James Schmidt

Pendulum Control Project

July 16, 2014

Abstract

The purpose of this project was to use technical mathematics and conceptual tools from theoretical controls to identify a model for a mechanical system, an inverted pendulum, and to design a controller to obtain desired behavior. The steps in this process began with system identification, which employed empirical means, and design by theory. The plant was identified to by $G_p = \frac{1}{s^2+s-12}$ and the controller was chosen to be a lead lag controller, namely $G_c = 90\frac{s+1}{s+0.1}\frac{s+5}{s+15}$, both of which together resulted in roughly the following performance specifications: $t_r < 0.25\,s$, $t_s < 1.5\,s$, and $\mathcal{O} \approx 10\%$.

1 Introduction

Feedback is ubiquitous. We live and die according to whether feedback is efficiently operative in our system (body). In engineering, we try to capture by mathematics the way which feedback works. It's important to notice, though, that feedback in many real systems is not mathematical: the body e.g. does not compute the required input from a transfer function. *We* model systems mathematically because by means of it *we* can realize the effects of feedback and thereby imitate many real systems.

Consider an inverted pendulum. It is identical to the ordinary pendulum except that its equilibrium point (vertical) is unstable rather than stable. But that instability can be removed by feedback: continuously maintaining (or trying to maintain) a position near the origin. It is a good model of human walking. However, we walk rather smoothly and have little trouble responding to disturbances. In this project, our aim is to implement a controller to stabilize the pendulum about its equilibrium point. Naturally, our efforts won't result in a product as feedback-saavy as the bipedal species. The reason forthis are threefold:

1. The mathematical model of the system does not exactly represent system dynamics.
2. The dynamics of the system are exaggerated (moment of inertia disproportionate).
3. The actuator is limited.

In what follows we will run through the mathematical derivation of the model to be used in the control design. This part will be entirely empirical: we will look at the system response to a step input and match parameters assuming a canonical system type, namely that the inverted pendulum acts like a second order system. The reason for making this assumption is that the system response *looks* like the system response of a second order system. But, as a precautionary note, it's not one! Then we will run through a design of the controller, which was also empirical. The theory guiding the design isn't (it's math), but determining a controller which actually works is, as will be discussed in §3. Finally we will show the results of the implementation and compare it to theoretically expected response. For the latter, we used both plots from Matlab and simulations from Simulink.

2 System Identification

2.1 Empirical Correlations

The following figure shows the empirical data of the pendulum for a step input of unitless input torque $u = 100$. We used the original code, varying from $-20°$ to $+20°$; another plot in Appendix B shows similar results varying the step input between $u = 300$ and $u = 0$. The results were averaged and given as follows

$$\zeta \approx 0.12 \tag{1}$$

$$\omega_d \approx 6.3 \tag{2}$$

$$\omega_n \approx 3.4 \tag{3}$$

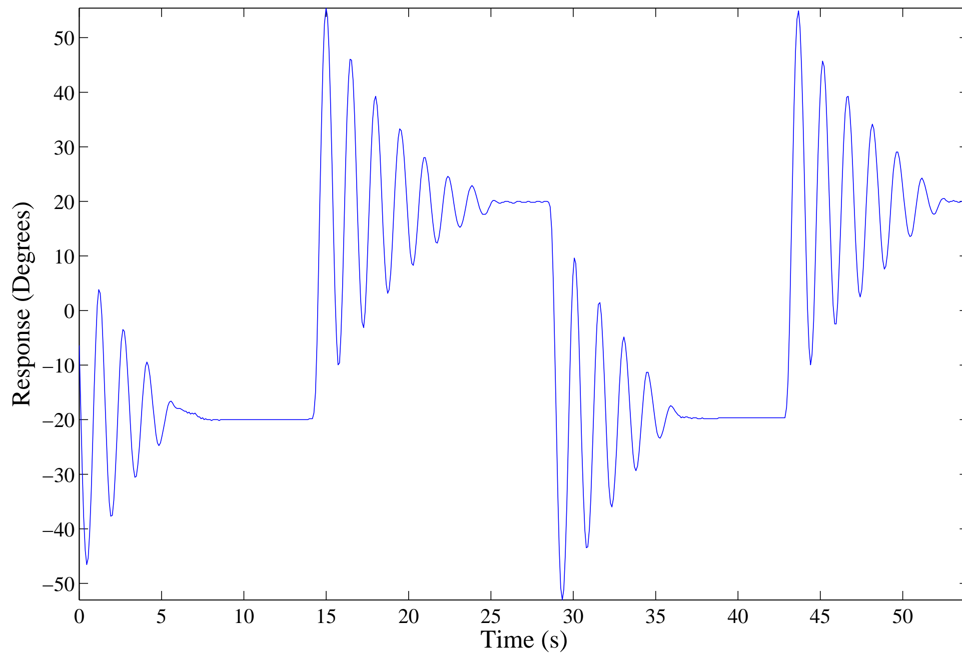


Figure 1: System Identification with $u = 100$

They were obtained by using the following two equations:

$$t_p = \frac{\pi}{\omega_d} \tag{4}$$

and

$$t_r = \frac{1.8}{\omega_n} \tag{5}$$

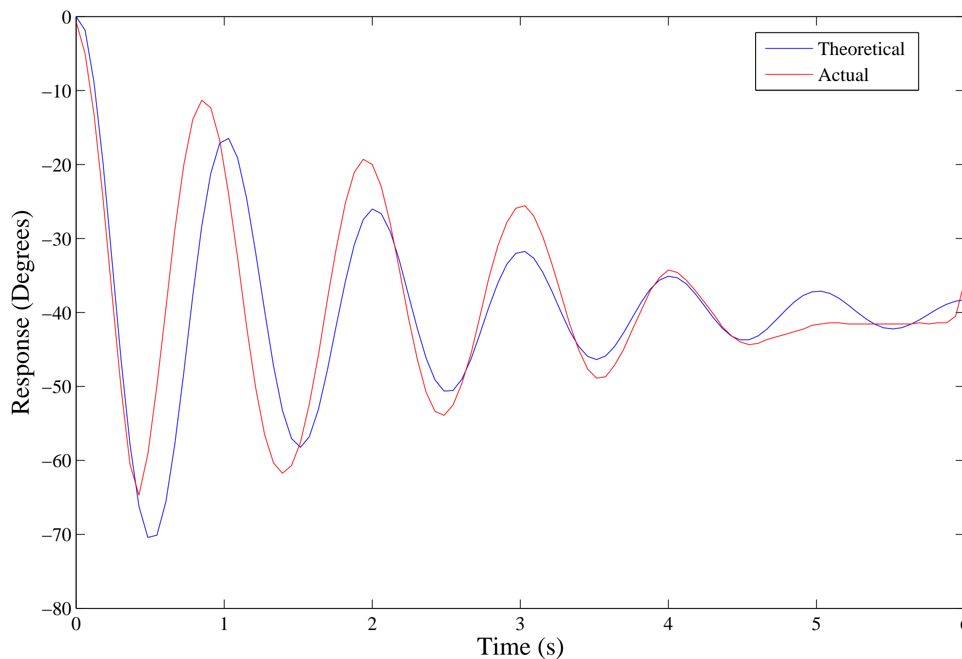


Figure 2: Experimental and Theoretical System

both from [1], and Figure 9.24[1], which is verified by

$$t_s = \frac{3}{\zeta \omega_n}. \tag{6}$$

As an aside, since the numerical values range from $-50°$ to $50°$, overshoot had to be normalized by some constant, for which we used the absolute range of motion of the pendulum ($50°$). It is important to point out exactly what these equalities mean. We are taking empirical data of a physical system and trying to fit it into a *model* of a second order system, whose transfer function is given by the equation

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}. \tag{7}$$

It is the second order system for which the use of these equations hold apriori (and even then, they are not exact but [good] approximations). We can assume that a theoretical second order system approximates the real system if in our extrapolation from the data of second order parameters, the associated second order system actually does line up with the real system. Figure 2 above shows their correspondence.

