System Characterization of an Inverted Pendulum

Garrett Rodrigues and Daniel Sotingco

May 6, 2009

Abstract

In this paper, we discuss our work in building a toy Segway out of a Lego Mindstorms NXT set. The system is essentially an inverted pendulum problem, and a HiTechnic gyroscopic rate sensor was used as the primary sensor for measuring tilt angle. We found sensor drift to be our primary obstacle during this project, and while the system does not currently balance itself, we have learned much about calibrating sensors, using additional sensors, and implementing shaping filters to correct for sensor drift. In its current state, our robot displays qualitatively correct behavior in attempting to recover a vertical position, but is still unable to balance on its own.

I. Introduction

This report describes the characterization and control of an inverted pendulum system. Our system was made using a Lego Mindstorms NXT kit, and we used a HiTechnic gyroscopic sensor in our feedback loop. We began by deriving governing equations for the plant, the inverted pendulum and the motor, and we modeled the system using a block diagram. Finally, we performed controls systems analysis on the open and closed loop system using tools such as root locus, Bode plots, and Nyquist diagrams. We used these plots to design and implement a compensating control system that would stabilize the plant. In addition to completing the theoretical control system design, another key portion of this project involved compensating for the error that resulted from our sensor.

II. System Characterization

A. Pendulum Characterization

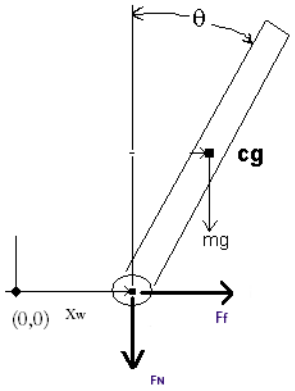

Fig. 1. The inverted pendulum system. We assume the inverted pendulum to be a rod of uniform length with the center of mass at some distance D from the motors.

First, we establish our position vectors to the center of mass of the pendulum. We define our coordinate system at some location on the surface on which the pendulum rests. The horizontal distance to the wheel is defined as X_w, and the vertical

distance to the wheel is 0.

$$
\begin{aligned}
x &= X_w + Dsin\theta && (1)\\
\dot{x} &= \dot{X_w} + Dcos\theta\dot{\theta} && (2)\\
\ddot{x} &= \ddot{X_w} - Dsin\theta\dot{\theta}^2 + Dcos\theta\ddot{\theta} && (3)\\
y &= Dcos\theta && (4)\\
\dot{y} &= -Dsin\theta\dot{\theta} && (5)\\
\ddot{y} &= -Dcos\theta\dot{\theta}^2 - Dsin\theta\ddot{\theta} && (6)\\
& && (7)
\end{aligned}
$$

After drawing free body diagrams, we sum forces on the system.

$$
\begin{aligned}
\Sigma F_x &= F_f && (8)\\
&= m\ddot{x} && (9)\\
&= m(\ddot{X_w} - Dsin\theta\dot{\theta}^2 + Dcos\theta\ddot{\theta}) && (10)\\
\Sigma F_y &= F_N - mg && (11)\\
&= m\ddot{y} && (12)\\
&= m(-D\dot{\theta}^2 cos\theta - Dsin\theta\ddot{\theta}) && (13)\\
\Sigma M_{cg} &= I\ddot{\theta} + B\dot{\theta} && (14)\\
&= D * F_N sin\theta - F_f Dsin(\frac{pi}{2} - \theta) && (15)\\
I\ddot{\theta} + B\dot{\theta} &= -mD\ddot{X_w} + mgDsin\theta && (16)\\
& && (17)
\end{aligned}
$$

After combining equations and rearranging, we find $\ddot{\theta}$ in terms of the other parameters. We then make small angle approximations to linearize the system and take the Laplace transform of the equations in order to obtain the pendulum transfer function.

$$
\begin{aligned}
\ddot{\theta} &= -2 * \zeta * \omega_n * \dot{\theta} + w_n^2\theta + K_p\ddot{X_w} && (18)\\
\frac{\theta}{X_w} &= \frac{K_p s^2}{s^2 + 2\zeta\omega_n s - w_n^2} && (19)\\
& && (20)
\end{aligned}
$$

We define the following parameters to simplify the appearance of our transfer function: $K_p = \frac{3}{4D}, \omega_n^2 = \frac{3g}{4D^2}, 2\zeta\omega_n = \frac{3*B_r}{4mD^2}$.

