ME 461 Laboratory #6
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.
5. Experiment with a simulation of the Segbot. Gain intuition from the simulation to help you tune the actual Segbot gains.

Exercise 1 (Mostly the same as exercise 5 of Lab 5 with more detail):

It will be helpful to start Lab 6 code with your Lab 5 code. *But note, you will not be using a CPU Timer Interrupt function for this code. You will need to move some of the code you wrote in the CPU Timer Interrupt function into other interrupt functions specified below.*

I would like you to organize your code in the following structure by completing these steps. If I forgot a step/code you think is necessary of course go ahead and include it.

- You will need to copy many of the initializations performed in Lab 3, Lab 4 and Lab 5 into your main() function. Copy all of the initialization code for ADCA. You will be using EPWM5 to trigger ADCA channels 2 and 3 every 1 millisecond. Also make sure the ADCA interrupt function has been enabled and setup in the PIEVector Table.

- Make sure to copy your setEPWM6A and setEPWM6B functions. Also make sure to call the correct code in main() to initialize EPWM6A/B and set them with a 20kHz carrier frequency as you did in Lab 5.

- Still in main() make sure to call your setupSPIB() function to initialize SPIB as used in Lab 4. Make sure the SPIB interrupt function has been enabled and setup in the PIEVector Table

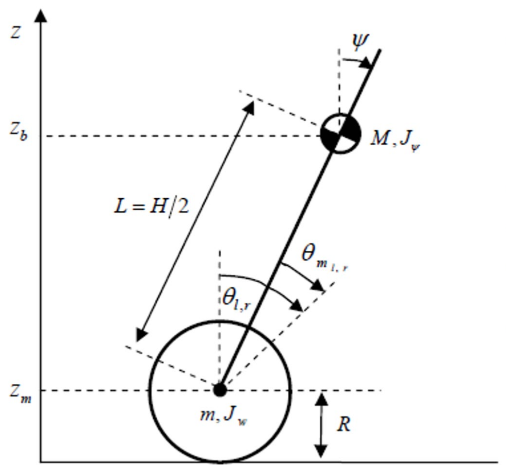
- Once all the initialization are finished, you will start coding the order the different feedback signal are sampled. You have already setup ADCA channels 2 and 3 to be triggered by EPWM5 every 1ms. So once you have enable interrupts in main() the code here and below will start the sequence of sampling 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, ADCA_ISR() function is called. Inside the ADCA1 interrupt, make sure to read the Joystick results and convert the 12 bit ADC values to voltages. Store the two voltages in global float variables for possible use in the control algorithms implemented in later exercises.

- Still inside the ADCA_ISR() function, after you have read the conversion results of each Joystick axes, start the SPI transmission and reception of the three accelerometer readings and the 3 gyro readings of the MPU-9250. This is identical code that you performed in a CPU Timer Interrupt function in Lab 4. Make sure to only start the SPIB transmission here in the ADC interrupt function and not also in the CPU Timer Interrupt function where your lab 4 code started the SPIB transmission. After the correct 16 bit words have been written to SPITXBUF the ADC_ISR() function can exit.

- 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 exercises given code, call your three accelerometer and gyro readings float accelx, float accely, float accelz, float gyrox, float gyroy and float gyroz. Convert to units of g for the accelerations and units of rad/s for the gyro rates.

- Still inside the SPIB interrupt function, call the readEncLeft() and readEncRight() functions to sense the left and right motor's angle. *Note that you will need to call the eQEP initialization function in main() to setup the eQep.* Store motor angles in two global variables. Finally set UARTPrint equal to 1 every 200 times into the SPIB interrupt so that the while(1) loop in main() is told to print every 200 ms. 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.

- The Segbot balance controller will run every 4 ms. but we will take care of that in the next section. For this section simply make sure that the polarities of your feedback and controller output are the same as the below figure. You will need to add calls to your setEPWM6A and setEPWM6B functions inside the SPIB interrupt function so that you can drive the motors open-loop as you did initially in Lab 5. Create global float uLeft and uRight variables and simply pass them to the correct setEPWM6 function. Add negatives where necessary to set the correct direction of the feedback and output commands. Remember the best place to add a negative is on the u value passed to the setEPWM6 functions and when you read the sensor data from the peripheral (ADC, SPI, eQEP). **Show Your TA your working coding and correct polarities.**



Exercise 2 Kalman Filter

Now that you have your program sampling all needed feedback signals every 1ms 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 using the Kalman filter equations to perform sensor fusion of the integral of the rate gyro and the accelerometer tilt value to produce a responsive tilt measurement. The accelerometer gives us a tilt measurement but the responsiveness of this tilt measurement is slow. The integral of the rate gyro, which also gives tilt angle, is a very responsive feedback but it has the problem that it drifts over time due to signal noise. Using the Kalman filter, the accuracy of the accelerometer and the responsiveness of the integral are fused together to produce a tilt measurement that is responsive and does not drift. I will be covering the Kalman filter in future lectures so for now I am giving you the code that performs the Kalman Filter equations. See the code below. This code should be copied into your SPIB_ISR() function.

