AverageIndex++;
    // Kalman Filter used
    tilt_array[AverageIndex] = kalman_tilt;
    gyro_array[AverageIndex] = tiltrate;
    LeftWheelArray[AverageIndex] = readEncLeft();
    RightWheelArray[AverageIndex] = readEncRight();

    if (AverageIndex >= 3) { // should never be greater than 3
        tilt_value = (tilt_array[0] + tilt_array[1] + tilt_array[2] + tilt_array[3])/4.0;
        gyro_value = (gyro_array[0] + gyro_array[1] + gyro_array[2] + gyro_array[3])/4.0;
        LeftWheel=(LeftWheelArray[0]+LeftWheelArray[1]+LeftWheelArray[2]+LeftWheelArray[3])/4.0;
        RightWheel=(RightWheelArray[0]+RightWheelArray[1]+RightWheelArray[2]+RightWheelArray[3])/4.0;
        AverageIndex = -1;
    }
}
```

```

        PieCtrlRegs.PIEIFR12.bit.INTx9 = 1; // Manually cause the interrupt for the SWI
    }
}
if((SpibNumCalls%200) == 0) {
    if(doneCal == 0) {
        GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1; // Blink Blue LED while calibrating
    }
    GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1; // Always Block Red LED
}

```

Before you go onto writing the controller code to balance the Segbot, let's go through the given code:

- For the first four seconds, the code is calculating the constant offsets of the three accelerometers and the three rate gyros. Actually, the first 2 seconds the code is doing nothing in order to let the IMU settle down after powering on. Then for the next 2 seconds, the code is summing up each reading received. At the end of those 2 seconds the summed variable is divided by 2000 to find the average offset over a 2 second span.
- After 4 seconds have elapsed, the offsets have been found, and `calibration_state` is set to 2. This means the offset calibration is complete and the balance algorithm can run.
- Now every 1 ms the Kalman filter equations are run to calculate the `kalman_tilt` value.
- The balancing control law works well at 4 ms. However, the Kalman equations are run every 1 ms. A 4-point average is performed to filter the feedback signals a bit more. A 4-point average is also performed on the left and right encoder position values.
- Finally, the Software Interrupt function, `SWI_isr()`, is triggered to run. You will write all the remaining Segbot balancing C code in this function. Change your `serial_printf()` to print the four 4-point averaged feedback signals: `tilt_value`, `gyro_value`, `LeftWheel`, `RightWheel`.

Exercise 3: Balancing the Segbot

At this point in the flow of the code, you enter the software interrupt function `SWI_isr()`. Inside the SWI you have all the feedback data collected. Complete the following steps to implement and run the balancing control law for the Segbot:

1. First, calculate left and right wheel speed in radians/second. As discussed in lecture use the continuous transfer function $125s/(s + 125)$ to estimate these velocities. Remember that this continuous transfer function becomes the discrete transfer function $(100z - 100)/(z - 0.6)$ at 0.004 seconds. Find the difference equation for this transfer function. With this difference equation, solve for a filtered version of `float velRight` and `float velLeft` (note: if you do not like the variable names I suggest, you can use your own naming convention, but leave the variables in the Kalman filter code alone).
2. Also with the same exact derivative approximation, $(100z - 100)/(z - 0.6)$ at 0.004, find the derivative of the `gyro_value` value and call it `gyrorate_dot`.
3. Now you have all the states needed to calculate your balancing control. The full state feedback controller equation is: $u = -kx$. In this case the states are $x_1 = \text{tilt}$, $x_2 = \text{gyro_value}$, x_3 is the average of the left and right motor velocities in radians/second, and $x_4 = \text{gyrorate_dot}$. Calculate this control law and assign it to the variable `ubal`. You should end up with the following equation:

$$\text{ubal} = -K_1 * \text{tilt} - K_2 * \text{gyro_value} - K_3(\text{velLeft} + \text{velRight})/2.0 - K_4 * \text{gyrorate_dot}$$

Use values $K_1 = -60$, $K_2 = -4.5$, $K_3 = -1.1$, and $K_4 = -0.1$.

4. `ubal` is the control effort for both motors, so set `uLeft = ubal/2` and `uRight = ubal/2`.
5. Write these values to the `setEPWM2X()` functions to drive the motors.
6. **Have your TA explain** the initialization procedure for the Segbot. You will need to start the Segbot on its kickstands and then switch on the amp switch. Keep the Segbot at rest for the first four seconds so it can calculate the IMU offsets.