B. Motor Characterization

At their core, the Lego servomotors that we are using take PWM signals and are controlled by voltage and current, but we found that they are specifically designed to be used in other ways. The lejos NXJ firmware that we used allowed us to set a default speed for the motors. By giving the motors values for the wheel diameter and track width of the model Segway, we used software to command the motor to move forward a desired distance at the specified speed. We tested this by programming the robot to move forward a length of fifty centimeters and measuring the actual distance traveled (since the robot was not yet capable of balancing on its own, we gently supported the model Segway to keep it upright). Over three or four runs, the robot consistently traveled within one centimeter of the target displacement, with human error from holding the system being the most likely source of error. This means that we need not worry about directly controlling the voltage and current into the motor in order to accurately specify torque, speed, and even displacement. Nevertheless, we did look into ways of experimentally determining motor parameters and found that it was impractical. Because the motors plug into the Lego brick with a special 6-wire digital platform cable, we cannot directly apply voltage and current to them without taking them apart. Online documentation from users who had done this revealed that it was very difficult to disassemble the Lego motors in such a way that they could be put back together, as the motors appear to be specifically designed to prevent end users from exposing the internal components [2]. However, we managed to find data from one site that gave the following experimentally determined motor parameters: an armature resistance of 6.8562Ω, a back electromotive force coefficient of 0.46839V·s/rad, a torque constant of 0.31739 N·m/A, and a viscosity resistance coefficient of 1.1278×10^{-3}N· m · s/rad [1].

However, in order to model our system, we used the DC motor transfer function: $\frac{\Omega}{Signal} = \frac{K}{\tau s+1}$. Then, in order to go from angular velocity to linear displacement such that our motor transfer function's output was the input for the plant, we multiplied this by $\frac{R}{s}$, where R is the radius of the motor's wheels.

C. Full Plant

The overall plant is the motor and pendulum system in series, and we can represent this using the block diagram shown in Figure 2.

$$\frac{\theta}{setSpeed} = \frac{K_p s^2}{s^2 + 2\zeta\omega_n s - w_n^2} * \frac{K * R}{(\tau s + 1)s} \tag{21}$$

$$\tag{22}$$

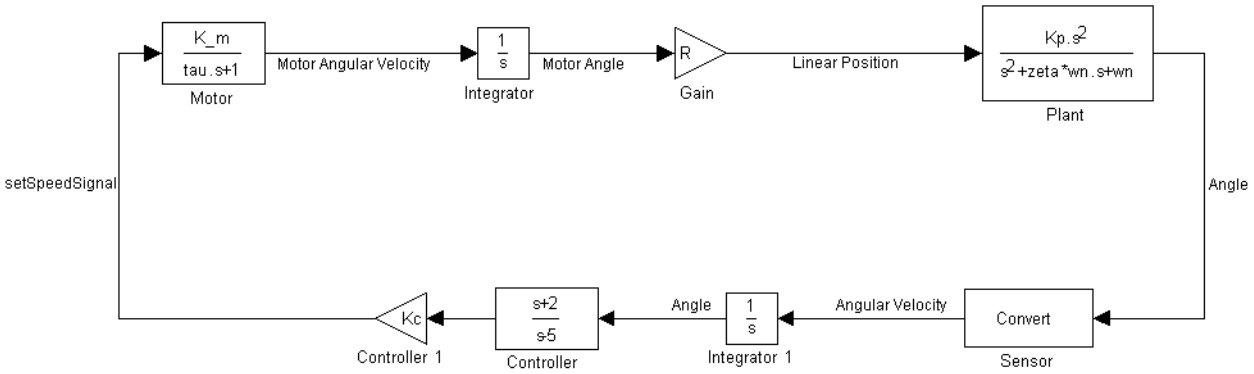

Fig. 2. The above block diagram is a model of the inverted pendulum system, including labeled signal values and transfer functions.

The compensated transfer function included in Figure 2 will be discussed further in the controller design section. We then plot the root locus of the motor-pendulum transfer function, and observe the locations of the poles.

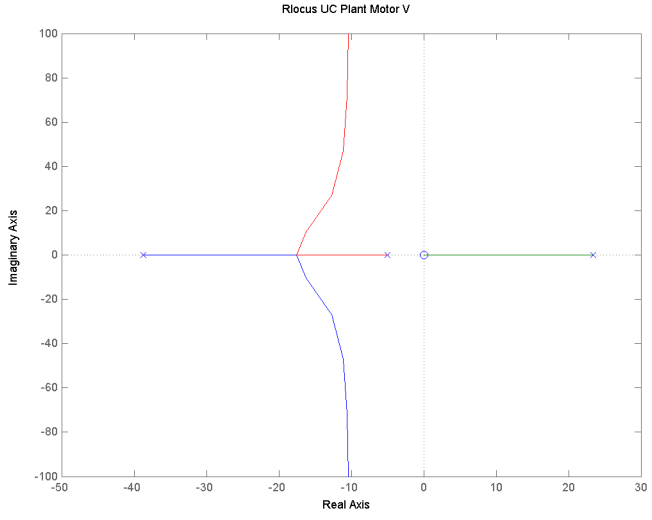

Fig. 3. A root locus plot of our uncompensated plant-motor system.

