Contents

1. Introduction

This is a brief how-to on using the code that has been written for the inverted pendulum. This code was originally designed to operate in many "modes." These fall into two basic categories:

1.1. Category 1: Inverted Stabilization. After the inverted pendulum was manufactured, it was necessary to determine if the plant was even able to invert the pendulum and maintain this state for an extended period. The easiest way at the time was to write a basic PID controller and tweak the gains to see if it is possible. After only a few hours of educated guesses, a controller that could keep the pendulum inverted with minimal steady state error was achieved. The code for this component is described in the following text.

1.2. Category 2: Excitation. Because of the need to capture dynamics of the system for system identification, code was developed to excite the system in many ways. This includes a step command, a ramp, a square wave, a sawtooth wave, and a sine wave. The specific parameters of these waves such as speed, acceleration, amplitude, and frequency can be modified in the code. Each of these excitation procedures is executed for one minute. While this is running, data about the position of both the pulley and the pendulum, as well as the input is written to a file on a PC.

Much of the support code should already be written to implement a full state feed back controller. This document will highlight the location and use of particular sections of code that may be useful in developing such a controller. This code is written, compiled, and loaded using the leaf labs Maple IDE. The microcontroller used is a Maple Leaf. This code is written in a language much like C but more specifications can be found at the Language Reference portion of LeafLabs website.

This file is located at: SysID/code/branches/inverted_pendulum/maple/pendulum_datalog.

Date: December 3, 2010.
2. **Physical System**

The system can be described as follows: The Maple Leaf $\mu$C is connected to an electronic speed controller (ESC) which uses power from a 13.8 volt, 25 amp DC power supply to drive a DC brushed motor. This motor drives a pulley which is linked to the large Al pulley via a toothed belt. This pulley has a 4096 count quadrature encoder mounted to the shaft. An arm on this pulley holds two, low friction, ball bearings. This holds a shaft from which the inverted pendulum swings freely in the plane parallel the rotational axis of the large pulley. One end of this shaft is connected to a 4000 count quadrature encoder. This rotational inverted pendulum is not an ideal system. Friction does exist. The belt does slip when commands of a large disparity are issued consecutively. But the designers attempted to minimize these effects and disregarding them would most likely yield a successful system design. A model of the inverted pendulum can be seen in Fig. ??.

![Graphical Model of Inverted Pendulum](image)

**Figure 1.** Graphical Model of Inverted Pendulum

If one wanted to obtain a mathematical model of this system, they may desire certain parameters. While these parameters may vary from model to model, some should be common across models. The author has attempted to determine some
of these parameters in advance for your convenience and to prevent dismantling of the pendulum by those who may be unfamiliar with its construction. These parameters are listed below in Table 1.

Table 1. Inverted Pendulum Model Parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m_1$</td>
<td>Mass of Pendulum</td>
<td>.1176</td>
<td>kg</td>
</tr>
<tr>
<td>$m_0$</td>
<td>Mass of Rotational Base</td>
<td>1.027</td>
<td>kg</td>
</tr>
<tr>
<td>$l_0$</td>
<td>Length of Actuator Arm</td>
<td>.1847</td>
<td>m</td>
</tr>
<tr>
<td>$l_1$</td>
<td>Length To Center of Inertia From Center of Rotation of Pendulum</td>
<td>.225</td>
<td>m</td>
</tr>
<tr>
<td>$J_{z0}$</td>
<td>Moment of Inertia of Actuator Arm about Rotational Axis</td>
<td>.01128</td>
<td>kg/m²</td>
</tr>
<tr>
<td>$J_{z1}$</td>
<td>Moment of Inertia of Pendulum Rod about Rotational Axis</td>
<td>.00075</td>
<td>kg/m²</td>
</tr>
<tr>
<td>$g$</td>
<td>Gravitational Acceleration</td>
<td>9.81</td>
<td>m/s²</td>
</tr>
</tbody>
</table>

3. Functions

For the purposes of developing and using a state space model, some functions will be useful. The functions and their descriptions are as follows:

3.1. main loop. This code is unique in that it does not have a "main" function but a "loop" function. The difference is that instead of ending once a program has run its course, as in a traditional C code, this program will, by default, repeat itself. This loop contains two important functions. The pidControl(pendPos) function calls the main control function. If one was to write another control function, it should be written to be placed here and return a velocity integer.