The results are fairly accurate. The theoretical overshoot is greater than the actual overshoot, but the amplitude following for the theoretical is greater than the actual. Thus perfectly matching the real system behavior with a second order model is impossible, and we have a trade-off between attempting to capture more of the tail or the front end. We opted for in between; so though not ideal, the result is nevertheless sufficiently approximate. Whether 'sufficient' is satisfactory will be made more apparent in the implementation of a controller. We should be able to predict with

reasonable uncertainty (namely, uncertainty corresponding to the uncertainty of the second order model) how the controlled system will behave.

The theoretical system is given by

$$\ddot{x} + \dot{x} + 3.5 = 0 \tag{8}$$

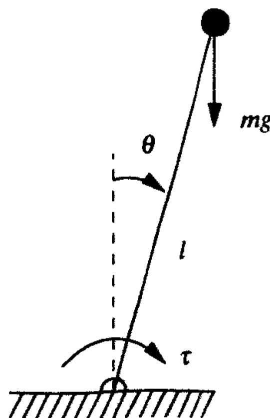
with plant (system) transfer function

$$G_p(s) = \frac{12}{s^2 + s + 12}, \tag{9}$$

the roots of which are approximately $-0.5 \pm 1.8i$. Again, given uncertainty and theoretical concessions, values in these equations are approximate and rounded for simplicity.

2.2 Inverting the Pendulum

Consider the following figure, borrow from [1] 10.11. The equations of motion are given by



$$ml^2\ddot{\theta} = mgl\sin(\theta) + \tau \tag{10}$$

Contrast this from the case of the non-inverted pendulum, for which the equations of motion are given simply by

$$ml^2\ddot{\theta} = -mgl\sin(\theta) + \tau. \tag{11}$$

The difference between the two equations consists in the sign preceding the gravity term. More importantly, these equations of motion can be linearized as

$$ml^2\ddot{\theta} \pm mgl\theta = \tau \tag{12}$$

for $\theta \approx 0$. Bracketing for the moment that the behavior of our system will *not* remain close to zero, the transfer function of input to output is given by

$$\frac{\theta}{\tau} = \frac{1}{ml^2 s \pm mgl} \tag{13}$$

where reference to (10) and (11) indicate that '+' refers to the noninverted pendulum and '-' to the inverted. Furthermore, this equation does not contain a damping term, but supposing it did, the gravity term would not be in it. So what falls out of this discussion is that the direction of gravity is responsible in changing the equations of motion between the inverted and noninverted case, and the result is that the θ term (not $\dot{\theta}$ or $\ddot{\theta}$) is altered only in sign.

With this preamble, we conclude that the transfer function for the inverted pendulum which we will control is

$$G_{p,inv}(s) = \frac{12}{s^2 + s - 12} \tag{14}$$

and the corresponding system dynamics are given by

$$\ddot{\theta} + \dot{\theta} - 3.5\theta = 0 \tag{15}$$

the roots of whose characteristic function are $\{3, -4\}$.

The following two plots are the root locus plots for the uncontrolled non-inverted and inverted pendulum, respectively.