- First copy all the given global variables towards the top of your C-file where the rest of your global variables are defined.
- Copy this 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. Look through this given code to see if you are already doing some these step and comment out your code if anything is redundant. I am mainly thinking here about the last few lines of this given code that blinks LEDs, triggers an UART print and acknowledges the SPIB interrupt.

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, for the first 2 seconds the code is doing nothing in order to let the IMU settle down after power 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 the 4 seconds have elapsed the offsets have been found and calibration_state is set to 2 which 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.
- Since the balancing control law works well at 4 ms. but 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 print to print the four 4 point averaged feedback signals to Tera Term every 200 ms, tilt_value, gyro_value, LeftWheel, RightWheel.

```
// Needed global Variables
float accelx_offset = 0;
float accely_offset = 0;
float accelz_offset = 0;
float gyrox_offset  = 0;
float gyroy_offset  = 0;
float gyroz_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 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 SpibNumCalls = -1;
float pred_P = 0;
float kalman_K = 0;
int32_t timecount = 0;
int16_t calibration_state = 0;
int32_t calibration_count = 0;
```

```
//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;
        gyrox_offset+=gyrox;
        gyroy_offset+=gyroy;
        gyroz_offset+=gyroz;
        calibration_count++;
        if (calibration_count == 2000) {
            calibration_state = 2;
            accelx_offset/=2000.0;
            accely_offset/=2000.0;
            accelz_offset/=2000.0;
            gyrox_offset/=2000.0;
            gyroy_offset/=2000.0;
            gyroz_offset/=2000.0;
            calibration_count = 0;
            doneCal = 1;
        }

    } else if(calibration_state == 2){

        accelx -=(accelx_offset);
        accely -=(accely_offset);
        accelz -=(accelz_offset-accelzBalancePoint);
        gyrox -= gyrox_offset;
        gyroy -= gyroy_offset;
        gyroz -= gyroz_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;
        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;

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

        if (SpibNumCalls >= 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;
            SpibNumCalls = -1;

            PieCtrlRegs.PIEIFR12.bit.INTx9 = 1;  // Manually cause the interrupt for the SWI
        }
    }

    timecount++;

    if((timecount%200) == 0)
    {
        if(doneCal == 0) {
            GpioDataRegs.GPATOGGLE.bit.GPIO31 = 1;  // Blink Blue LED while calibrating
        }
        GpioDataRegs.GPBTOGGLE.bit.GPIO34 = 1;  // Always Block Red LED

        UARTPrint = 1;  // Tell While loop to print
    }

    SpibRegs.SPIFFRX.bit.RXFFOVFCLR=1;  // Clear Overflow flag
    SpibRegs.SPIFFRX.bit.RXFFINTCLR=1;  // Clear Interrupt flag
    PieCtrlRegs.PIEACK.all = PIEACK_GROUP6;
```

Exercise 3 Balance 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 that you need to implement and run the balancing control law for the Segbot.

- First, calculate left and right wheel speed in radains/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. Find the difference equation for this transfer function to find the equation that given. With these difference equation solve for vel_Right and float vel_Left.

- Now you have all the states needed to calculate your balancing control. The full state feedback controller equation is: u = -k*x. I this case the states are x1 = tilt, x2 = gyrorate and x3 is the average of the current left and right velocity in radians/second. Calculate this control law and assign it to the variable “ubal”. Ubal = -K1*tilt – K2*gyrorate – K3*(velLeft + velRight)/2. Use K1 = -30, K2 = -2.8, K3 = -1.0

- “ubal” is the control effort for both motors, so set uLeft = ubal/2 and uRight = ubal/2;

- Write these values to the setEPWM6A and 6B functions to drive the motors.

- **Have your TA explain** the initialization procedure for the Segbot. You will need to start the Segbot at a certain angle then switch on the amp switch. Keep the Segbot at rest for the first four seconds so it can calculate the IMU offsets.