The code immediately following that function on lines 33 through 40 is a "Forcing Function." It is used to excite the system in a particular manner. This section of code, which is currently commented out should never be uncommented while the code of line 31 is also uncommented and vice-versa. The forcing functions are discussed in more detail in section ??

```c
void loop ()
{
    long pendPos = 0;
    long pulleyPos = 0;
    int i = 0;
    int vel = 0;
    Timer* decoder;
    decoder = (Timer*)TIMER4_BASE;
    char data[65];
    data[0] = '\0';
```
time = 0;
while(1) {
    if(run) {
        run = false;
        // timing pin
        (GPIOC_BASE)->BSRR = BIT(8);
        if(time > 60000000) { // shut down after 60 seconds
            drive(0);
            while(1) {} } }
pendPos = getPendDiffPos();
pulleyPos = getPulleyDiffPos();

    // forcingFunction (time, freq, amp, func);
    vel = forcingFunction (time, 1.0, 0.25, TRIANGLE);
    if(vel > MAX_VEL)
        vel = MAX_VEL;
    if(vel < -1 * MAX_VEL)
        vel = -1 * MAX_VEL;

    drive(vel);

    // log the data through the USB
    unsignedToAscii(data, (unsigned long) time);
    appendAsciiSigned(data, (long) pulleyPos);
    appendAsciiSigned(data, (long) pendPos);
    appendAsciiSigned(data, (long) vel);
    // appendAsciiSigned(data, (long) error);
    appendChar(data, '\n');
    SerialUSB.write(data);
    (GPIOC_BASE)->BSRR = BIT(8);
} }
3.2. `getPendAbsPos()` and `getPendDiffPos()`. These functions are similar in that they both return the position of the pendulum. The `getPendDiffPos()` returns the position of the pendulum at a given time and is the function that will most likely be used for stabilization. It returns an unsigned integer with values 0 to 4000 where 0 is strait down and 2000 is strait up.

The `getPendAbsPos()` function operates similarly except that after one complete revolution is made, the value is not reset to 0. Instead, it continues to increment. For example, if the pendulum starts at 0 (strait down) and makes two complete revolutions, the function will return 8000 (4000 counts/revolution * 2 revolutions). It can return very large values.

3.3. `getPulleyAbsPos()` and `getPulleyDiffPos()`. These functions are similar to those discussed above. The `getPulleyDiffPos()` returns an integer value from 0 to 4096 where 0 is the initial position of the pulley. Once a complete revolution is made, the value is reset to zero.

The `getPulleyAbsPos()` is similar to the function above. It too returns an integer value, but does ”wind up” and is not reset to 0 at each revolution which may make taking the derivative easier.

3.4. `drive(int vel)`. The drive function issues a drive command to the electronic speed controller in the form of a modulated pulse width (PWM). Once a value is written, that value is held and a constant PWM is issued until the function is called again. Because the motor is powerful, there are limits placed in the code that artificially saturate the signal to prevent the pulley from reaching high speeds. I would recommend that you do not change these limits. The system should be completely capable of issuing any stabilization command necessary to invert the pendulum, even with these limits in place.

```cpp
void drive(int vel)
{
    // Serial.println(vel, DEC);
    // PORTC |= 0x40;  //digitalWrite(ENA, HIGH);
    // PORTC != 0x20;  //digitalWrite(ENb, HIGH);
    if(vel > NEUTRAL)
    {
        // vel = vel;
        (GPIOB_BASE)->BSRR = BIT(11);  //digitalWrite(INa, HIGH);
        (GPIOB_BASE)->BRR = BIT(13);  //digitalWrite(INb, LOW);
    }
    else if(vel < NEUTRAL)
    {
        vel *= -1;
        (GPIOB_BASE)->BSRR = BIT(11);  //digitalWrite(INa, LOW);
        (GPIOB_BASE)->BRR = BIT(13);  //digitalWrite(INb, HIGH);
    }
    else
```
3.5. pidControl(int pos). The controller that was developed by trial and error for this system is contained in the pidControl(int pos) function. This function is passed the current position of the pendulum and issues a command to the drive function based on the position. In the code below, lines 15 and 16 are where the actual control equation exists. The rest is simply data manipulation. The while loop at line 5 ensures that the position is unwrapped to a positive value. The mod operator at line 7 unwraps the data that is above the counts per revolution. The two previous operations ensure that the data is unwrapped to between 0 and 4000.