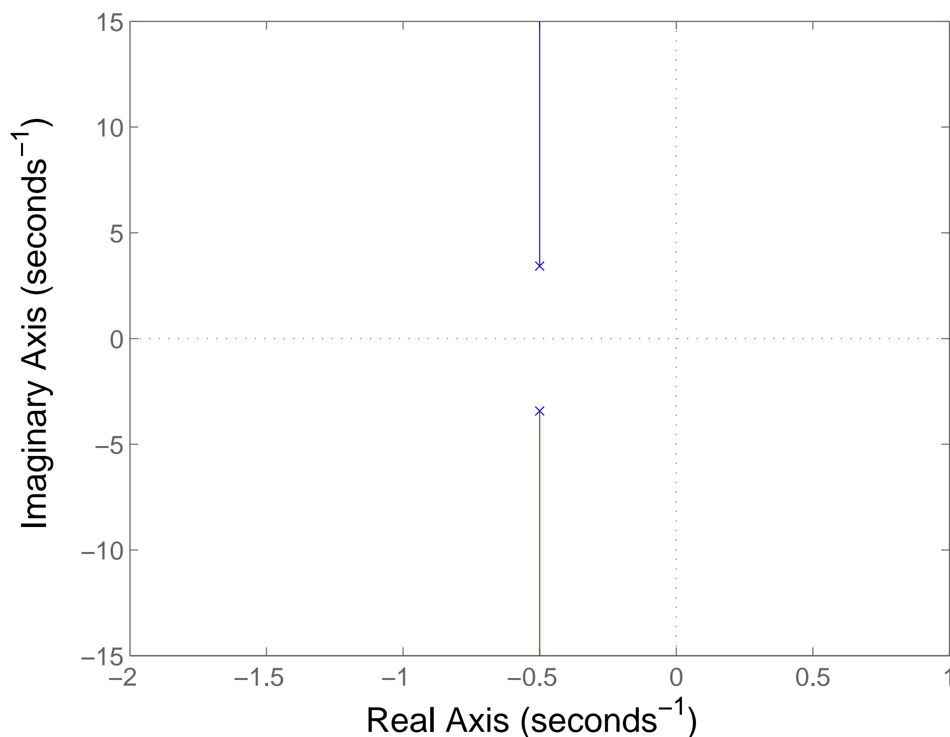


Figure 3: Root Locus for $G_p = \frac{12}{s^2+s+12}$

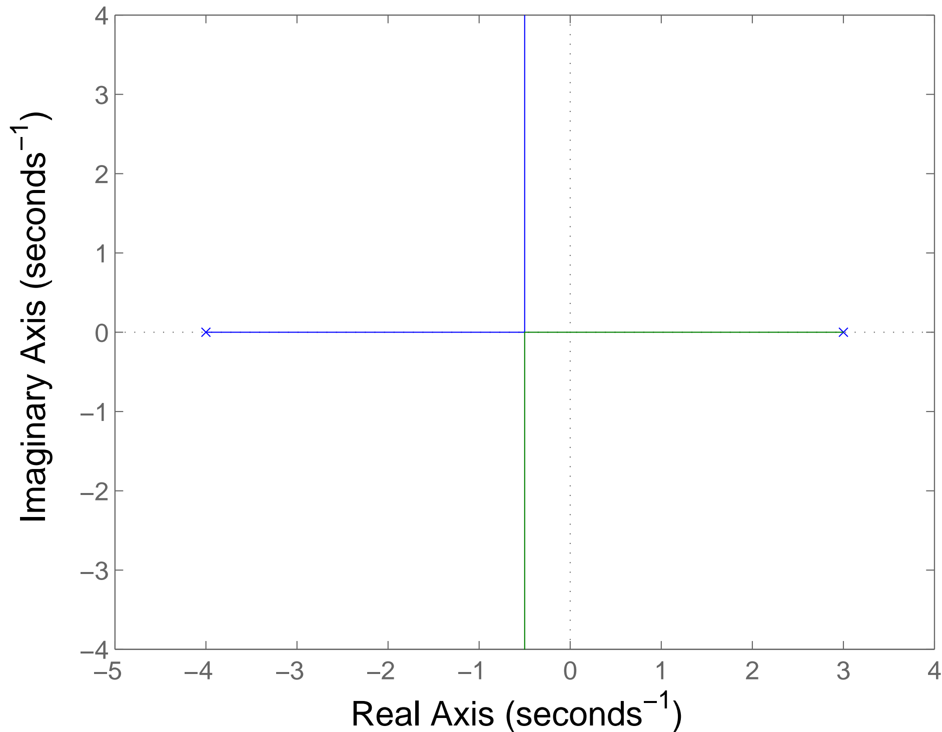


Figure 4: Root Locus for $G_p = \frac{12}{s^2+s-12}$

3 Designing a Controller

It's easy to see that in theory we can choose a controller to satisfy unrealistic performance specifications. Consider, e.g., a lead lag controller

$$C_{ll} = 10,000 \frac{s+.1}{s+.001} \frac{s+1}{s+100} \tag{16}$$

Its behavior is shown in the following plot.

'Unrealistic' here has a technical meaning. The controller we implement isn't working simply by computer simulation. It operates in a physical system through a processor with limited computing power, and a motor with limited actuating power. These limits include speed of computation, memory allocation, allowable output torque, communication clarity (noise), etc. Since the controller is working at $20\,Hz$, it's more likely that the motor, not microcontroller, is the real culprit for any limits on system behavior.

Therefore, our ambitions must be reigned. Theoretically we can be aggressive as we want and design a controller with $t_r < 0.001\,s$ and $\mathcal{O} < 1\,\%$, but most likely the motor won't be able to

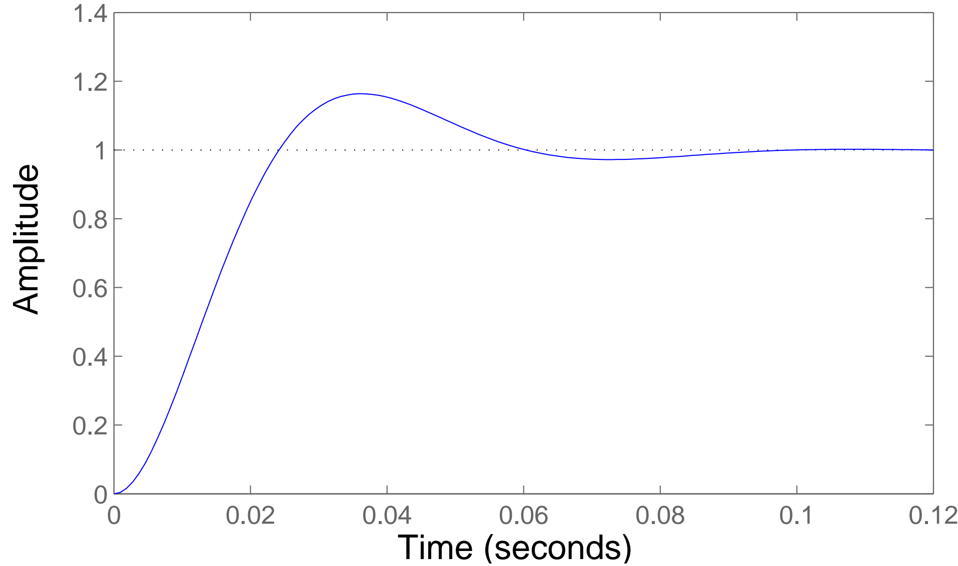


Figure 5: Expected System Behavior with 'Unrealistic' Controller

handle it.[1] In fact, this was verified by experiment, using a lead lag controller

$$G_c(s) = 20\frac{s+2}{s+8}\frac{s+1}{s+0.01} \tag{17}$$

as shown in Figure 6.

However, the expected response wasn't even that unreasonable (c.f. Figure 7), especially compared to a controller which did in fact work as desired (c.f. §4).

Reference to Figure 4 will be necessary to consider a design of a controller. Note that as it stands, the closed loop system is unstable. We didn't encounter any apparent instabilities in §2.1 because we weren't looking at the behavior of the closed loop inverted system. In fact, we weren't considering the inverted system at all. For the hanging pendulum, both the open and closed loop transfer function are stable. The closed loop inverted system can be stabilized alone with a sufficiently high gain, but in this case the behavior still isn't desirable (close to the real axis it will take *forever* to reach the desired value and if we ramp up the gain a little to move the poles farther from the origin, then it will very quickly oscillate like crazy).

Therefore, in order to proceed our first task will be to pull the root locus plot to the left. However, taking into account the discussion of the divergence of real behavior from theoretical, our approach won't be quite so calculated as it could be. To reiterate, the reason for this is that we can't determine apriori whether a designed controller will in fact work as expected in the real

[1] It's worth noticing that these kinds of limitations are present everywhere in engineering systems. I would even say from my experience as an undergrad that half of engineering practice is not applying theory but deciding what to do when blockades require preference on tradeoffs. For senior design, e.g., we wanted a motor which was both really torquey and really fast, but unfortunately- unless you want to shell out more cash for a monster motor- opting for one *ipso facto* results in sacrificing the other.