- **Demo your Segbot balancing**

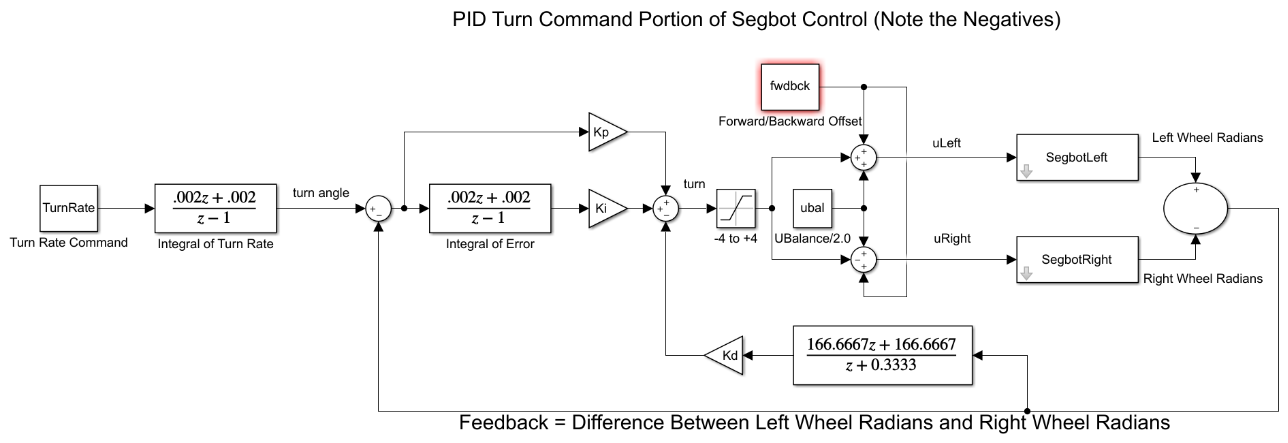
Exercise 4 Steering the Segbot

When playing with your Segbot in exercise 3 after getting it to balance, you may have noticed the Segbot drifting to the left or to the right. If was not drifting you probably noticed that it was pretty easy to turn the Sebot 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 angle of the motor, only the velocity of the motor. In this exercise you will implement a PID control algorithm to command the Segbot to an angle and also to a given turn rate. This will allow you to steer the Segbot just as you did the robot car in Lab 5.

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 over powers the balance control, the Segbot will fall over. Remember that the limit of the control effort, “uLeft” and “uRight”, is -10 to +10. To keep the turn control command from over powering 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 bullet items 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.

- 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.)

- 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 a WhlDiff_1 (old value) variable, a vel_WhlDiff variable and a vel_WhlDiff_1 (old value) variable.

- 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.

- Create an error variable and find the error between turnref and your feedback signal WhlDiff. errorDiff = turnef-WhlDiff;

- Similar to what you did in Lab 5, integrate errorDiff using the Trapezoidal rule. You will have to create an error old variable and an integral old variable.

- Calculate your PID control to give you the turn command.

  turn = Kp*errorDiff + Ki*intDiff - Kd*vel_WhlDiff

- 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.

- Saturate turn between -4 and 4.

- Save the current values to the "old" variables, errorDiff_1, vel_WhlDiff_1, intDiff_1.

- Now you are ready in calculate the control effort for the left and right wheel. Here create another global variable "FwdBackOffset". For now keep it at zero. This variable will be used to drive the Segbot forward and backward. Calculate your uLeft and uRight. Notice that turn is subtracted from right and added to left.

  uRight = ubal/2.0 – turn + FwdBackOffset;

  uLeft = ubal/2.0 + turn + FwdBackOffset;

- Start out with Kp = 3.0, Ki = 20.0, Kd = 0.08.

- Build and run your code. When running and balancing, in a Code Composer watch expression change the value of turnref to 1 and then other numbers to see your Segbot turn. Also at this time, adjust your Kp, Ki and Kd gains to see how they affect the turn step response. **Show your TA.**

- To steer the Segbot around the room it is an easier task when commanding a turn rate instead of a turn angle. There may be some situations where commanding an angle is better. For example having 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.