The if statement on line 9 prevents integral windup. The if statement on line 13 sets a bound for the controller. If the pendulum were to move this far from the target, it would be unable to recover and move in a seemingly random fashion. To prevent this, the limits were added. The operation on line 27 is intended to prevent integral wind up as well.
vel = (P_GAIN * (TARGET - pos) - D_GAIN * (pos - lastPendPos) + I_GAIN * integral);

// vel = vel / 20;
}
else
    vel = NEUTRAL;

// SerialUSB.print('#');
// SerialUSB.println(pos);
// SerialUSB.println(integral);

lastPendPos = pos;
// decay the integral
integral = (integral * 49) / 50;

return vel;
}

4. Data Capture

The inverted pendulum is equipped with two high accuracy incremental quadrature encoders. Encoder one is mounted below the pulley and tracks the pulley position. Encoder two is mounted on the pulley arm and returns the position of the pendulum. Because these encoders are so accurate, they interrupt very often. For this reason, a microcontroller with a high clock speed such as the Maple Leaf is used. An Arduino will not able able to process the data fast enough to work well with these encoders without additional hardware.

Data is streamed back to the computer via USB. Typically with most microcontrollers, this is achieved using a serial.print. Because this type of communication is very slow. To better utilize the speed of the Maple Leaf, SerialUSB.write(char[] data) was used to speed up data transfer. The optimal data size to transfer at a given time is 64 bites.

To capture this data on a Unix based system, enter the following command into the terminal.

cat /dev/cu.usbmodem24361 > "fileName".txt

Of course the text in quotation marks is left to your discretion and the quotes are not necessary when naming the file. The 24361 is a number that may change when you disconnect and reconnect the Maple leaf. You can find this number by opening the MapleIDE application and clicking on the serial monitor application. At the top of the serial monitor window in the title box, the appropriate number will be displayed. The text file written is ASCII formatted and contains four values per sample, time, pulley position, pendulum position, and velocity.
5. Forcing Functions

The forcing functions were designed to allow the operator to persistently excite the system in various ways. Many types of forcing functions exist:

- Constant
- Ramp
- Square
- Sawtooth
- Triangle
- Sine
- Double Sine
- AM Sine

The forcing function is passed the current time (in $\mu$sec) as an integer, the frequency (in Hz) as a floating point number, the amplitude of the desired signal with values that range from 0 to 1, and the type of function used.

The *Constant* forcing function simply issues a constant PWM command to the ESC that is proportional to the amplitude. The amplitude is a floating point number that should range from 0 to 1.

The *Ramp* forcing function will start at zero and ramp up to the maximum velocity at a given frequency in Hz.

The *Square* forcing function will deliver a PWM that is modulated via square wave about 0 with an amplitude of 0 to 1 where 0 is stationary and 1 is maximum velocity at the input frequency.

The *Sawtooth* forcing function generates a sawtooth waveform at a given frequency and amplitude.

The *Triangle* forcing function generates a triangle waveform at a given frequency and amplitude.

The *Sine* forcing function outputs a sine wave at a given frequency and amplitude.

The *Double Sine* forcing function outputs a summation of two waveforms at potentially different frequencies. One frequency is set by the function, and the other must be set in the function by a multiplication of the given frequency. This is done in line 7 of the code below.

```c
    case DOUBLE_SINE:
        period = (int) (1000000.0 / freq);
        while (t < period / -2)
            t += period;
        while (t > period / 2)
            t -= period;
        force = 0.5 * amp * MAX_VEL * \sin(freq * 2.0 * PI * t / 1000000.0)
                + 0.5 * amp * MAX_VEL * \sin(freq * 2.0 * PI * PI * t / 1000000.0);
        break;
```
The AM Sign forcing function is designed to change the amplitude sinusoidally. The amplitude of the waveform is the multiplication of the two waveforms. The amplitude of each is set to be equal but can be adjusted in the code. The frequency of each contributing sine wave is different as well. One is set to the frequency given in the function and the other is set to that value multiplied by a constant. This allows the frequency to be changed in the function call and two output frequencies to stay proportional to each other. This can be done by editing line 7 of the case code below.

```c
  case AM_SINE:
    period = (int) (1000000.0 / freq);
    while(t < period / -2)
      t += period;
    while(t > period / 2)
      t -= period;
    force = amp * MAX_VEL * sine(freq * 20.0 * t / 1000000.0) * amp
      * MAX_VEL * sine(freq * 2.0 * PI * t / 1000000.0);
    break;
  default:
    break;
```