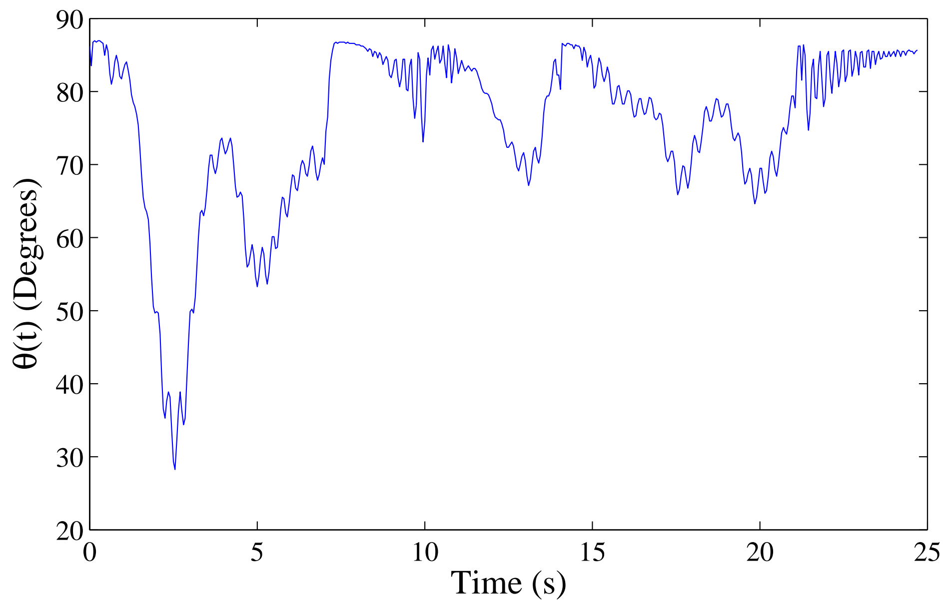


Figure 6: Actual Response with Aggressive Controller

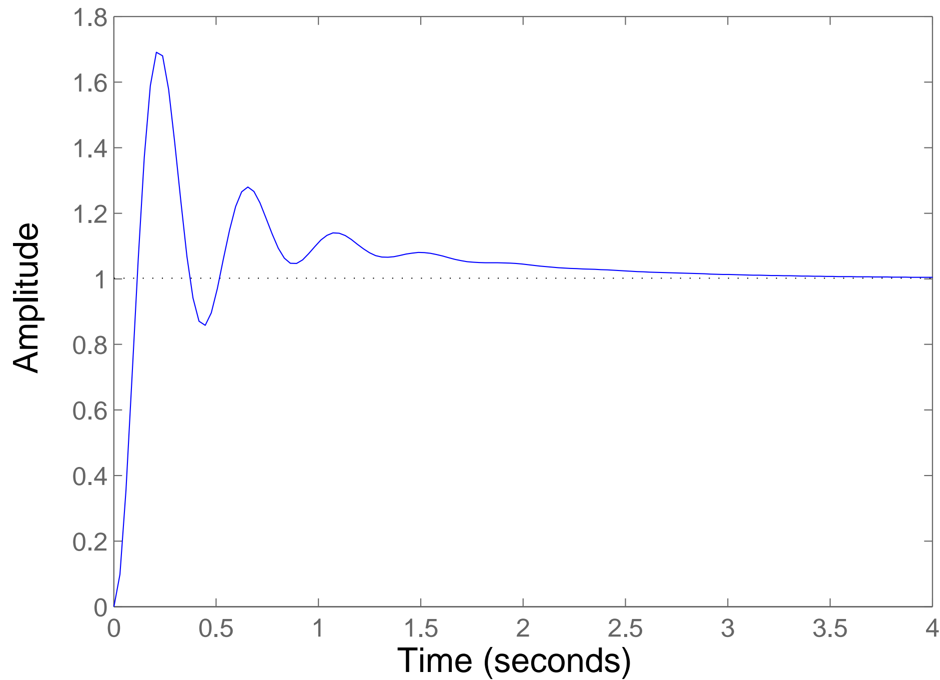


Figure 7: Expected Response with Aggressive Controller

system. Therefore, we have to use a trial and error approach: design a controller with satisfactory system parameters, upload it into the controller, and see if it works. If it doesn't, then design a more modest controller.

The fundamental philosophy to designing a controller is: pull stuff to the left and down towards the real axis. Together these decrease overshoot, rise time, and settling time, all of which are desirable features for almost any system behavior.[2]

4 Implementation

In this section we present the results of a controller which we managed to successfully implement. A video of the first controller used can be found at `http://www3.nd.edu/~aschmid5/`.

We used a lead lag controller with

$$G_{lead}(s) = \frac{s+5}{s+15} \tag{18}$$

and

$$G_{lag}(s) = \frac{s+1}{s+0.1}, \tag{19}$$

the root locus of which with our system can be found in Appendix B. A sufficiently high gain needs to be chosen to both stabilize the system and provide for satisfactory performance, but it is evident that if too high a gain is chosen then performance will not meet specifications: overshoot will be too high and there will be little damping.

In cases where controller limits aren't immediately observable, or where those limits are able to be more precisely correlated before hand with numerical values, and where actuation of an overly aggressive controller could break components of the system, it would be advantageous to employ more careful foresight in the design stage than we did. In our case, the worst that could happen is that the pendulum breaks off, and luckily (admittedly, I say this perhaps too flippantly) that didn't happen. If it did, then I guess I would have to say: it would be good to be careful ahead of time!

Our Controller was

$$90\frac{s+5}{s+15}\frac{s+1}{s+0.1} \tag{20}$$

As it stands, this transfer function is useless. We need to translate the transfer function from the continuous domain, where did the analysis, into the discrete time domain where the computer operation lives, so that it can be used in a program. The programed controller runs through a loop and at each iteration compares data from previous time steps; it works because the difference in values over a short time interval approximates derivatives.

To perform the transformation, in Matlab we used the 'c2d' function with a frequency of $20\,Hz$, and Tustins method, the result of which was

$$G_{c,dis} = \frac{0.8365z^2 - 1.446z + 0.6189}{z^2 - 1.45z + 0.4523} \tag{21}$$

[2]To be precise, overshoot is related to the angle which poles make with the real (or equivalently, imaginary) axis, whereas rise and settling time are related to distance from origin and imaginary axis, respectively.

Note that this transfer function is the transfer function of the controller; we need only it because $u = G_c e$- it represents the ratio of input to error= and we have knowledge of these values during each loop. Even though the plant dynamics don't appear at all in our program, they directed our design of the controller (so implicitly they're there).

Multiplying out, we obtain

$$(z^2 - 1.45z + 0.4523)U = (0.8365z^2 - 1.446z + 0.6189)E \tag{22}$$

Since z's represent time-shift, the update law is given as

$$u_{new} = (8365err_{new} - 14460err(T-1) + 6189err(T-2) + 14500u(T-1) - 4523u(T-2))/10000 \tag{23}$$

As it stands, we can't accept this new torque without question, because there's a limit to how much the motor can accurately output. Therefore, we need to saturate u:

$$u = \begin{cases} sgn(u) \cdot 400 & \text{if } |u| > 400 \\ u & \text{otherwise} \end{cases} \tag{24}$$

Obviously, if the system wanted to output more, and we don't let it, then it's not going to respond in exactly the way we want it to. We might as well catalog the causes of nonlinearity:

1. Saturation in motor actuation.
2. Assuming- not entirely incorrectly- that the physical system (i.e. inverted pendulum) is second order.
3. Noise in the system.
4. Physical inefficiency, e.g. friction in components.

Figure 8 shows the implemented controller for our system, at a variety of angles. Intuitively, the performance seems more like what would be expected (c.f. Figures 9 and 10) for small angles and deviates for larger angles. We expect it to deviate for large angles since the derivation which mathematically made a second order approximation plausible relied on that $\theta \approx 0$, which clearly doesn't hold for large angles.

The expected performance specificatios are verified by the Figure 11, for which the expected rise time is about $t_r \approx 0.3\,s$. The overshoot is similar, with the behavior overshooting only once and remaining below for the duration of its response. The actual response oscillates a good deal more than the theoretical, but again, this disparity can be due to the innumerable nonlinearities listed above.

References

[1] B. Goodwine. *Engineering Differential Equations.* Springer, 2010.

