Yoseph Ayele Kebede

UID: 114196729

Dr. Mitchell

ENPM701 Assignment 8

DC Motor Control using Encoder Count and IMU

Assignment #8

DC Motor Control via Encoder Count and IMU

Question #1

Motor Control Algorithm via Encoder and IMU

1.2 Rectangular Loop

Video w/ Encoder alone:
https://youtu.be/TuNZdERHT2A?si=z2oagG120bK2T8Us

Video w/ Encoder and IMU:
https://youtu.be/seDikZTLaBA?si=ijqq2HkFpOOfZaZL

1.3 The Algorithm

```
import serial
import RPi.GPIO as gpio
import time
import numpy as np
import math
import matplotlib.pyplot as plt
import pprint

#### Initialize GPIO pins ####

def init():
    gpio.setmode(gpio.BOARD)
    gpio.setup(31, gpio.OUT) # IN1
    gpio.setup(33, gpio.OUT) # IN2
    gpio.setup(35, gpio.OUT) # IN3
    gpio.setup(37, gpio.OUT) # IN4

    gpio.setup(7, gpio.IN, pull_up_down = gpio.PUD_UP)
    gpio.setup(12, gpio.IN, pull_up_down = gpio.PUD_UP)
```

```
def pwmInit(pwms):

    pwms[0].ChangeDutyCycle(0)
    pwms[1].ChangeDutyCycle(0)
    pwms[2].ChangeDutyCycle(0)
    pwms[3].ChangeDutyCycle(0)

    return pwms

def gameover(pwms):
    pwms[0].ChangeDutyCycle(0)
    pwms[1].ChangeDutyCycle(0)
    pwms[2].ChangeDutyCycle(0)
    pwms[3].ChangeDutyCycle(0)

def forward(pwms,vals):
    #init()
    # Left wheels
    #gpio.output(31, True)
    pwms[0].ChangeDutyCycle(vals[0])
    #gpio.output(33, False)
    pwms[1].ChangeDutyCycle(vals[1])

    # Right wheels
    #gpio.output(35, False)
    pwms[2].ChangeDutyCycle(vals[2])
    #gpio.output(37, True)
    pwms[3].ChangeDutyCycle(vals[3])


def reverse(tf,pwms,vals):
    #init()
    # Left wheels
#     gpio.output(31, False)
    pwms[0].ChangeDutyCycle(vals[0])
#     gpio.output(33, True)
    pwms[1].ChangeDutyCycle(vals[1])
    # Right wheels
#     gpio.output(35, True)
    pwms[2].ChangeDutyCycle(vals[2])
#     gpio.output(37, False)
    pwms[3].ChangeDutyCycle(vals[3])
```

```python
def pivotleft(tf):
    init()
    # Left wheels
#     gpio.output(31, False)
    pwms[0].ChangeDutyCycle(vals[0])
#     gpio.output(33, True)
    pwms[1].ChangeDutyCycle(vals[1])
    # Right wheels
#     gpio.output(35, False)
    pwms[2].ChangeDutyCycle(vals[2])
#     gpio.output(37, True)
    pwms[3].ChangeDutyCycle(vals[3])


def pivotright(tf):
    init()
    # Left wheels
#     gpio.output(31, True)
    pwms[0].ChangeDutyCycle(vals[0])
#     gpio.output(33, False)
    pwms[1].ChangeDutyCycle(vals[1])
    # Right wheels
#     gpio.output(35, True)
    pwms[2].ChangeDutyCycle(vals[2])
#     gpio.output(37, False)
    pwms[3].ChangeDutyCycle(vals[3])


# Distance to encoding conversion
# x_meters * (1 rev / (2pi*0.0325m)) = # wheel rev = 960 counter
def meter2encoder(x_dist):
    encod = round(float(x_dist / (2*math.pi*0.0325))*960)

    return encod

def encoder2meter(encd):
    dist = round(float((encd / 960) * 2*math.pi*0.0325),1)

    return dist

def rot2encoder(deg):

    # Approximate radius of rotation computed from Baron
    radius = 0.111#0.146 # meters
    # Angle needed for robot to rotate
```

```
    arc = ((deg * math.pi) / 180) * radius
    distance = round(float(arc / (2*math.pi*0.0325))*960)

    return distance

def encoder2deg(encd):

    # Approximate radius of rotation computed from Baron
    radius = 0.111

    arc = (encd / 960) * 2*math.pi*0.0325
    deg = round((arc / radius) * (180 / math.pi),1)

    return deg


def userInput(duty):
    print("*" * 70,'\n')
    print("Welcome to the Grand Challenge \n")
    print("*" * 70,'\n')

    print("Duty cycle for each motor in cardinal directions obtained from
multiple trial run experimentation \n")
    pprint.pprint(duty)
    print("*" * 70,'\n')

    print("Provide sequence of commands the robot should move in as
[direction_value direction_value direction_value]")
    print("Example: f_2 l_90 f_2")

    print("This would drive the robot 2 meters forward, rotate 90 left(ccw) and 2
meters forward again")
    print("+" * 70, '\n')

    sequence = list(input("Provide drive sequence for robot: ").split())
    sequence = [tuple(element.split("_")) for element in sequence]
    print("Your chosen sequence is: \n")
    print(sequence)

    return sequence

def encoderControl(direction, error_encoder, duty):

    # Initialize left and right motor duty cycles
    valL = 0
```

```
    valR = 0
    thresh = 0

    if direction in ['f','rev']:
        thresh = 20
    else:
        thresh = 5

    if error_encoder > thresh: # when left motor advances more than the right

        # Give power to corresponding motors
        valL = duty[direction]['motion']['lMotor'][0]
        valR = duty[direction]['motion']['lMotor'][1]


    elif error_encoder < -thresh: # when right motor advances more than the left

        # Give power to corresponding motors
        valL = duty[direction]['motion']['rMotor'][0]
        valR = duty[direction]['motion']['rMotor'][1]

    else:

        # Give power to corresponding motors
        valL = duty[direction]['start'][0]
        valR = duty[direction]['start'][1]

    return valL, valR

def drive2goal(direction, error_encoder, duty, pwms):

    # Convert yaw angles to encoder and add to encoder counts from
    # both motors

    valL,valR = encoderControl(direction, error_encoder, duty)

    # Based on the direction drive the motors accordingly
    if direction == 'f':

        # Drive forward with the above duty cycles
        pwms[0].ChangeDutyCycle(valL)
        pwms[3].ChangeDutyCycle(valR)

       # forward((pwm1,pwm2),vals)
```

```
    elif direction == 'rev':

        # Drive in reverse with the above duty cycles
        pwms[1].ChangeDutyCycle(valL)
        pwms[2].ChangeDutyCycle(valR)

        #reverse((pwm1,pwm2),vals)

    elif direction == 'l':