Two of the poles are stable for all values of K and remain in the left half plane. However, there is one pole in the Right-Half-Plane (RHP) that moves toward the Left-Half-Plane (LHP) for increasing gain, but never makes it out of the RHP plane. Furthermore, the uncompensated system is unstable for all K values and cannot simply be compensated with increased gain.

III. Hardware

The Lego Mindstorms NXT set is the basis for our hardware. As a package with a microcontroller, motors, sensors, and building blocks, it was a natural choice for a Controls project. The heart of an NXT system is the NXT brick, which contains a 32-bit ARM7 microcontroller with 256 kilobytes of flash memory and 64 kilobytes of RAM, as well as an 8-bit AVR microcontroller with 4 kilobytes of flash memory and 512 bytes of RAM. Three ports are available for Lego NXT motors, and four ports are available for NXT-compatible sensors. The NXT set ships with a graphical programming language called NXT-G that is based on LabView. We wanted to use a programming environment that would give us more control over the system, and the two that we considered were Matlab and lejos NXJ. Packages are available for designing NXT control systems using Matlab and Simulink; however, we found that our Matlab builds were missing one required toolbox for these packages, and so we opted to use the lejos NXJ programming environment. This involved replacing the firmware on the NXT brick with the lejos firmware, but allowed us to program the robot using Java. We use two Lego servomotors for our system. Each motor drives one wheel, and the motors are linked together via software. Each motor contains a rotation sensor that can measure displacement. We chose to use the HiTechnic gyroscopic sensor as the primary sensor for our system. The gyro sensor returns a rotational velocity along one axis that can be integrated to provide a measurement for tilt angle. We considered other options, namely the Lego ultrasonic sensor that measures distance and the Lego light sensor. Online forums for Lego enthusiasts indicated that the gyro sensor was the most appropriate sensor for our application.

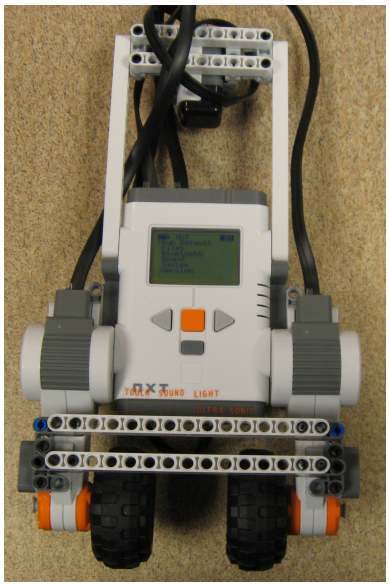

Fig. 4. **Photo of our NXT Segway.** A gyroscopic sensor is mounted above the brick, and bracing members are used at various points to stiffen the chassis. Additional sensors can be mounted in the front or rear.

IV. Control System

After observing the root locus of the uncompensated system, we find that our system needs dynamic compensation. Because of the zero at the origin, we know that the only way to get the RHP pole out into the LHP is to have it break off the real axis. In order to do this, we need another RHP pole. Then, in order to draw the locus of these two poles into the LHP, we add a low-frequency zero into the system. After playing parameter values, we determined the following transfer function to be adequate.

$$G(s) = \frac{s+2}{s-5} \tag{23}$$

$$\tag{24}$$

As is apparent in Figure 5, the root locus appears as expected and for some gain value, the locus of the two open-loop unstable poles become stable.

From Figure 6, we see that in order to get some positive phase margin, we need a gain on the order of 2000.

The Nyquist plot shows a region with two counterclockwise encirclements of the $-1/k$ point. From the equation Z = N+P, we know that in this region, Z = 0, indicating that our system becomes stable (we added an additional RHP pole such that P = 2).