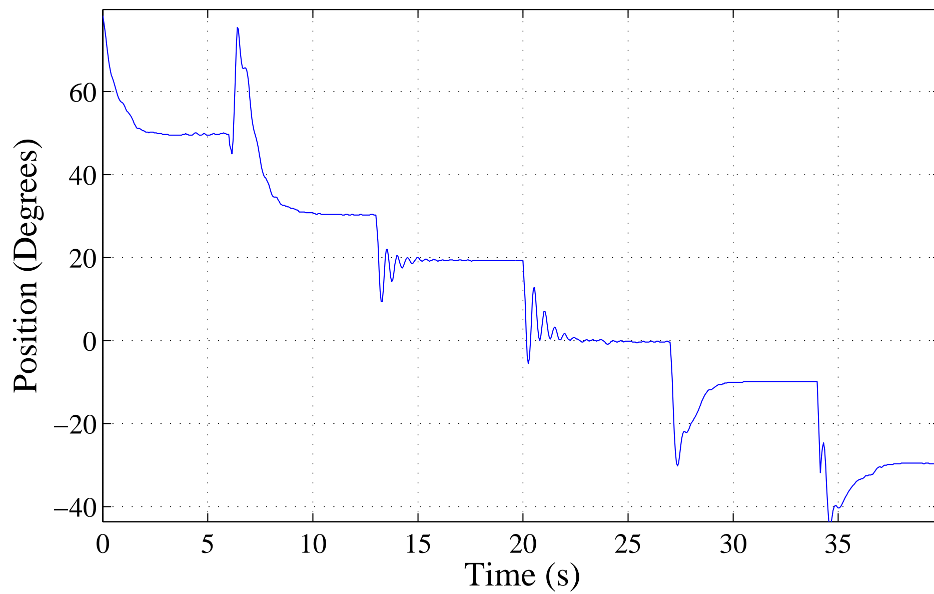


Figure 8: Basic Lead Lag Controller Behavior for System

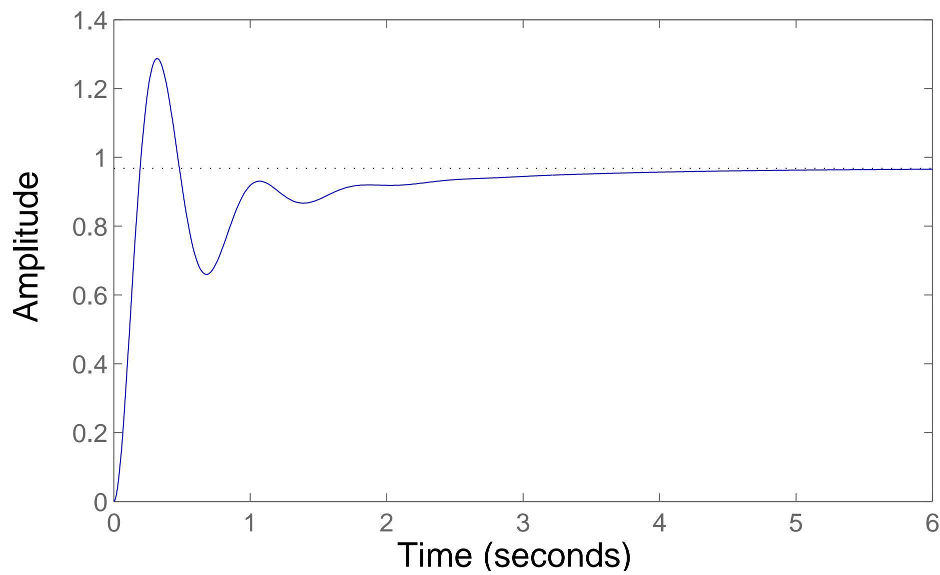


Figure 9: Expected Response with Lead Lag Controller, with 'step(feedback(*,*))' Command