        # Pivot left with the above duty cycles
        pwms[1].ChangeDutyCycle(valL)
        pwms[3].ChangeDutyCycle(valR)

        #pivotleft((pwm1,pwm2),vals)

    else:

        # Pivot right with the above duty cycles
        pwms[0].ChangeDutyCycle(valL)
        pwms[2].ChangeDutyCycle(valR)

        #pivotright((pwm1,pwm2),vals)

    return pwms

def imu_serial(ser):

    count = 0

    while True:
        # Read for imu from serial
        if(ser.in_waiting > 0):

            count +=1

            # Read serial stream
            line = ser.readline()

            # Avoid first n-lines of serial information
            if count > 10:

                # Strip serial stream of extra characters

                line = line.rstrip().lstrip()
```

```
                line = str(line)
                line = line.strip("'")
                line = line.strip("b'")

                # Return float
                line = float(line)
                break

    return line

def imuQuick(ser):

    # Read serial stream
    line = ser.readline()

    # Strip serial stream of extra characters

    line = line.rstrip().lstrip()

    line = str(line)
    line = line.strip("'")
    line = line.strip("b'")

    # Return float
    line = float(line)

    return line

def dispPlot():

    states = np.genfromtxt("FLBR_in_motion_encoder_states.txt", dtype=str)

    size = states.shape

    print(size,'\n')

    x = np.int64(states[:,0].astype(np.float64))
    y = np.int64(states[:,1].astype(np.float64))

    # fig plot
    fig, ax = plt.subplots(1,1)

    # title
    fig.suptitle('Robot Path')
```

```python
    ax.plot(x,y,ls='solid', color='blue',linewidth=2, label='Robo-path')

    ax.set(title="Robot Trajectory",
           ylabel="Yi",
           xlabel="Xi")

    plt.savefig('Robot-path-encoder-imu.png')
    plt.show()
    plt.close()

def main():
    #### Main Code ####
    init()

    # Identify serial connection
    ser = serial.Serial('/dev/ttyUSB0', 9600)

    # Initialize FL and BR button count
    counterBR = np.uint64(0)
    counterFL = np.uint64(0)
    buttonBR = int(0)
    buttonFL = int(0)

    # Initialize pwm and duty cycle

    # initialize pwm signal to control motor
    pwm01 = gpio.PWM(31, 50)  # BackLeft motor
    pwm11 = gpio.PWM(33, 50) # FrontLeft motor
    pwm22 = gpio.PWM(35, 50) # FrontRight motor
    pwm02 = gpio.PWM(37, 50)  # BackRight motor

    pwms = [pwm01,pwm11,pwm22,pwm02]

    # Experimentally found duty cycle values for left and right motor
    # in movements of the four cardinal directions, converted to  dictionary
    dutyset = [('f', dict([('start',(35,40)),
                           ('motion',dict([('lMotor',(40,50)),
                                           ('rMotor',(50,45))])
                        )]
                    )),
               ('rev', dict([('start',(35,40)),
                           ('motion',dict([('lMotor',(22,30)),
                                           ('rMotor',(45,35))])
                        )]
```

```
                 )),
            ('l', dict([('start',(80,80)),
                        ('motion',dict([('lMotor',(80,84)),
                                        ('rMotor',(90,80))])
                         )])
             ),
            ('r',dict([('start',(90,94)),
                       ('motion',dict([('lMotor',(80,84)), #90,94
                                       ('rMotor',(90,80))])) # 99,90
                       ])
             )]

duty = dict(dutyset)

# Initialize pwms with 0 duty cycle so that we can pass voltage signals
# later
#map(lambda x: x.start(0), pwms)
pwms[0].start(0)
pwms[1].start(0)
pwms[2].start(0)
pwms[3].start(0)

# Open .txt file to save data
f = open('FLBR_in_motion_encoder_states.txt','a')
outstringF = ''

f_1 = open('IMU_and_encoder_states.txt','a')
outstring = ''
# Initialize variables to record encoder count
encoder_dist = 0

# Initialize variables to store commanded direction and value
direction = "placeholder"
drive = 0

# PID constants
Kp = 0.1
Ki = 0.1
Kd = 0.1

# Initialize variable to store the error between currrent location
# and destination
error = 0

# Initialize variable to save difference between left and
```

```
# right motor encoder count
error_encoder = 0

# Coordinates for horizontal (x) and vertical(y)
Xr = [0]
Yr = [0]

pose = (0,0)

# Initialize yaw in degrees from imu
yaw = 0

# Initialize a boolean to keep track of robot completing task at hand
completed = False

print("Initializing IMU, and clipping first few data reads ... \n")

# Initialize minimum pose in encoder count robot moved
pos_encoder = 0

# Total distance
s = 0
# Angle
ang = []
angLast = 0
delta = 0
deltaX = 0
deltaY = 0
# Save cmmd anchor
start = [0,0]

try:
    # Take user input commands
    sequence = userInput(duty)

    cnt = 0

    yaw1 = imu_serial(ser)

    xT = 0.0
    yT = 0.0

    while len(sequence) > 0:
        # Initialize boolean from start or after completion of action
        completed = False
```

```
# Current yaw at the start of action
yaw1 = imuQuick(ser)

# Pop out the first command from the sequence
cmd = sequence.pop(0)
print("Current task -> ", cmd)

if yaw1 <= 360 and yaw1 > 180:
        yaw1 = -(360 - yaw1)

# Assign direction and value from ordered pair
direction = cmd[0]
drive = float(cmd[1])
goal = 0

xT = pose[0]
yT = pose[1]

# Convert drive (linear/angular) to encoders
if direction in ['f','rev']:
    # convert linear distance to encoder count
    encoder_dist = meter2encoder(drive)
    print(f"Driving {direction}")
    ang.append(0)
    goal = encoder_dist


else:

    # Convert angle to encoder
    encoder_dist = rot2encoder(drive)
    print(f"Pivoting {direction}")