- 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.
- Build, run and test your code. Set turnrate to a small value and see that your Segbot turns at a certain rate.
- Add the following code to your SerialRXA function. Using Tera Term and your 6ft USB cable, steer your Segbot around the lab by pressing the ‘q’ and ‘r’ keys for left and right, and ‘3’ and ‘s’ to go forward and backward. Any other key stops the Segbot. **Show your TA.**

```
// This function is called each time a char is recieved over UARTA.
void serialRXA(serial_t *s, char data) {
   numRXA ++;
   if (data == 'q') {
      turnrate = turnrate - 0.2;
   } else if (data == 'r') {
      turnrate = turnrate + 0.2;
   } else if (data == '3') {
      FwdBckOffset = FwdBckOffset - 0.2;
   } else if (data == 's') {
      FwdBckOffset = FwdBckOffset + 0.2;
   } else {
      turnrate = 0;
      FwdBckOffset = 0;
   }
}
```

Takehome Exercise Simulating the Segbot

For this take-home exercise, I would like to thank Yorihisa Yamamoto who is (was?) an employee at Mathworks and put together this documentation and Simulink files for controlling a self-balancing robot made out of Legos. I will be referencing his paper that describes the equations of motion of a Segbot. All his work can be found at the link https://www.mathworks.com/matlabcentral/fileexchange/19147-nxtway-gs-self-balancing-two-wheeled-robot-controller-design. If you download his files and find the docs folder you will find his paper NXTway-GS Model-Based Design.pdf.

The goal with this problem is to design a balancing control law to stabilize the Lab Segbot we built and have already controlled in this lab. The parameters of the Lab Segbot are given to you for this problem. If you study the equations.m file you will find some of the parameters are educational guesses. *If you find this problem very interesting you could extend this work into your final project. For example it would be very interesting to perform a comparison of what controller gains work on the real Segbot compared to what works in simulation. This is not what you need to do for this take-home assignment.* With these equations, you will be able to check the balancing controller given in Exercise 3 and design your own balancing controller gains.

The controller I would like you to design and simulate for this exercise is just the balancing control law. For the balancing control, we only need to think of the Segbot looking at its side so motor θ_right equals motor θ_left and the voltage applied to both motors will be the same value v. Looking at the below figures we are only interested in the left figure that is looking at the side view of the Segbot which means φ is always going to be zero. So taking equations 3.13, 3.14 from the NXTway paper (Equation 3.15 is equal to zero because φ is zero) and setting φ to zero the equations of motion for the Segbot become: (See the NXTway paper for more details)

$$\psi = body\ rotation\ angle, \theta = motor\ rotation\ angle, v = voltage\ applied\ to\ one\ motor$$

$$\alpha = \frac{nK_t}{R_m}, \qquad \beta = \frac{nK_tK_b}{R_m} + f_m$$

$$\begin{bmatrix} ML^2 + J_\psi + 2n^2J_m & MRLcos(\psi) - 2n^2J_m \\ MRLcos(\psi) - 2n^2J_m & (2m+M)R^2 + 2J_w + 2n^2J_m \end{bmatrix} \begin{bmatrix} \ddot{\psi} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} MgLsin(\psi) + 2\beta\dot{\theta} - 2\beta\dot{\psi} - 2\alpha v \\ MLR\dot{\psi}^2 \sin(\psi) - 2\beta\dot{\theta} + 2\beta\dot{\psi} + 2\alpha v \end{bmatrix}$$

$$linearize\ with\ 2nd\ order\ Taylor\ series, \cos = 1, \sin = angle\ , and\ squared\ states = 0$$

$$\begin{bmatrix} ML^2 + J_\psi + 2n^2J_m & MRL - 2n^2J_m \\ MRL - 2n^2J_m & (2m+M)R^2 + 2J_w + 2n^2J_m \end{bmatrix} \begin{bmatrix} \ddot{\psi} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} MgL\psi + 2\beta\dot{\theta} - 2\beta\dot{\psi} - 2\alpha v \\ -2\beta\dot{\theta} + 2\beta\dot{\psi} + 2\alpha v \end{bmatrix}$$

$$assign\ D = \begin{bmatrix} ML^2 + J_\psi + 2n^2J_m & MRL - 2n^2J_m \\ MRL - 2n^2J_m & (2m+M)R^2 + 2J_w + 2n^2J_m \end{bmatrix}, which\ is\ a\ constant\ and\ invertable$$