5 Appendices

5.1 Appendix A: Code

```
/* Authors:
   Bill Goodwine, April 6, 2009.
```

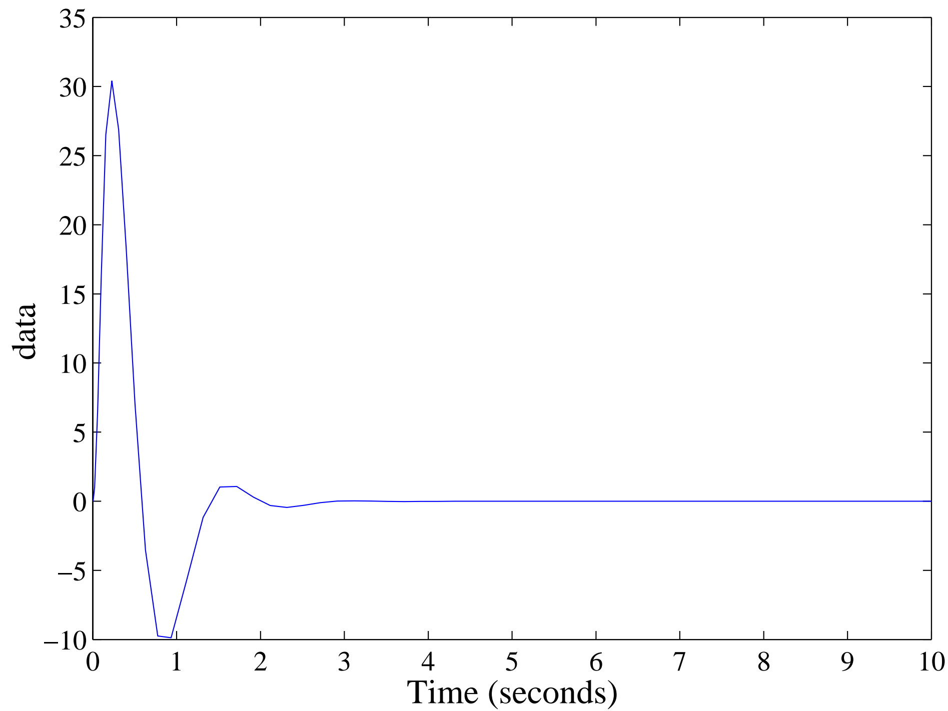


Figure 10: Expected Response with Lead Lag Controller, using Simulink

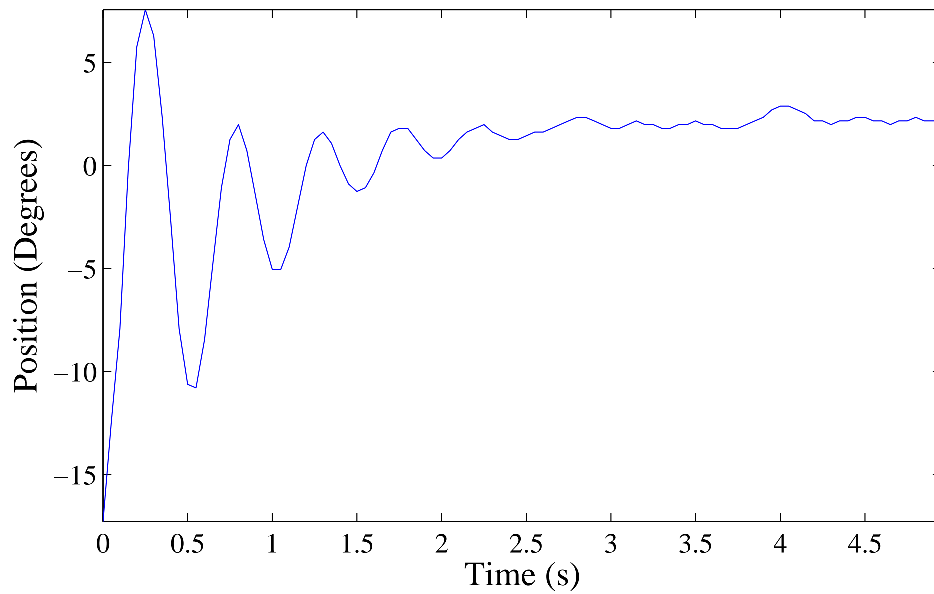


Figure 11: Actual Response with Lead Lag Controller

```
   Raymond Le Grand, May 26, 2010.
   Blair Rasmus, Derek Wolf, John Gallagher, November 13, 2011
   James Schmidt April 30,2014
*/
```

```
#include "hc11.h"
#include "mc.h"
#include <math.h>
#include "vectors.c"
#include "serial.c"
#define CW 0  //Defines direction of pendulum movement (encoder)
#define CCW 1
#define OFFSET 1830
//This is the difference between the encoder zero and the pendulum straight up position.
//This may change slightly with each pendulum.

#define SCALE 18
/* the SCAlE constant represents the scale of the position decoder,
which is degrees per signal tick, but since the microcontroller only does integer math,
we will define the scale as an integer and divide by 100 every time.
*/

#define MAX_U 400
/*
   The MAX_U constant represents the maximum amount of PWM signal that the system can handle,
without the signal being so fast that there are current/voltage spikes.
It is strongly recommended that this value not be changed.
*/
#define CONTROL_LOOP_FREQ 20 //Frequency of control loop calculations in Hz
#define CLOCK_FREQ 9830400
#define PWM_FREQ 880
// Initializing controller variables
long pos=0, pos_deg=0, u=0, u1=0, u2=0, err=0, err1=0, err2=0, r=0;
//keeps track of current angle. 'pos' is in encoder counts, and 'pos_deg' is in degrees*100
// Initializing PWM variables
unsigned int counts_total=((long)CLOCK_FREQ/4)/PWM_FREQ, t=0;
// counts_total is used for the counter for the PWM interrupt.  This should give a 880Hz interrupt.
// We divide the clock frequency by 4 because the counter increments
//every fourth clock cycle when using a prescale of 1
unsigned int counts_high;
unsigned int counts_low;
// Initializing Timing Variables
unsigned int PWM_interrupt_scale=PWM_FREQ/CONTROL_LOOP_FREQ;
 //Sets the ratio of PWM interrupts to control loop interrupts
unsigned int PWM_interrupt_counter = 0; //this keeps track of ticks from control loop interrupt, used for timing
long k=80; // This is our gain. Ideally it would be something like 20,
 //but we're limited by the physical control mechanism.
void printpos(void);
void setnew(int val);
int main(void)
{
//Initializing controller variables
long u=0; //This is what we use to store the calculated value for torque that we need
 // Initialize hardware
init_ports();
init_interrupts(); //this also init's the interrupts for tracking position
set_torque(0); //starts out at 0% torque
 pause(brief); // power-on delay
init_serial(); //Initialize serial communication
welcome(); //Display welcome message
pause(brief);
set_zero();
while(1)
{
//this checks to see if the pendulum is in top position,
//which allows for greater position accuracy
if(check_encoder_top()){
pos = OFFSET/18; //pendulum has reached center, so reset position to zero + OFFSET.
}
PORTA ^= 0x10; //0b01000000; //toggle pinA.4 on/off to show user that interrupt is 20Hz with blinking LED
if(PWM_interrupt_counter>=PWM_interrupt_scale/*control_loop_limit*/){
 //this checks to see if it is time to do 20Hz control calculations
```

```
PWM_interrupt_counter=0;
PORTA ^= 0x40; //0b01000000; //toggle pinA.6 on/off to show user that interrupt is 20Hz with blinking LED
//////////////////////// 20 Hz Operations//////////////////////////////////
//////This where you need to calculate/set the torque///////////////////
//pos_deg=pos*((int)SCALE); //calculate the current position in degrees*100
out_unsigned_dec(t);
out_string(" ");
if(pos<0){
out_string("-");
out_unsigned_dec(-pos);
}
else{
out_unsigned_dec(pos);
}
out_string(" ");
out_unsigned_dec(u);
carriage_return();
err=(r-pos)*k;
 //u = (8416*err -13840*err1 + 5565*err2 +13910*u1 -3910*u2)/10000;
u=(8783*err-16640*err1+7867*err2+16640*u1-6650*u2)/10000;
//setnew(u);
if(u>400){
u=400;
}
if(u<-400){
u=-400;
}
set_torque(-u);

u2=u1;
u1=u;
err2=err1;
///////////////End of 20Hz Operations////////////////////////////////////////
t=t+50;
if(t==10000){
r=-100;
}
if(t==20000){
r=0;
}
if(t==30000){
r=110;
}
if(t==40000){
r=150;
}
if(t==50000){
r=200;
}
if(t==60000){
r=300;
}
}

}
}
//**** This begins the Interrupt Code ****/
// Programing interupts for PWM
 void OC3_handler(void){
if(!(PORTA & OC3)){  //(portA.5==low) so set the TOC3 to
//\\the time at which we want to end the low part of the PWM cycle
TOC3 = TOC3 + counts_low;
}else{
TOC3 = TOC3 + counts_high; // Set TOC3 to the time at which we want to end the high part of the cycle
PWM_interrupt_counter++;
}
    TFLG1 |= OC3;
 }
```

```
// Programming Interrupts for Tracking Movement
void PAI_handler(void)
{
//this checks direction of pendulum, then increments position variable
if((PORTA & 0x02 /*0b00000010*/) == 0){
pos++;
}else{
pos--;
}
TFLG2 |= PAIF; //reset interrupt flag
}
/* default interrupt handler (empty, just returns) */
void default_handler(void) {}
void printpos(void){
if(pos<0){
out_string("-");
out_unsigned_dec(pos)
}
else{
out_unsigned_dec(pos);
}
carriage_return();
}
/**** End of Interrupt Code  ****/
// Function to initialize PWM
void init_interrupts(void){
// this also initializes position encoder
        asm(" sei"); //disable interrupts
        BAUD=BAUD9K_Turbo; //Use BAUD38K for non-turbo mode of microcontroller
// set register to next time for each interrupt
    TOC3 = TCNT + counts_total;
// arm all interrupts
TMSK1 = 0x0;
       TMSK1 |= OC3;
//pulse accumulator setup: used to receive signal from decoder that gives pendulum angle
TMSK2 |= 0x10; //0b00010000; this enables pulse accumulator interrupt
// acknowldege all interrupts, in case they were already triggered
TFLG1 |= OC3; //flag for pulse accumulator
TFLG2 |= PAIF; //flag for pulse accumulator
PORTA |= (OC3); //start off both PWM ports high
TCTL1=OL3; /*want PORTA.5 to toggle every time there's an interrupt, but nothing else*/

TCTL2=0xC0; //0b11000000; // this turns on error checking from h-bridge on pinA3

        asm(" cli"); //enable interrupts
}
// Function to set PWM duty cycle, which changes torque
void set_torque(long p_rate){
// Accepts desired Torque percentage as an input,
//and uses the global direction flag to know which direction to apply torque
while((PORTA & OC3)); //while (portA.5==high) do nothing,
//b/c want to wait until low cycle has started,
//which means that we can load next high-low cycle without messing up PWM period
// This calculation is 50% + (p_rate%).
// 50% PWM = 0torque, and 95% PWM is Max torque in CW direction.
p_rate=((unsigned int)(counts_total/10)*p_rate)/(unsigned int)10;
counts_low=(unsigned int)counts_total/(unsigned int)2-p_rate/(unsigned int)10;
//divide p_rate by ten to get it as 0-40 instead of 0-400
counts_high = counts_total-counts_low;
}
// Read direction signal
unsigned char check_encoder_dir(void){
unsigned char dir_flag;

if((PORTA & 0x02/*0b00000010*/) == 0){
dir_flag=0;
}
else{
```

```
 dir_flag=1;
}
return(dir_flag);
}
// Read vertical position sensor (tells when pendulum is vertical)
int check_encoder_top(void){
int top_flag;
//this checks pin A.2 to see if pendulum is vertical
if(!(PORTA & 0x04/*0b00000100*/)){ //note that this line was inverted to account for top signal being inverted
top_flag=0;
}
else{
top_flag=1;
}
return top_flag;
}
// Pause function waits for specified number of clock cycles before continuing
void pause(unsigned int duration)
{
unsigned int time;
time=duration;   // small delay routine
while(time>0)
time--;
}
void setnew(int val){
if(val>400){
val=400;
}
if(val<-400){
val=-400;
}
}
// Initialize ports
void init_ports(void)
{
/* enable pulse accumulator on PA7, falling edge
PA3 is input capture IC4 */
PACTL = 0x40; //0b01000100;
//PACTL = 0b00000100; //this is the code to disable it
PORTA = 0xCF; //0b11001111; // disable H-bridge, photointerrupter
DDRD =  0x07; //0b00000111; // sets D0-D2 as outputs, the rest are inputs
PORTD = 0x04; //0b00000010; // clears Port D
PACNT = 0x00; //0b00000000; // clear Pulse Accumulator
}
// Function to prompt user to move pendulum through zero to initialize angle counter
void set_zero(void){
     int de_ch=0,index=0;
 while((PORTD & 0x08 /*0b00001000*/) == 0);   // make sure ok button has been released
 out_string("Move Through Vertical Position");
 carriage_return();
 pause(SECOND);
 pause(SECOND);
 pos=0;
 out_string("DIR   POS ");
 carriage_return();
     while(check_encoder_top()==0){ //checks to see if pendulum is at top position
//if pendulum not at the top, then keep looping
de_ch=check_encoder_dir(); //check the direction of the pendulum
if(index==1000){ // only updates screen every 1000 iterations
out_unsigned_dec(de_ch); //print out direction
//output position, account for positive/negative numbers
if(pos>=0){
out_string(" ");
out_unsigned_dec(pos*SCALE/100);
}else{
  out_string("-");
  out_unsigned_dec(-pos*SCALE/100);
}
```

```
out_char(NEWLINE); //go back to column zero, but same line
index=0;
}else{
index++;
}
}
//pendulum has reached the top,so stop looping
pos = OFFSET/18;
carriage_return();
out_string("you finished!");
carriage_return();
pause(SECOND);
}
void welcome()
{
pause(brief);
out_string("AME 30315");
out_string(" Pendulum Project ");
carriage_return();
carriage_return();
}
// initialize MicroStamp 11.  This function is called by _start, which is defined in crs0.s
// A __premain() is created by default by GCC compiler, but we have overwritten the default
// so that we can move the register block, which must be done within first 64 bus cycles
void __premain(void)
{
*(unsigned char volatile *)(0x3D) = 0x01; //Register block will start at 0x1000 instead of default 0x0000
TMSK2 = 0x0C; //0x0D; // =1101b, a prescale of 4 for the output compare,
// which must be set within 64 cycles of microcontroller reset,
// which is why we set it here
CONFIG = 0x04;  // disable COP timer
}
```