**APPENDIX A. MAPLE LEAF CODE TITLED PENDULUM_DATALOG**

```c
/*
 * Title: Pendulum_datalog
 * Authors: Craig Bidstrup
          Ralph Grehek
          Andrew Muxen
 * Date: November 22nd, 2010
 * Purpose: This code is intended to drive the inverted pendulum with
           either
           the predefined functions or a PID controller. It is to be
           loaded
           on the leaf lab maple (http://leaflabs.com/devices/maple/).
 */
#include "ascii.h"

#define TIMER5_BASE 0x4000C0

typedef struct {
  volatile uint16 CR1;
  uint16 RESERVED0;
  volatile uint16 CR2;
```
uint16 RESERVED1;
volatile uint16 SMCR;
uint16 RESERVED2;
volatile uint16 DIER;
uint16 RESERVED3;
volatile uint16 SR;
uint16 RESERVED4;
volatile uint16 EGR;
uint16 RESERVED5;
volatile uint16 CCMR1;
uint16 RESERVED6;
volatile uint16 CCMR2;
uint16 RESERVED7;
volatile uint16 CCER;
uint16 RESERVED8;
volatile uint16 CNT;
uint16 RESERVED9;
volatile uint16 PSC;
uint16 RESERVED10;
volatile uint16 ARR;
uint16 RESERVED11;
volatile uint16 RCR;
uint16 RESERVED12;
volatile uint16 CCR1;
uint16 RESERVED13;
volatile uint16 CCR2;
uint16 RESERVED14;
volatile uint16 CCR3;
uint16 RESERVED15;
volatile uint16 CCR4;
uint16 RESERVED16;
volatile uint16 BDTR; // Not used in general purpose timers
uint16 RESERVED17;
// Not used in general purpose timers
volatile uint16 DCR;
uint16 RESERVED18;
volatile uint16 DMAR;
uint16 RESERVED19;
}
//***************************************************************************
// H-bridge motor controller defines
//***************************************************************************
#define PWM_OUT 0
#define INa 30
#define INb 32
#define NEUTRAL 0
#define MAX_VEL 65536/2 //16384
//***************************************************************************
// quadrature encoder defines
#define INDEXx 1
#define CHAx 12
#define CHBx 11
#define INDEXy 2
#define CHAy 5
#define CHBy 9

#define CPR 4000
#define TARGET 2000
#define PULLEY_CPR 4096

// PID controller defines
#define P_GAIN 600
#define D_GAIN 3000
#define I_GAIN 10
#define DEAD_BAND 3000

#define CONSTANT 0
#define RAMP 1
#define SQUARE 2
#define SAWTOOTH 3
#define TRIANGLE 4
#define SINE 5
#define DOUBLE_SINE 6
#define AM_SINE 7

#define testPin 37  // used for timing, watch pin with O-scope, should never have 100% duty cycle!!!
#define HEARTBEAT_PERIOD 1000  // in microseconds

volatile boolean run = false;
volatile unsigned int time = 0;
// Setup up the Global variables used for the encoders
//**************************************************************
volatile short X_offset = 0;
volatile long longPulleyPos = 0;
volatile short Y_offset = 0;
volatile long longPendPos = 0;
volatile unsigned int error = 0;

//**************************************************************
// Setup up the Global variables used for the drive
//**************************************************************
volatile int integral = 0;
volatile int lastPendPos = 0;
volatile short dir = 1;  // used only in the forcing function
volatile float triangleCounter = 0;