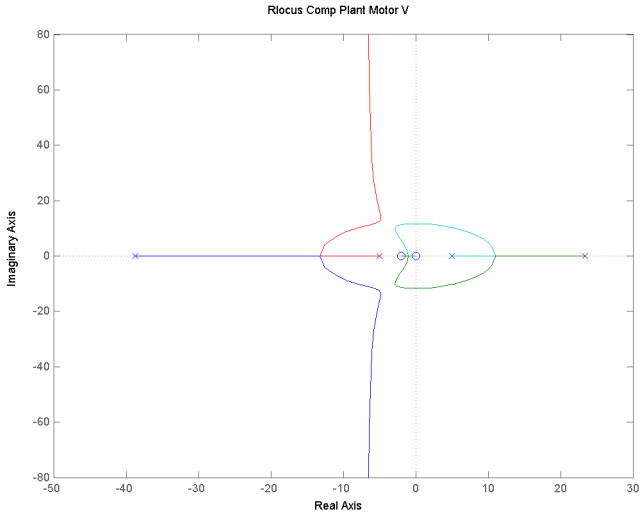

Fig. 5. A root locus plot of our plant-motor system with an implemented control system.

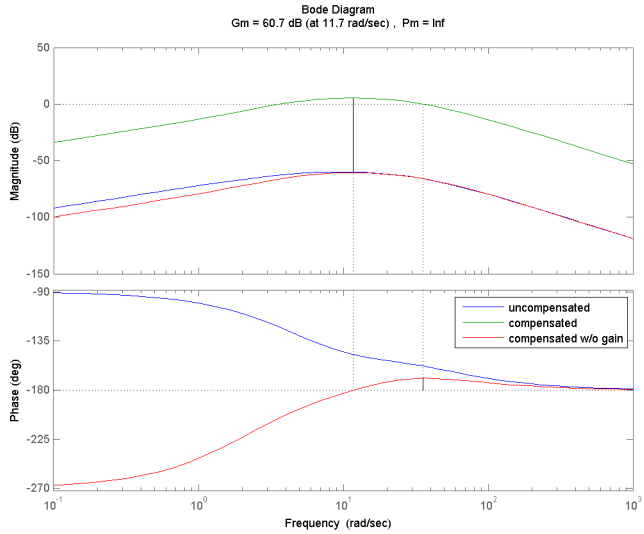

Fig. 6. The bode plot of the uncompensated system, the compensated system parameterized with gain, and the compensated system with a gain of 2000.

V. Gyro Sensor Problems

Integrating the gyro sensor signal to obtain an accurate tilt angle reading was more difficult than we expected. We began by using Euler integration: after each tick of the timer, the gyro sensor reading was multiplied by the timestep and added to the previous angle measurement to obtain the new angle. We quickly moved to fourth-order Runge-Kutta integration to obtain more accuracy. Some experimentation found that the fastest timestep we could use on the brick was 10ms. Below this value, angle measurements turned out to be very inaccurate, and we suspect that the brick was not able to integrate fast enough to keep up with timesteps below 10ms. Once we added additional calculations for our control system, we found that the fastest timestep we could use while obtaining an accurate angle measurement was 50ms. We determined this by rotating the robot about ninety degrees and reading the angle measurement on the LCD screen. For timesteps below 50ms, the angle measurement was consistently about twenty degrees too low. As expected, we also noticed integration error when the timestep was raised too high, and we found 50ms to be an optimal value for accurate angle measurement.

We soon found that sensor drift was a major problem. We had noticed that the gyro would provide a reading of 1deg/s or 2deg/s when it was being held perfectly still, and we adjusted for this offset in our code. This provided us with very accurate tilt angles in the first few seconds of our program's runtime. However, as more time passed, the reading would drift, and the

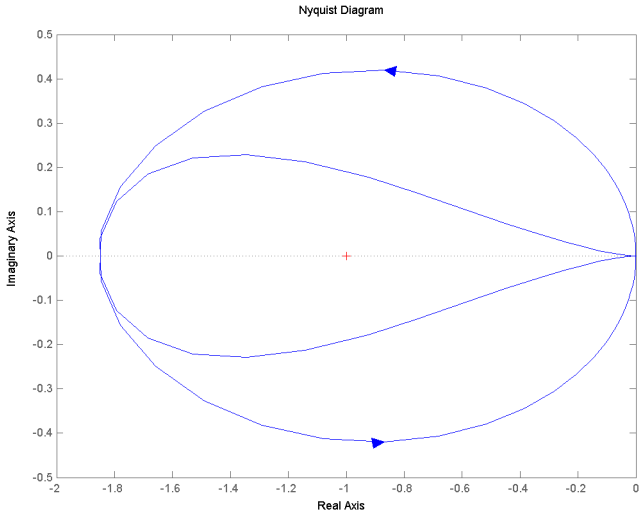

Fig. 7. The Nyquist plot of the compensated system with the gain set to 2000.

angle measurement would be centered at some nonzero angle. This prevented us from being able to tell if our control system was actually doing its job, as the ever-changing offset would cause the robot to topple itself. We did a test to figure out how the sensor drift changed over time and found that the drift decreases roughly linearly over time. We intended to use this test program as part of a calibration algorithm to find the sensor drift at the beginning of every balancing run, but it was never completely integrated into our main program, and would likely be used in future work on the NXT Segway.