5.2 Appendix B: More Plots

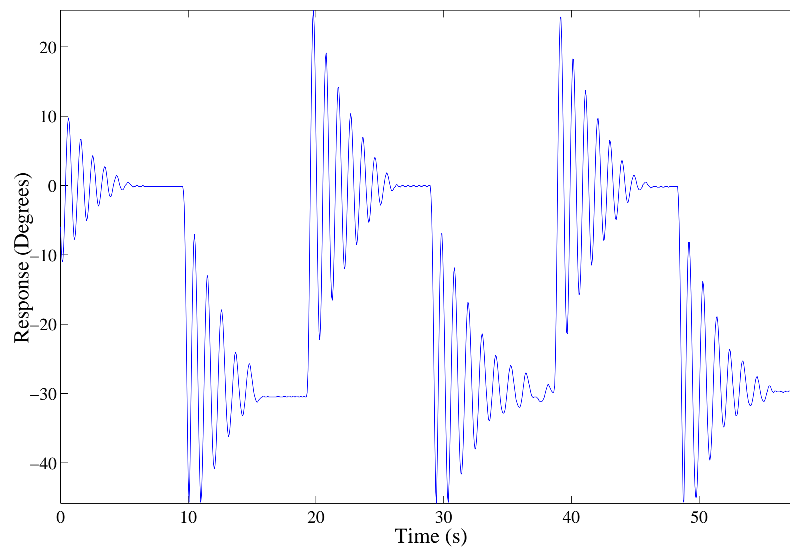


Figure 12: System Identification with $u = 200$

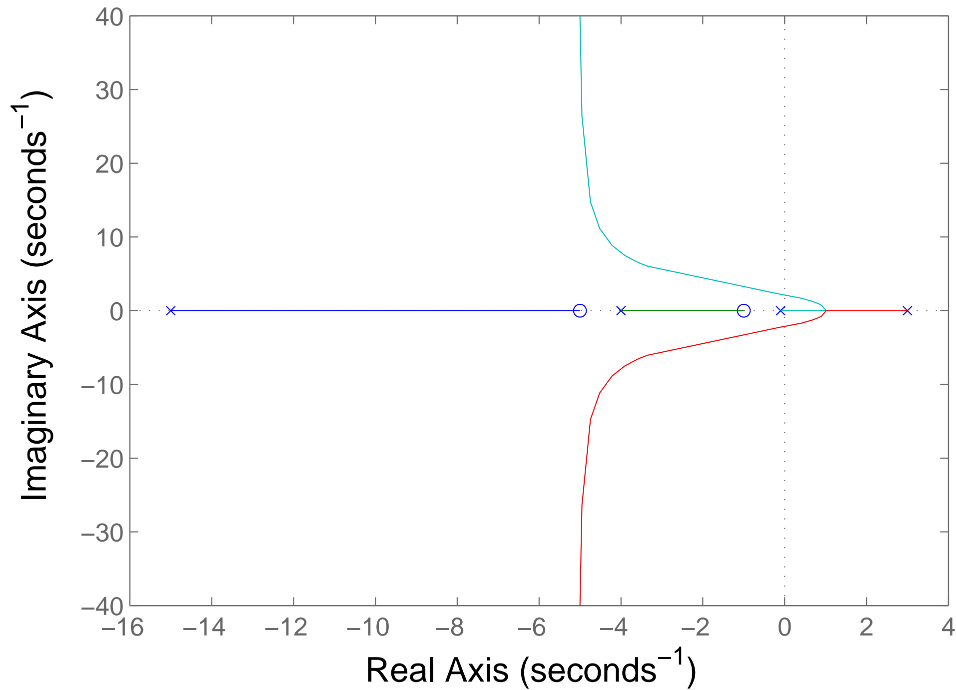


Figure 13: Basic Shape of Root Locus with Lead Lag Control

5.3 Appendix C: C2D

Here we present the computations for converting a controller from continuous to discrete time. The transform, using 'Tustin's method', can be obtained simply by substituting

$$s = \frac{2}{T}\frac{1-\frac{1}{z}}{1+\frac{1}{z}} \tag{25}$$

into $G(s)$ to obtain $G_{dis}(z)$.