// The setup() method runs once, when the sketch starts

void setup() {
    // initialize the digital pin as an output:
    pinMode(testPin, OUTPUT);
    pinMode(PWMOUT, PWM);
    Timer2.setPrescaleFactor(1);

    initPendDecoder();
    initPulleyDecoder();
    initHeartbeat();
    initMotor();

    pinMode(INDEXy, INPUT);
    attachInterrupt(INDEXy, encoderIndex, RISING);

    calibratePendulum();

    delay(1000);
}

// the loop() method runs over and over again,
// as long as the Arduino has power

void loop() {
    long pendPos = 0;
    long pulleyPos = 0;
    int i = 0;
    int vel = 0;
    Timer* decoder;
decoder = (Timer*)TIMER4_BASE;
char data[65];
data[0] = '\0';
time = 0;
while(1) {
    if(run) {
        run = false;
        // timing pin
        (GPIOC_BASE)\0BSRR = BIT(8):
        if(time > 60000000) { // shut down after 60 seconds
            drive(0);
            while(1) {};
        }
        pendPos = getPendDiffPos();
pulleyPos = getPulleyDiffPos();
    }
    /**************************************************
    /***************************************************/
    /**************************************************
    /*!!!!\0REPLACE THIS CONTROLLER CALL WITH YOUR CONTROLLER!!!!*/
    /***************************************************/
    /***************************************************/
    drive(pidControl(pendPos)); /* <= this one!!!! */
    /*
    // forcingFunction(time, freq, amp, func);
    vel = forcingFunction(time, 1.0, 0.25, TRIANGLE);
    if(vel > MAX_VEL)
        vel = MAX_VEL;
    if(vel < -1 * MAX_VEL)
        vel = -1 * MAX_VEL;
    drive(vel);
    */
    // log the data through the USB
    unsignedToAscii(data, (unsigned long) time);
    appendAsciiSigned(data, (long) pulleyPos);
    appendAsciiSigned(data, (long) pendPos);
    appendAsciiSigned(data, (long) vel);
    // appendAsciiSigned(data, (long) error);
    appendChar(data, '\n');
    SerialUSB.write(data);
    (GPIOC_BASE)\0BRR = BIT(8);
}
void heartbeat(void);

void heartbeat(void) {
    run = true;
    time += HEARTBEAT_PERIOD;
}

void encoderIndex() {
    if (digitalRead(INDEXy)) { // this is to "debounce" the interrupt so
        // we don’t get a false positive
        error = (((Timer*)TIMER4_BASE)->CNT); // store what the error was for
        // the last revolution
        if (((Timer*)TIMER4_BASE)->CNT > CPR / 2)
            error = CPR - error;
        // if (((Timer*)TIMER4_BASE)->CNT > 0)
        // longPendPos += CPR;
        // else
        longPendPos = CPR;
        ((Timer*)TIMER4_BASE)->CNT = 0; // reset the counter so we don’t
        // build up error
    }
}

void pendOverflow() {
    longPendPos += CPR;
    ((Timer*)TIMER4_BASE)->CNT = 0; // reset the counter so we don’t
    // t overflow the timer
}

void pendUnderflow() {
    longPendPos = CPR;
    ((Timer*)TIMER4_BASE)->CNT = 0; // reset the counter so we don’t
    // t overflow the timer
}

void pulleyOverflow() {
    longPulleyPos += PULLEY_CPR;
    ((Timer*)TIMER3_BASE)->CNT = 0; // reset the counter so we don’t
    // t overflow the timer
}

void pulleyUnderflow() {
    longPulleyPos = PULLEY_CPR;
}
((Timer*)TIMER3_BASE)->CNT = 0;  // reset the counter so we don’t overflow the timer
}

// ************************************************************
// Main Init Routine
// ***************************************************************
void initPendDecoder(void) {
  pinMode(5, INPUT);
  pinMode(9, INPUT);
  Timer* encoderTimer;
  encoderTimer = (Timer*)TIMER4_BASE;
  encoderTimer->CCMR1 = 0x1F1F;  // CC1S=01
  encoderTimer->CCMR2 = 0x0000;
  encoderTimer->CCER = 0x0011;
  encoderTimer->SMCR = 0x0003;  // SMS=011
  encoderTimer->CR1 = 0x0001;  // CEN=1
  encoderTimer->CR2 = 0x0000;
  encoderTimer->PSC = 0x0000;
  encoderTimer->ARR = CPR;