    if direction == 'l':
        goal = encoder_dist
    elif direction == 'r':
        goal = -encoder_dist

    ang.append(goal)

    print("Turn to this angle: ", drive)

# Check Angle difference
angle_diff = 0.0
```

```
            cnt = 0

            tick = 0
            s = 0
            # Check if robot completed prior task
            while not completed:
                cnt += 1

                # Compute the difference between the left and right encoders
                error_encoder = counterFL - counterBR

                # Command the robot to drive to user commanded postion
                # in sequence
                pwms = drive2goal(direction, error_encoder, duty,
pwms)

                # Count encoders for left and right motors
                if int(gpio.input(12)) != int(buttonBR):
                    buttonBR = int(gpio.input(12))
                    counterBR += 1

                if int(gpio.input(7)) != int(buttonFL):
                    buttonFL  = int(gpio.input(7))
                    counterFL += 1

                # Compute the difference in distance between start state  and
goal state
                # by considering the minimum of the two encoders
                pos_encoder = min(counterFL,counterBR)

                # Global error between start and goal state will be
                error = encoder_dist - pos_encoder

                # Save robot path
                if (cnt % 50) == 0:

                    if direction in ['f','rev']:
                        s = encoder2meter(pos_encoder)

                        delta = ang[0] - angLast

                        if tick == 0:

                            xT = s*math.cos(math.radians(delta))
                            yT = s*math.sin(math.radians(delta))
```

```
                        #Record encoder states to txt file
                        outstringF = str(xT) + ' ' + str(yT) + '\n'
                        print(outstringF)
                        f.write(outstringF)

                    if tick == 1:

                        xT = 0
                        yT = s
                        #Record encoder states to txt file
                        outstringF = str(Xr[-1]) + ' ' + str(Yr[-1]+yT) +
'\n'

                        print(outstringF)
                        f.write(outstringF)

                    if tick == 2:

                        xT = -s
                        yT = 0

                        #Record encoder states to txt file
                        outstringF = str(Xr[-1]+xT) + ' ' + str(Yr[-1]) +
'\n'

                        print(outstringF)
                        f.write(outstringF)

                    if tick == 3:

                        xT = 0
                        yT = -s

                        #Record encoder states to txt file
                        outstringF = str(Xr[-1]+xT) + ' ' + str(Yr[-1]+yT) +
'\n'

                        print(outstringF)
                        f.write(outstringF)

                    if tick == 4:

                        xT = s
                        yT = 0

                        #Record encoder states to txt file
```

```
                        outstringF = str(Xr[-1]+xT) + ' ' + str(Yr[-1]+yT) +
'\n'
                        print(outstringF)
                        f.write(outstringF)

                else:

                    #Record encoder states to txt file
                    outstringF = str(Xr[-1]) + ' ' + str(Yr[-1]) + '\n'
                    print(outstringF)
                    f.write(outstringF)

            if (error >= 0 and error <= 10) and (direction in ['f','rev']):

                print("counterBR: ", counterBR, "counterFL: ", counterFL)
                angle_diff = encoder2deg(pos_encoder) - yaw1
                print("Angle rotated: ", angle_diff)
                print("Expected turn: ", encoder_dist)
                print(angle_diff)

                Xr.append(xT)
                Yr.append(yT)

                pose = (Xr[-2]+Xr[-1],Yr[-2]+Yr[-1])

                print("current ordered pair: ", pose)

                completed = True

                counterBR = 0
                counterFL = 0

                angLast = ang[-1]
                pwms = pwmInit(pwms)


                for pwm in pwms:
                    pwm.start(0)

                time.sleep(2)
                print(f"Drive {direction} action completed! \n")

                tick += 2

            # Check if task is completed
```

```
            if (error >= 0 and error <= 4) and (direction in ['r','l']):

                print("counterBR: ", counterBR, "counterFL: ", counterFL)
                #print(angle_diff)
                angle_diff = encoder2deg(pos_encoder) - yaw1
                print("Angle rotated: ", angle_diff)
                print("Expected turn: ", encoder_dist)
                completed = True

                counterBR = 0
                counterFL = 0
                s = 0
                angLast = ang[-1]
                pwms = pwmInit(pwms)


                for pwm in pwms:
                    pwm.start(0)

                pose = (Xr[-1],Yr[-1])

                time.sleep(2)
                print(f"Drive {direction} action completed! \n")

                tick -= 1
        else:
            print("Destination Reached")
            gameover(pwms)

            for pwm in pwms:
                pwm.stop()

            gpio.cleanup()
            f.close()

            # Create a 2D NumPy array
            arr = np.hstack((np.array(Xr).T, np.array(Yr).T))

            f_1.write(str(arr))
            f_1.close()


    except KeyboardInterrupt:
        print("Keyboard Interrupted")
        gameover(pwms)
```