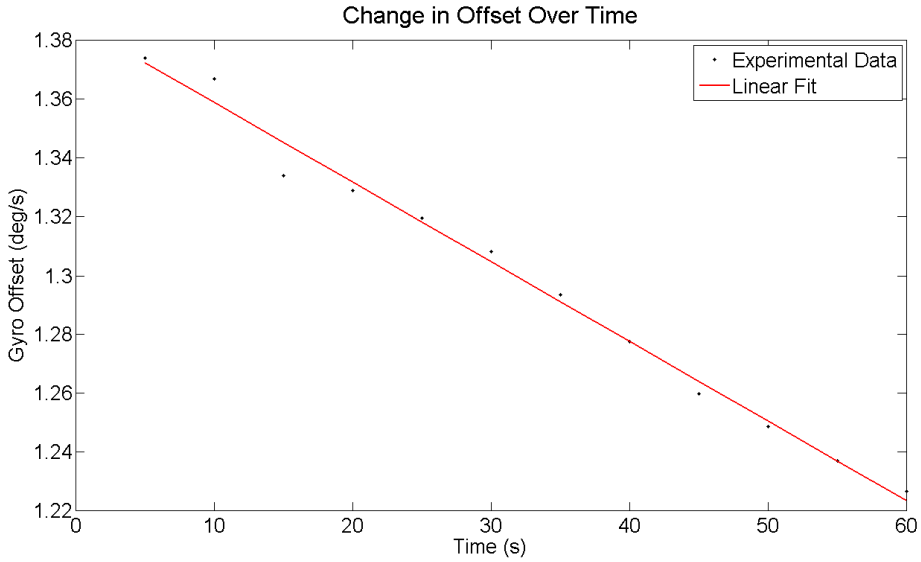

Fig. 8. **Plot of gyro offset over time.** The gyroscopic sensor's drift is not constant and appears to decrease linearly as time passes. The linear fit is given by $y = -2.71 \times 10^{-3}x + 1.39$

We continued our efforts to correct for the sensor drift by using additional sensors. We first installed the ultrasonic sensor onto the system and recorded its distance measurement when the robot was standing vertically. We then reset the tilt angle whenever the ultrasonic sensor reported this distance again. However, the ultrasonic sensor measures distances on the order of centimeters and also turned out to be too slow for this purpose. We then replaced the ultrasonic sensor with a light sensor. Unfortunately, this did not solve our problem because the light sensor turned out to be oversensitive, and noise from ambient light would often interfere with its measurements.

Finally on a suggestion from Daniel Cody, we attempted to use frequency shaping to improve our gyro measurement. We did this by using an integrator that does not have the form $1/s$ in the s-plane, but rather $s/(\tau s + 1)^2$. This has the effect of attenuating low-frequency data, which improves noise rejection, but also has the unfortunate property that all angle

measurements eventually return to zero. The design philosophy behind this decision was that the system would primarily need to reject high-frequency disturbances after being started in the vertical position, and that if our control was good enough at these high frequencies, the measurement's tendency to return to zero would not be a problem. We found that implementing this shaping filter provided us with qualitatively better behavior, as the system would try to right itself more quickly, but still not enough to prevent the robot from toppling over. We tinkered with the value of τ to shift the attenuation in the filter and also tried different gain values for our controller. Some combinations worked better than others, but none were able to keep the robot upright for more than about a second.

VI. Conclusion

Although our NXT Segway did not quite work as planned, the numerous challenges provided us with many learning opportunities. We found firsthand that struggling with sensors is a major part of controller design, and learned of creative ways to correct this problem, such as using calibration, additional sensors, and frequency shaping. We intend to pursue these ideas further in order to make our system work.

VII. Acknowledgments

The authors would like to thank Daniel Cody and Professor Kent Lundberg for their assistance on this project.

References

[1] Watanabe, Ryo. "NXT Motor." http://web.mac.com/ryo_watanabe/iWeb/Ryo's Holiday/NXT Motor.html
[2] "NXT Motor Internals." http://www.philohome.com/nxtmotor/nxtmotor.htm