  Timer4.setCompare1(CPR - 1);
  Timer4.setCompare2(-1 * CPR + 1);
  Timer4.attachCompare1Interrupt(pendOverflow);
  Timer4.attachCompare2Interrupt(pendUnderflow);
  encoderTimer->CNT = 0x0000;  // reset the counter before we use it
}

void initPulleyDecoder(void) {
  pinMode(11, INPUT);
  pinMode(12, INPUT);
  Timer* encoderTimer;
  encoderTimer = (Timer*)TIMER3_BASE;
  encoderTimer->CCMR1 = 0x1F1F;  // CC1S=01
  encoderTimer->CCMR2 = 0x0000;
  encoderTimer->CCER = 0x0011;
  encoderTimer->SMCR = 0x0003;  // SMS=011
  encoderTimer->CR1 = 0x0001;  // CEN=1
  encoderTimer->CR2 = 0x0000;
  encoderTimer->PSC = 0x0000;
  encoderTimer->ARR = PULLEY_CPR;

  Timer3.setCompare1(PULLEY_CPR - 1);
  Timer3.setCompare2(-1 * PULLEY_CPR + 1);
  Timer3.attachCompare1Interrupt(pulleyOverflow);
  Timer3.attachCompare2Interrupt(pulleyUnderflow);
encoderTimer->CNT = 0x0000; // reset the counter before we use it

void initHeartbeat(void) {
    Timer1.setChannel1Mode(TIMEROUTPUTCOMPARE);
    Timer1.setPeriod(HEARTBEAT_PERIOD); // in microseconds
    Timer1.setCompare1(1); // overflow might be small
    Timer1.attachCompare1Interrupt(heartbeat);
}

// Returns the absolute
// position of the pendulum, where 0 is straight down
// and 2048 is straight up.
// The position is not reset once a complete revolution
// is made but "winds up"
unsigned short getPendAbsPos()
{
    unsigned short rawPos = 0;
    unsigned short pos = 0;
    rawPos = ((Timer*)TIMER4_BASE)->CNT;

    Serial.println(X_offset);
    pos = rawPos - Y_offset;
    while (pos > CPR)
    {
        pos -= CPR;
    }

    return pos; // mask it so we only get 12 bit precision:// ((rawPos - X_offset) % CPR);
}

// Returns the differential
// position of the pendulum, where 0 is straight down
// and 2048 is straight up.
// This function resets the position to zero once a
// complete revolution is made
long getPendDiffPos()
{
    short rawPos = 0;
    long pos = 0;
    rawPos = ((Timer*)TIMER4_BASE)->CNT;
// Serial.println(X_offset);  
pos = ((long) rawPos) - Y_offset + longPendPos;  
return pos; // mask it so we only get 12 bit precision // ((rawPos - X_offset) % CPR);

//******************************************************************************  
// Calls ReadEncoder(Y_ENCODER) and returns the absolute // position of the pendulum, where 0 is straight down // and 2048 is straight up. //******************************************************************************  
unsigned short getPulleyAbsPos()  
{  
  unsigned short rawPos = 0;  
  unsigned short pos = 0;  
  rawPos = ((Timer*)TIMER3_BASE)->CNT;  
  // Serial.println(X_offset);  
  pos = rawPos - X_offset;  
  // if (pos < 0)  
  //  pos += CPR;  
  return pos; // mask it so we only get 12 bit precision // ((rawPos - X_offset) % CPR);

//******************************************************************************  
// Returns the differential // position of the pulley, where 0 is straight down // and 2048 is straight up. //******************************************************************************  
long getPulleyDiffPos()  
{  
  short rawPos = 0;  
  long pos = 0;  
  rawPos = ((Timer*)TIMER3_BASE)->CNT;  
  // Serial.println(X_offset);  
  pos = ((long) rawPos) - X_offset + longPulleyPos;  
  // while (pos < 0)  
  //  pos += CPR;  
  return pos; // mask it so we only get 12 bit precision // ((rawPos - X_offset) % CPR);
// This section deals mostly with the drive system for the pendulum

// Init the motor driver

void initMotor()
{
    pinMode(INa, OUTPUT);
    pinMode(INb, OUTPUT);
    digitalWrite(INa, LOW);
    digitalWrite(INb, LOW);
}

