My Project Documentation

Purpose

The purpose of this lab is to create a self balancing bike robot named the You're-Almost-There. Using a PID control scheme, the balancing bike was able to keep itself upright.

Usage

The appropriate pins will need to be set up to have the system running properly The pin descriptions are outlined in the classes in the other subsections. The I2C data bus must be enabled and be able to read the digital readings from the BNO055_Base driver The following code can be directly copied and pasted into the STM32 microcontroller, with motor shield, to run the self balancing mechanism. The battery must be fully charged, and if possible have a 24V DC source for more stability. The bike must initially be as straight as possible since the IMU takes the first measurement as the reference point of the balance.

Testing

You're-Almost-There initially could not balance itself with just a proportional controller. We added in the derivative element into the control scheme, but the system still could not reliably balance. After hours of testing, we were able to find out that the derivative gain had the wrong signage and gave the wrong signals to set the duty cycle of the motor. By correcting the signage, the bike was able to balance even with disturbances. To create a quicker responding system, we added in the integral element into the control scheme as well. To find the correct gains, we set all gain to 0 and adjusted the proportional gain to create a stable oscillation near the balance reference angle. After find the proportional gain, the derivative gain was adjusted to dampen out the oscillation as much as possible. After reaching close to a critically damped response, the integral gain was adjusted to reach steady state even quicker.

Bugs

Initially, the controller was running at such a high frequency and high priority, that it did not give the other tasks a chance to be run. We wanted to make sure the timing of the controller was at least 5~10 times faster than the time constant of the motor with the load. We fine tuned the timing frequency and the priority so that all the tasks have a chance to run. We set the ultrasonic sensor task as the highest priority for safety of the user and the bike. The ultrasonic sensor also had very buggy readings where if the readings were taken too quickly, it would pick up noise and return false values. We decided to run the ultrasonic sensor every 0.5 seconds so that the readings can be more stable. The IMU was also very buggy where the tuple returned from the function would not be stored into a variable. A 0 would be stored everytime, so we were able to mitigate that by not using a intermediate variable in the reading of the IMU. The remote sends code to the receiver in DEC protocol Due to the restrained time, the DEC protocol was not decoded so the input pin is triggered when the first 0 is read from the pin. The remote task can be unreliable due to the negligance of the DEC protocol.

Limitations

We purposefully did not use a thicker based wheel or counter balancing weight to stabilize our system. We wanted to fully balance the robot by using reaction wheel dynamics. The robot had trouble balancing due to the lack of inertia from the reaction wheel and the torque from the motor. On 24V DC power the robot is stable and can achieve equilibrium, but the voltage rating of the motor is only 12V DC. It is possible to run the motor with 24V DC power source, but the motor heats up due to high current. In the future, a higher rate torque motor can be used to mitigate this problem. The struture of the bike also creates vibration when the reaction wheel reaches around 600 RPM. The vibration does not affect the balance immensely, but if the accelerometer readings were very crucial in future applications, the IMU would return inaccurate values.For our purposes, the system was robust enough to handle minor vibrations. The driving motor, during our rigorous tests had mechanical failure. The gear box within the DC motor experiences slippage and there are times where the gears get jammed. In that case, the motor does not spin. The user needs to firmly tap the backside of motor housing to realign the motors in the gear box. If the motor experiences extensive stress and saturates the duty cycle fluctuating between -100 and 100 constantly, there is a chance for the flange coupler to give out. The coupler is set by 4 set screws and can get loose due to the constant impulses of torque.

Location

The code is located at the Mercurial server http://wind.calpoly.edu/hg/mecha11

NOTES

The controller was hard coded into the task in main. See attached code or mecurial repository for full description of the controller algorithm.

EncoderDriver.Encoder Class Reference

Public Member Functions

def **__init__** (self, input_pin1, input_pin2, input_timer)
Set up a timer/counter in quadrature decoding mode to read an. More...

def **read** (self)
Updates and returns the encoder position using the number in the. More...

def **zero** (self)
Set the encoder counter to zero. More...

Public Attributes

timer
Timer sets up the proper time channel based on the user input for the encoder reading.

ch1
Creates a timer channel in encoder counting mode in timer ch 1.

ch2
Creates a timer channel in encoder counting mode in timer ch 2.

position
Stores the total displaced position of the motor.

theta_old
Stores the subsequent displaced position of the motor.

Constructor & Destructor Documentation

◆ __init__()

```
def EncoderDriver.Encoder.__init__ ( self,
                                     input_pin1,
                                     input_pin2,
                                     input_timer
                                   )
```

Set up a timer/counter in quadrature decoding mode to read an.

optical encoder. User must specify which pin and timers the

which the encoder is connected to. To declare which pins you are

using, use the following code (NUCLEO-L476RG):

For pin C6 and C7 and timer 8

```
tim8 = pyb.Timer(8, prescaler=1, period=65535)

pinC6 = pyb.Pin (pyb.Pin.board.PC6, pyb.Pin.IN)

pinC7 = pyb.Pin (pyb.Pin.board.PC7, pyb.Pin.IN)
```

For pin B6 and B7 and timer 4

```
pinB6 = pyb.Pin (pyb.Pin.board.PB6, pyb.Pin.IN)

pinB7 = pyb.Pin (pyb.Pin.board.PB7, pyb.Pin.IN)

tim4 = pyb.Timer(4, prescaler=1, period=65535)
```

Parameters

input_pin1	pin that connects first input from the encoder to the microcontroller.
input_pin2	pin that connects second input from the encoder to the microcontroller.
input_timer	sets up timer for timer channels

Member Function Documentation

◆ read()

def EncoderDriver.Encoder.read (self)

Updates and returns the encoder position using the number in the.

encoder counter. Do this by taking the difference of the new position

from the old position. The read function takes into account the over

flows and underflow of the encoder.

◆ zero()

def EncoderDriver.Encoder.zero (self)

Set the encoder counter to zero.

Reinitializes shared variables

```
    position, theta_old and timer.counter() to zero
```

The documentation for this class was generated from the following file:

- EncoderDriver.py

motordriver.MotorDriver Class Reference

This class implements a motor driver for the ME405 board. More...

Public Member Functions

def **__init__** (self, pin1, pin2, pin3, timer)
Creates a motor driver by initializing GPIO pins and turning the motor off for safety. More...

def **set_duty_cycle** (self, level)
This method sets the duty cycle to be sent to the motor to the given level. More...

Public Attributes

p1
defines the pin of the PWM that the user inputs for the instance of the object

p2
defines the output pin that powers the motor to spin in one direction

p3
defines the output pin that powers the motor to spin in one direction

tim
defines the timer that the PWM will run on

Detailed Description

This class implements a motor driver for the ME405 board.

NOTE: reversing the signal of p2 and p3 will spin the motor in the reverse direction

Constructor & Destructor Documentation

◆ __init__()

```
def motordriver.MotorDriver.__init__ ( self,
                                       pin1,
                                       pin2,
                                       pin3,
                                       timer
                                     )
```

Creates a motor driver by initializing GPIO pins and turning the motor off for safety.

example(PA10,PB4,PB5,3)

Member Function Documentation

◆ set_duty_cycle()

```
def motordriver.MotorDriver.set_duty_cycle ( self,
                                             level
                                           )
```

This method sets the duty cycle to be sent to the motor to the given level.

Positive values cause torque in one direction, negative values in the opposite direction.

Parameters

level	A signed integer holding the duty cycle of the voltage sent to the motor

The documentation for this class was generated from the following file:

- motordriver.py