$$then\ this\ equation\ can\ be\ rewritten$$

$$\begin{bmatrix} \ddot{\psi} \\ \ddot{\theta} \end{bmatrix} = D^{-1} \begin{bmatrix} MgL & -2\beta & 0 & 2\beta \\ 0 & 2\beta & 0 & -2\beta \end{bmatrix} \begin{bmatrix} \psi \\ \dot{\psi} \\ \theta \\ \dot{\theta} \end{bmatrix} + D^{-1} \begin{bmatrix} -2\alpha \\ 2\alpha \end{bmatrix} v$$

$$since\ D\ is\ a\ constant\ symmetric\ matrix, D^{-1}\ is\ also\ a\ constant\ symmetric\ matrix$$

$$define\ D^{-1} = \begin{bmatrix} DI_{11} & DI_{12} \\ DI_{12} & DI_{22} \end{bmatrix}$$

$$set\ the\ states\ of\ our\ linearized\ system\ to\ \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} \psi \\ \dot{\psi} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

$$then\ our\ four\ state\ space\ system\ is$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ DI_{11}(MgL) & DI_{11}(-2\beta) + DI_{12}(2\beta) & 0 & DI_{11}(2\beta) + DI_{12}(-2\beta) \\ 0 & 0 & 0 & 1 \\ DI_{12}(MgL) & DI_{12}(-2\beta) + DI_{22}(2\beta) & 0 & DI_{12}(2\beta) + DI_{22}(-2\beta) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ DI_{11}(-2\alpha) + DI_{12}(2\alpha) \\ 0 \\ DI_{12}(-2\alpha) + DI_{22}(2\alpha) \end{bmatrix} v$$

3.1 Two-Wheeled Inverted Pendulum Model

NXTway-GS can be considered as a two wheeled inverted pendulum model shown in Figure 3-1.

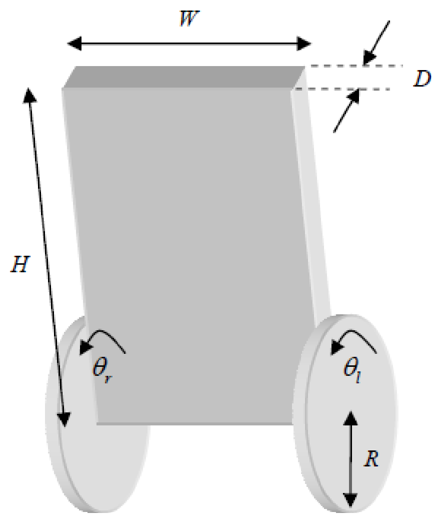


Figure 3-1 Two-wheeled inverted pendulum

Figure 3-2 shows side view and plane view of the two wheeled inverted pendulum. The coordinate system used in 3.2 Motion Equations of Two-Wheeled Inverted Pendulum is described in Figure 3-2.

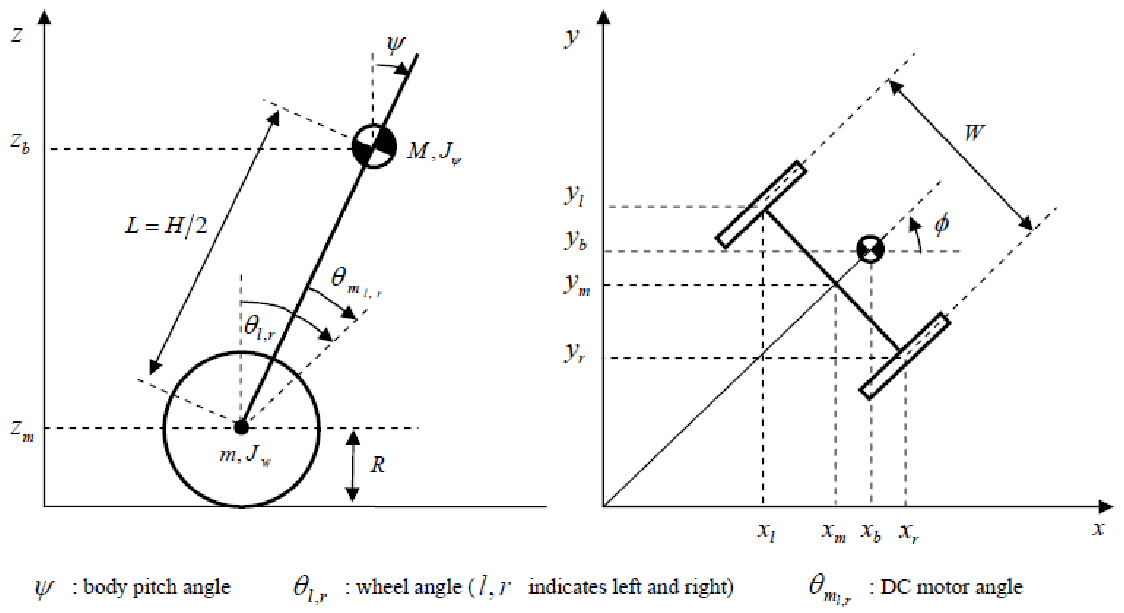


Figure 3-2 Side view and plane view of two-wheeled inverted pendulum

These Figures came from the NXTway paper from Mathworks.