// Calibrate the position of the encoder at first startup.
// This function will measure the original position of the
// pendulum and assume it is stationary in the down position.
// Then it will give a nudge and wait for the decoder chip
// to sense the index position from the encoder. At that point
// the decoder chip resets back to zero. The position of
// the encoder just before the reset is the difference
// between the new zero and the straight down position.
// From this we now know the absolute position of the pendulum.
//
// sets global int offset which is the distance between
// the zero position and the down position of the pendulum.

void calibratePendulum()
{
    short past_pos = 0;
    short current_pos = 0;
    int vel = MAX_VEL;
    int i = 0;
    int dir = 1;

    past_pos = current_pos = ((Timer4)TIMER4_BASE) >> CNT;

    drive(NEUTRAL + (vel * dir)); // nudge the motor to get the pendulum
    delay(200);
    brake();
    delay(200);
    dir = -1;
```plaintext
drive(NEUTRAL + (vel * dir)); //nudge the motor to get the pendulum swinging
delay(300);
brake();
delay(300);
drive(NEUTRAL);

while(i < 50)
{
  current_pos = ((Timer*)TIMER4_BASE)->CNT;
  // SerialUSB.println(past_pos);
  if(current_pos - past_pos < 2 & past_pos - current_pos < 2)
    i++;
  else
    i = 0;
  delay(100);
  past_pos = current_pos;
}

//inform the user the calibration is done
//drive(NEUTRAL + (vel * dir)); //nudge the motor to get the pendulum swinging
delay(30);
brake();
drive(NEUTRAL);
delay(2000);

//reset the positions, but not the pendulum counter, it should already be 0
longPendPos = 0;
longPulleyPos = 0;
((Timer*)TIMER3_BASE)->CNT = 0;
Y_offset = current_pos;

} //********************************************************************
// PID controller
//********************************************************************
int pidControl(int pos)
{
  int vel = 0;
  //remap the pendulum position
  while(pos < 0)
    pos += CPR;
  pos %= CPR;
```
```cpp
if (pos < TARGET + 10 && pos > TARGET - 10)
{
    integral += TARGET - pos;
}
if (pos < TARGET + 50 && pos > TARGET - 50)
{
    vel = (P_GAIN * (TARGET - pos) - D_GAIN * (pos - lastPendPos) +
           I_GAIN * integral);
    // vel = vel / 20;
}
else
    vel = NEUTRAL;

    // SerialUSB.print('#');
    // SerialUSB.println(pos);
    // SerialUSB.println(integral);

    lastPendPos = pos;
    // decay the integral
    integral = (integral * 49) / 50;
return vel;
}

/************
// drive the motor
************
void drive(int vel)
{
    // Serial.println(vel, DEC);
    // PORTC |= 0x40;    //digitalWrite(ENa, HIGH);
    // PORTC != 0x20;   //digitalWrite(ENb, HIGH);
    if (vel > NEUTRAL)
    {
        // vel = vel;
        (GPIOB_BASE)->BSRR = BIT(11);    //digitalWrite(INa, HIGH);
        (GPIOB_BASE)->BRR = BIT(13);     //digitalWrite(INb, LOW);
    }
    else if (vel < NEUTRAL)
    {
        vel *= -1;
        (GPIOB_BASE)->BRR = BIT(11);     //digitalWrite(INa, LOW);
        (GPIOB_BASE)->BSRR = BIT(13);    //digitalWrite(INb, HIGH);
    }
    else
    {
        vel = NEUTRAL;
        (GPIOB_BASE)->BRR = BIT(11);     //digitalWrite(INa, LOW);
    }
```
```c
542 (GPIOB_BASE)->BRR = BIT(13); //digitalWrite(INb, LOW);
544 // Serial.println(vel, DEC);
546 vel += DEAD_BAND; //this is an attempt to reduce our deadband in the
548 //do final safety check
550 if(vel < 0)
552 vel = 0;
554 else if(vel > MAX_VEL)
556 vel = MAX_VEL;
558 //drive the motor
560 // analogWrite(MOTOR_OUT, vel);
562 pwmWrite(PWM_OUT, vel);
564 
566 //******************************************************************************
568 // brake the motor
570 //******************************************************************************
572 void brake()
574 {
576 (GPIOB_BASE)->BSRR = BIT(11); //digitalWrite(INa, HIGH);