We verify for a simple lead or lag compensator, $G_l(s) = \frac{s+a}{s+p}$. Inserting $s(z)$, we ostensibly get

$$G_{dis}(z) = \frac{\frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} + a}{\frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}} + p} \tag{26}$$

which after clearing away the inverse z's results in

$$\frac{\frac{2}{T}(z-1) + a(z+1)}{\frac{2}{T}(z-1) + p(z+1)} \tag{27}$$

or

$$\frac{(2/T + a)z + (a - 2/T)}{(2/T + p)z + (p - 2/T)} \tag{28}$$

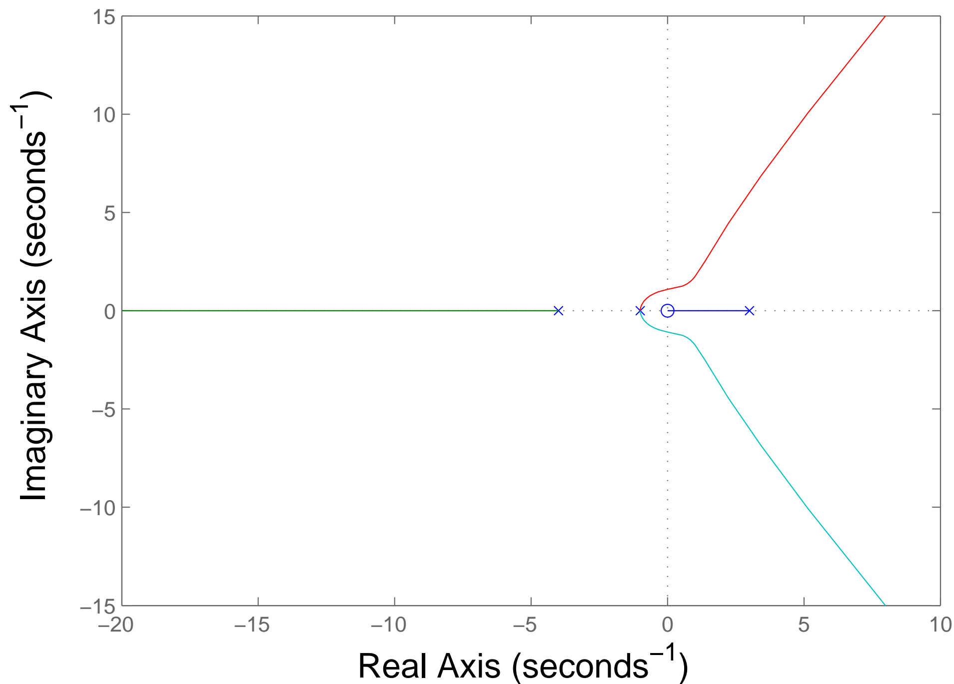


Figure 14: Basic Shape of Root Locus for PID Control for *Inverted* Pendulum

and finally normalizing the highest order coefficient in the denominator, the final form is

$$\frac{\frac{2/T+a}{2/T+p}z + \frac{a-2/T}{p+2/T}}{z + \frac{p-2/T}{p+2/T}}. \tag{29}$$

Indeed, for $T = 1/20$, $a = 3$, $p = 5$, $c2d(G(s))$ in Matlab produces

$$G_{dis}(z) = \frac{0.9556z - 0.8822}{z - 0.7778} \tag{30}$$

which agrees with our calculations.

5.4 Appendix D: PID

In this report, it would have been nice to include a different kind of controller. Of course, I could have included data of lead lag controllers with different parameters, but the controller we used gave us satisfactory performance and it seems like unnecessary detail to do anything more with this (after all, I am- *Deo volente*- graduating in two weeks and it's better not to add too much pressure before then).

But a PID controller was unable to be implemented. Here's why: look at Figure 14; the root locus plot shows that the system is unstable no matter what k is. Nevertheless, a PID controller would have been worked on the hanging pendulum, as Figure 15 indicates.

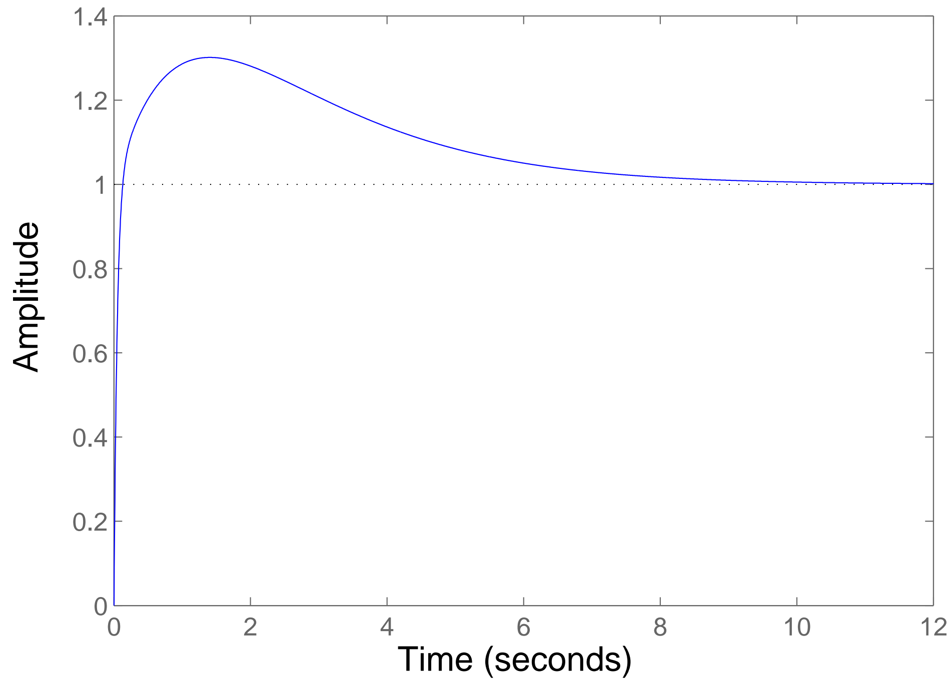


Figure 15: Theoretical Response with PID Control for *Hanging* Pendulum

5.5 Appendix E: Screenshots