DE2 Electronics 2
Additional notes on PID Control Class

I realised from yesterday Lab Session that many of you are struggling with PID controller. The problem lies with the following:

1. It is generally not easy to tune a PID control of such an unstable system in the first place.
2. It is even harder because you may have mistakes in your Python code. It is not easy to know if the error is in the tuning or the code.
3. The derivative term is really bad for two reasons. If the measurement (e.g. motor speed) is noisy, then the derivative of is very noisy, particularly after divided by dt (which is small).

Theo Bui also pointed out to me the inconsistency between my lectures slides and the instructions for Lab 7 and the Challenges. On the slides, I suggested the derivative term can come from the gyroscope (which provide a relatively clean **ptich_dot** measurement). In my Lab instruction, I suggested that you calculate **error_dot** by **(current_error – last_error)/dt**. I was wrong!

I spent last evening thinking hard about this problem and decided to write a PID Class for you to use. Note that I have NOT tested this, but I hope this works. There are two versions: version 1 is designed for Challenge 5 self-balancing, where **pitch** and **pitch_dot** are available.

Version 2 is designed for controlling the speed of the motor. You can download these two versions from course webpage.

Version 1 – Self-balancing control

```python
'''
-------------------------------------------------------
Name: PID Controller 1
Creator: Peter Cheung
Date: 16 March 2020
Revision: 1.0
-------------------------------------------------------
PID Controller version 1
This version uses pitch_dot directly from Gyro.
This avoids noisy derivative term and specific to
self-balancing.
Useful for self-balancing challenge
-------------------------------------------------------
'''
import pyb

class PIDC:
    def __init__(self, Kp, Kd, Ki):
        self.Kp = Kp
        self.Kd = Kd
        self.Ki = Ki
        self.error_last = 0       # These are global variables to remember various states of contr
        self.tic = pyb.millis()
        self.error_sum = 0

    def getPWM(self, target, pitch, pitch_dot):

        # error input
        error = target - pitch          # e[n]

        # derivative input
        derivative = -pitch_dot         # negative feedback

        toc = pyb.millis()
        dt = (toc-self.tic)*0.001       # find dt as close to when used as possible
        # Integration input
        self.error_sum += error*dt

        #   Output
        PID_output = (self.Kp * error) + (self.Ki * self.error_sum) + (self.Ki * derivative)

        # Store previous values
        self.error_last = error
        self.tic = toc

        pwm_out = min(abs(PID_output), 100)             # Make sure pwm is less than 100

        if PID_output > 0:                              # Output direction (need to check)
            direction = 'forward'
        elif PID_output < 0:
            direction = 'back'
        else:
            direction = 'stop'

        return pwm_out, direction
```

Usage:

pid = PIDC(4.0, 0.5, 1.0)   # create pid object with these Kp,Kd,Ki values
pwm, direction = pid.getPWM(0.0, pitch, pitch_dot)# work out PWM drive value

Explanations
Control variable is pitch angle, and the output is PWM duty cycle and direction.

Lines 17 to 24:	initialization code. Make these variables visible outside the class. So you can check last error value with pid.error_last.
Line 29:	calculate current error
Line 32:	instead of calculating error dot, use pitch_dot from gyroscope. Allowed here because pitch_dot is a good approximation to error_dot in this case and it is NOT noisy.
Line 35:	calculate dt in seconds to use later for integral term
Line 37:	integral term
Line 40:	do the PID control computation
Lines 43-44:	Update "states" of the PID controller by storing previous values
Line 46:	Limit PWM duty cycle to 0 to 100
Lines 48-53:	Return direction of motor to achieve balance

Version 2

Here is where version 2 differs from version 1:

```
24     def getPWM(self, target, speed):
25
26         # error input
27         error = target - speed                    # e[n]
28
29         # derivative input
30         derivative = error - self.error_last      # error_dot. assume dt is constant
31                                                   # 1/dt is absorbed into Kd
32                                                   # this avoid division by small value
```

Line 24:	No speed_dot because it is not available.
Line 30:	calculate error_dot as (e[n]-e[n-1]). Assume that 1/dt is constant and absorbed into Kd. In fact, I think you can make Kd=0.0 for motor speed control. It is not critical here.

Also we only control the speed of the motor, not direction for this case. So, not need to check for direction.

For Challenge 3, controlling the speed of motor, I think a simple Proportional or Proportional-Integral controller would suffice.