7. **Demo your Segbot balancing. Once you have balancing working power your Segbot with a battery. You will probably notice the Segbot either drifting forward or backward. Adjust the variable `accelzBalancePoint` in a CCS Watch Expression until you improve this drift. It will not be perfect. We will fix this drift even better in Exercise 5. Change `accelzBalancePoint` in the declaration to the value you found.**

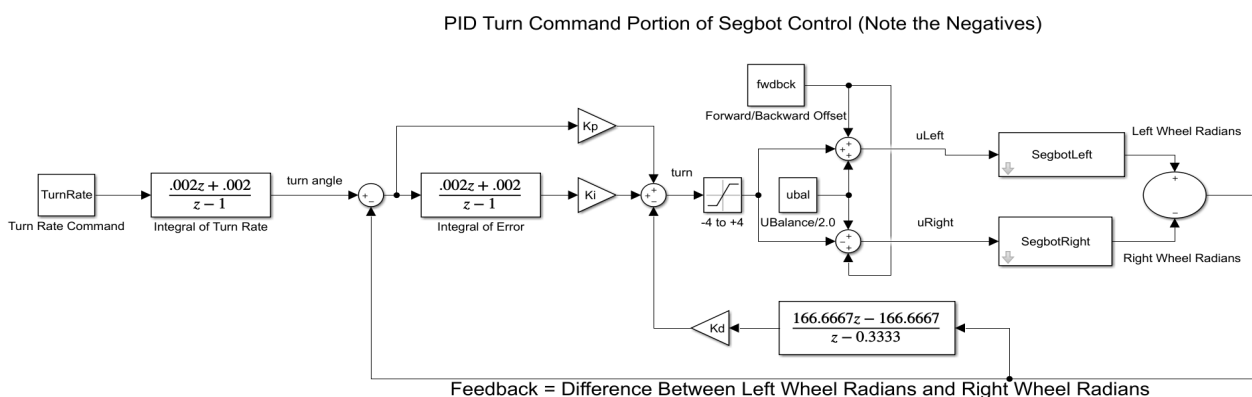
Exercise 4: Steering the Segbot

When playing with your Segbot in exercise 3 after getting it to balance, you may have noticed the Segbot turning and drifting to the left or to the right. If it was not drifting, you probably noticed that it was pretty easy to turn the Segbot left or right. The reason for this is that in exercise 3 only the balance control was implemented. The balance control is not watching the bearing angle of the Segbot, only the velocity of the motors. In this exercise you will implement a PID control algorithm to command the Segbot to an angle and to a given turn rate. This will allow you to steer the Segbot just as you did the robot car in Lab 6.

When designing a control law to steer the Segbot, we have to keep in mind that the most important part of the controller is the balancing controller. If the turn command to the Segbot overpowers the balance control, the Segbot will fall over. Remember that the limit of the control effort, u_{Left} and u_{Right} , is -10 to $+10$. To keep the turn control command from overpowering the balance control, the turn command will be limited between -4 to $+4$. It is possible to change this if you need faster turns but some limit is necessary otherwise the balance control will not be able to do its job.

The control law that is explained below attempts to control the difference between the Segbot's left and right wheel angles. If the difference between the wheel angles is commanded to stay at zero, then the Segbot will not turn. When the Segbot is turning one wheel is turning more than the other, or if the Segbot is turning in place one wheel is turning positive and the other wheel is turning negative. This control law is assuming minimal wheel slippage, which is usually a good assumption.

Use the below block diagram and numbered tasks to implement the steering control. Make sure to note the positive and negatives in the summer blocks.



Think of this control initially as separate from the balancing control. The last step will be to saturate the control and add it to the left wheel's balance control and subtract it from the right wheel's balance control.

1. Create a global float variable `WhlDiff` and set it equal to the difference between the left wheel's angle in radians and the right wheel's angle in radians (note: if you do not like the variable names I suggest below, you can use your own naming convention).
2. Find the rate of change (velocity) of this `WhlDiff` feedback by performing similar filter equations as done in exercise 3 but here using the filter $250s/(s + 250)$ which in discrete time is $(166.667z - 166.667)/(z - 0.333)$ at 0.004 second sample period. You will need to form the difference equations of this discrete filter. Remember to create variables `WhlDiff_1` (old value), `vel_WhlDiff`, and `vel_WhlDiff_1` (old value).
3. Create a float variable `turnref`. In the block diagram above it is labeled "turn angle." By changing this `turnref` variable, you will be able to command the Segbot to turn to an angle and stop. Note that `turnref` is not directly the angle the robot will turn to. You would have to take into account the wheel's diameter and the distance between the wheels for that to be accomplished. You will see in later steps that for steering around the Segbot it is better to command a turn rate, so we will not take time to calibrate the `turnref` to an actual angle of the Segbot.
4. Create an error variable and find the error between `turnref` and your feedback signal `WhlDiff`. So, `errorDiff = turnref - WhlDiff`.
5. Similar to what you did in Lab 6, integrate `errorDiff` using the trapezoidal rule. You will have to create an old error variable and an old integral variable.
6. Calculate your PID control to give you the turn command:


```
turn = Kp*errorDiff + Ki*intDiff - Kd*vel_WhlDiff;
```
7. Guard against integral windup by checking if the absolute value, `fabs()`, of `turn` is greater than 3. If it is then set `intDiff` to the old `intDiff` variable.
8. Saturate turn between -4 and 4.
9. Save the current values to the "old" variables, `errorDiff_1`, `vel_WhlDiff_1`, `intDiff_1`.
10. Now you are ready in calculate the control effort for the left and right wheel. Calculate your `uLeft` and `uRight`. Notice that `turn` is subtracted from `uRight` and added to `uLeft`.