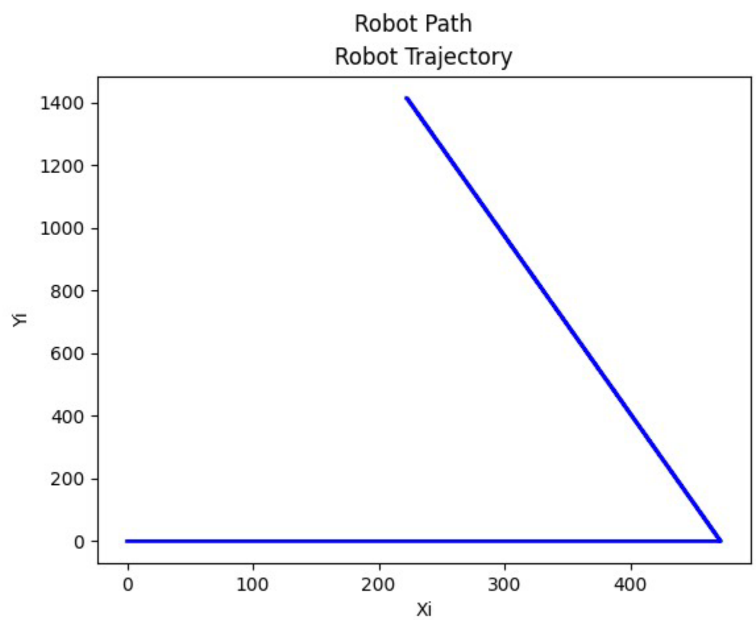
```python
        for pwm in pwms:
            pwm.stop()
        gpio.cleanup()
        f.close()
        print("Tasks Interrupted!")

        # Create a 2D NumPy array
        arr = np.hstack((np.array(Xr).T, np.array(Yr).T))

        f_1.write(str(arr))
        f_1.close()

if __name__ == "__main__":

    # Begin Program
    print("*" * 30, "PROGRAM STARTERD","*"*30, "\n")
    main()

    # Plot result
    dispPlot()
```