578 (GPIOB_BASE)->BSRR = BIT(13); //digitalWrite(INb, HIGH);
580 
582 //******************************************************************************
584 // generate a force to excite the pendulum system for data logging
586 // t: current time in us
588 // freq: in Hz
590 // amp: between 0.0 and 1.0
592 // func: which function to use
594 //******************************************************************************
596 int forcingFunction(int t, float freq, float amp, int func)
598 {
600 int force = 0;
602 int period;
604 int t_periodic;
606 switch(func)
608 {
610 case CONSTANT:
612 force = amp * MAX_VEL;
614 break;
616 //******************************************************************************
618 case RAMP:
620 force = ((freq * MAX_VEL) / 1000000.0) * t;
622 break;
624 //******************************************************************************
626 case SQUARE:
628 force = dir * amp * MAX_VEL;
630 
632 } // end forcingFunction()
634 
636 
638 
640 
642 
644 
646 
648 
650 
652 
654 
656 
658 
660 
662 
664 
666 
668 
670 
672 
674 
676 
678 
680 
682 
684 
686 
688 
690 
692 
694 
696 
698 
700 
702 
704 
706 
708 
710 
712 
714 
716 
718 
720 
722 
724 
726 
728 
730 
732 
734 
736 
738 
740 
742 
744 
746 
748 
750 
752 
754 
756 
758 
760 
762 
764 
766 
768 
770 
772 
774 
776 
778 
780 
782 
784 
786 
788 
790 
792 
794 
796 
798 
800 
802 
804 
806 
808 
810 
812 
814 
816 
818 
820 
822 
824 
826 
828 
830 
832 
834 
836 
838 
840 
842 
844 
846 
848 
850 
852 
854 
856 
858 
860 
862 
864 
866 
868 
870 
872 
874 
876 
878 
880 
882 
884 
886 
888 
890 
892 
894 
896 
898 
900 
902 
904 
906 
908 
910 
912 
914 
916 
918 
920 
922 
924 
926 
928 
930 
932 
934 
936 
938 
940 
942 
944 
946 
948 
950 
952 
954 
956 
958 
960 
962 
964 
966 
968 
970 
972 
974 
976 
978 
980 
982 
984 
986 
988 
990 
992 
994 
996 
998 ```
// if(!(t % ((int)(1000000.0 / freq))))) {
  if((t % ((int)(1000000.0 / freq))) < HEARTBEAT_PERIOD) {
    dir *= -1;
  }
  break;
} //*******************************************************************************/
case SAWTOOTH:
  t_periodic = t % ((int)(1000000.0 / freq));
  force = dir * freq * amp * MAX_VEL * (t_periodic / 1000000.0);
  break:
//*******************************************************************************/
case TRIANGLE:
  t_periodic = t % ((int)(1000000.0 / (freq / 2)));
  triangleCounter += 65 * dir * amp * freq;
  if(triangleCounter > amp * MAX_VEL || triangleCounter < -1 * amp * MAX_VEL) {
    dir *= -1;
  }
  force = (int)triangleCounter;
  break;
//*******************************************************************************/
case SINE:
  period = (int)(1000000.0 / freq);
  while(t < period / -2)
    t += period;
  while(t > period / 2)
    t -= period;
  force = amp * MAX_VEL * sin(freq * 2.0 * PI * t / 1000000.0);
  break:
//*******************************************************************************/
case DOUBLE_SINE:
  period = (int)(1000000.0 / freq);
  while(t < period / -2)
    t += period;
  while(t > period / 2)
    t -= period;
  force = 0.5 * amp * MAX_VEL * sin(freq * 2.0 * PI * t / 1000000.0) + 0.5 * amp * MAX_VEL * sin(freq * 2.0 * PI * PI * t / 1000000.0);
  break;
//*******************************************************************************/
case AM_SINE:
  period = (int)(1000000.0 / freq);
  while(t < period / -2)
    t += period;
  while(t > period / 2)
    t -= period;
force = amp * MAX_VEL * sine(freq * 20.0 * t / 1000000.0) * amp * MAX_VEL * sine(freq * 2.0 * PI * t / 1000000.0);
    break;
    default:
    break;
}
return force;

float sine(float x)
{
    while(x < -1 * PI)
        x += 2 * PI;
    while(x > PI)
        x -= 2 * PI;
    return x - x*x*x/6 + x*x*x*x*x/120 - x*x*x*x*x*x*x*x/5040;
}