```
uRight = ubal/2.0 - turn;
uLeft = ubal/2.0 + turn;
```
11. Start out with `Kp = 3.0`, `Ki = 20.0`, `Kd = 0.08`.
12. Build and run your code. When running and balancing, in a Code Composer watch expression, change the value of `turnref` to 1 or even a larger number to see your Segbot turn. Then set

`turnref` to a negative number and your Segbot should turn the other way. To see the effects of poorly tuned PID gains, change to $K_p = 1.0$, $K_i = 20.0$, $K_d = 0.01$. Again, change `turnref` positive and then negative to apply step changes in the Segbot's turn. How does the Segbot react with these poorly tuned gains? **Show your TA, then put the PID gains back to the first set. For your final project, you may want to tune these PID gains to get the Segbot to respond to turn commands with a different response, faster or slower depending on your application.**

13. To steer the Segbot around the room, it is easier to command a turn rate instead of a turn angle. There may be some situations where commanding an angle is better. For example, to have the Segbot dance, it may be nice to command an angle. The reference to this PID control is the `turnref` or an angle. To change to commanding a turn rate, you will add another integral equation to integrate the turn rate command to solve for `turnref` (see block diagram above).
14. Create a float variable `turnrate` along with an old `turnrate_1`, and an old `turnref_1`. Use again the trapezoidal rule to integrate `turnrate` to give you the `turnref` needed each 4 ms in your PID control equations.
15. Build, run, and test your code. In a CCS Watch Expression, set `turnrate` to a small value and see that your Segbot turns at a certain rate.

Exercise 5: Driving the Segbot

The Segbot can now balance and turn. We will now implement a way to drive the car forward and backward. For this Segbot speed controller, I am giving you a bit less detail on how to implement this final part of the controller in order to have you think through the PI controller implementation yourself. This speed control will add one additional control effort to our `uRight` and `uLeft` equation. The final equation for `uRight` and `uLeft` will be as follows:

```
uRight = ubal/2.0 - turn - ForwardBackwardCommand;
```

```
uLeft = ubal/2.0 + turn - ForwardBackwardCommand;
```

Note: the negative sign on `ForwardBackwardCommand` is due to how the Segbot directions have been defined.

Perform the following numbered tasks to guide you in implementing this speed control of the Segbot.

1. The feedback for this speed control will be the average wheel velocity $(velLeft + velRight)/2.0$. Error will be $eSpeed = (Segbot_refSpeed - (velLeft + velRight)/2.0)$.
2. Implement a PI control using this error and starting with `KpSpeed = 0.35` and `KiSpeed = 1.5`:

```
ForwardBackwardCommand = KpSpeed*eSpeed + KiSpeed*IK_eSpeed;
```
3. To not allow integral windup, set `IK_eSpeed` to its old value from 4ms ago if

```
fabs(ForwardBackwardCommand) > 3.
```
4. Saturate `ForwardBackwardCommand` to not be less than -4 and not greater than 4.

Once you have implemented this last feedback loop for the Segbot controller, download and run your code. With `Segbot_refSpeed = 0`, you should find that the Segbot does a better job of holding itself at zero speed. It still moves back and forth a bit to stay balanced, but it should stay in place.

With the controller fully working, setup your code to communicate with LabVIEW. Your timer interrupt function from Lab 6 should have the code for communicating with LabVIEW. Move that code into the software interrupt `SWI_isr()` function. Change the `numTimer?Calls` count variable to the `numSWIcalls` count variable used in the SWI. Copy over your kinematic equations that calculate the x , y , and bearing (θ) pose of the Segbot (these are the same equations for both the Lab 6 Robot configuration and the Segbot configuration). Also, change the two lines of code that when `NewLVData` is equal to one. Assign `Segbot_refSpeed` to `fromLVvalues[0]` and `turnrate` to `fromLVvalues[1]`.

Remind yourself of the steps given in LabVIEW assignment #3 and Lab #6 to connect LabVIEW with your robot. Then, put your Segbot on the floor and first get the Segbot balancing. Then, telnet into your robot's ESP32, run `tcpLVCOM`, run your LabVIEW program, and start steering the Segbot around the floor. To command the Segbot to go backwards you will need to add a case in your LabVIEW program for when the 's' key is pressed. If the 's' key is pressed subtract 0.1 from `Vref`.

Lab Checkoff

1. Demonstrate your "Merged" code working. Sampling the joystick, IMU, optical encoders and commanding PWM.
2. Demonstrate the Kalman filter code working.
3. Demonstrate your Segbot Balancing.
4. Demonstrate the turn PID code working with good PID gains and poor PID gains.
5. Demonstrate the Segbot running on the floor commanding it forward and backwards and turning left and right.
6. Submit all your written code after adding comments explaining what you learned. Also be clear what code is for what exercise.