Summary

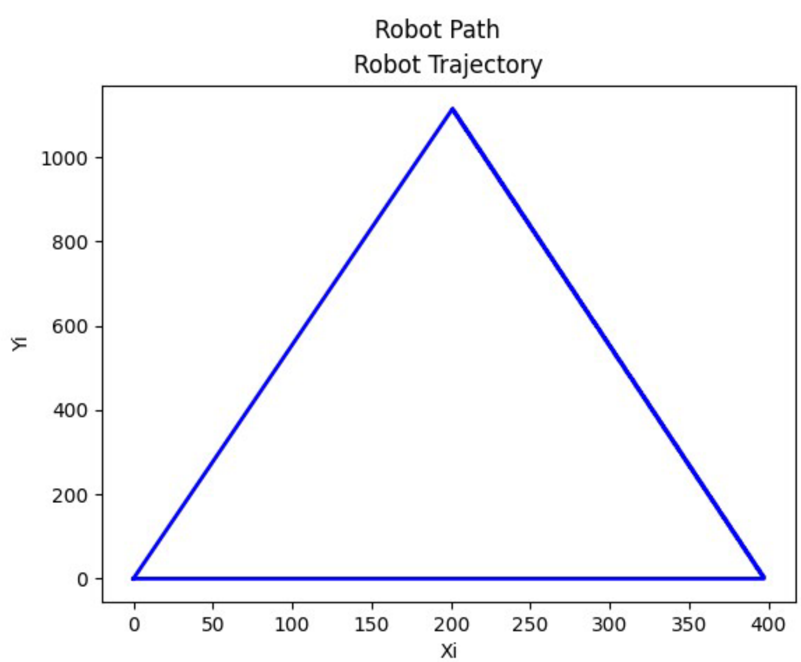
The addition of the IMU sensor on top of the encoder shows an improved performance compared to the encoder only run. The encoder run shows to have no correction after the completion of each command in the sequence; that is, if the robot undershoots or overshoots, it will continue to the next task without adjusting itself thereby propagating error one after the other. The IMU combined set up however, will close the error gap after every task in the sequence thereby ensuring that the robot attains the desired goal state at the end of the user designed sequence.

1.3 Robot Path Plot for Rectangular loop

Encoder Only Guidance



Encoder and IMU Only Guidance



This is an unsuccessful attempt of drawing the map of rectangular loop because IMU kept restarting from 0 everytime I call it within the run. Summary has explained results very well up above.