If you have not yet, download this exercises' segbotsimfiles.zip file. Unzip it to a folder on your PC. Then launch Matlab and change Matlab's working directory to the directory where you unzip these files. Run the M-file "equations.m" This file defines all the parameters of the Lab Segbot and finds the A and B matrices of the linear approximation of the Segbot when it is close to its upright balancing position. You should find that it calculates the linear system to be

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \psi \\ \dot{\psi} \\ \dot{\theta} \end{bmatrix} and$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 156.5119 & -80.9696 & 80.9696 \\ -58.0076 & 57.0138 & -57.0138 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ -253.0930 \\ 178.2126 \end{bmatrix} v$$

Before you use this model to design your own controller, use the K gains used in Exercise 3 to balance the actual Segbot, but make sure to multiply each gain by 6.0/10.0 to convert from the PWM units between +/-10 of the actual Segbot to the +/-6 volt units of the simulation. Make sure everything is working on your PC by loading the Simulink file segbot.slx and run the equations.m file if you have not already. (equations.m sets K to the gains given in Exercise 3 for you and multiplies them by 6.0/10.0) Run the Simulink simulation and notice the response. Change the tap to 4.5N and see if the Segbot is able to recover from that tap. Do those gains balance the Segbot in simulation? Find the eigenvalues of closed loop A matrix (A-B*K). Use these pole locations as a starting point for your design below.

Now given the system equations and the simulation, your job is to use the "place" function in Matlab to design your controller v=-K*x. You need to figure out what pole locations will give you controller gains that control the Segbot sufficiently. You will run the command in Matlab, K = place(A,B,mypoles). A and B are given so you just need to find three "good enough" stable poles that allow the Segbot to balance. For this homework our definition of "good enough" is that your controller must be able to keep the Segbot balanced even when an external tap force of 4.5 newtons is applied and the corresponding response has minimal oscillations. An external tap force is applied by setting the amplitude of the block labeled "Finger Pushing At Top of Body". Use the defaults of Period 4secs, Pulse Width 5% and Phase delay 2 secs. Start out with pole locations of the controller that already balanced the actual Segbot and see if you can find more responsive gains (i.e. quicker settling back to zero tilt angle.) In addition to viewing the animation, also view the scope plot of tilt, motor angle and control effort used to balance. Continue to experiment with different pole locations. Find the following:

1. Set of eigenvalues that are more responsive than the given gains. Try these gains on your actual Segbot but make sure to multiply them by 10.0/6.0 to convert between units of the simulation and units of the actual Segbot output. How well do they work compared to the given gains? Remember we are not simulating the Kalman filter so these gains may not work the same as in simulation. I am interested to see what happens. Make a video both of your simulation and of the actual Segbot. Make sure to list your K gain values in the simulation video.

2. Set of eigenvalues that are too responsive and cause the Segbot exert too much control effort and not work well or at all. Do not try these on your actual Segbot. Just make a video of the simulation run. Make sure to show your K gain values in the video.
3. Set of eigenvalues that are not responsive enough and cause the Segbot either to oscillate a bunch or even fall over when given a tap of 4.5 or less. Do not try these on your actual Segbot. Just make a video of the simulation run. Make sure to show your K gain values in the video.

Using your good controller in item 1, try a few things in simulation. First off, change the “Finger Pushing …” block to a 50% pulse width and change the amplitude to 0.4. This is simulating you pushing the Segbot for 2 seconds. Does the response of the Segbot make sense? Why does it lean back when you are pushing it? Create a video of this working.

Second, zero the amplitude of the “Finger Pushing …” block and change the amplitude of the “Psi Offset …” block to 0.2 radians. This is changing the operating point of the linearized system from 0 for Psi to 0.2. This is a method for telling the Segbot to go forward and backward. The offset is applied for 2 secs and then set back to zero. Does the Segbot stop as you would expect? Produce a video of this animation working.