Laser Tracker Appendix

Developed By: Alex Oh, Ryan Taylor, Arianna Santiago

Table of Contents

Section I - Appendix

A. main.py

```
from runner import Runner
import logging

# main method instantiates Runner class and calls run method
def main():

    myInst = Runner()
    myInst.run()

if __name__ == "__main__":
    main()
```

B. runner.py

```
import threading
from tracker import Tracker
from distancefinder import DistanceFinder
from motorcontrol import MotorControl
from motorlogger import MotorLogger
from retracer import Retracer
from servo import Servo
from Motor import Motor
import time
import logging

# Runner class: initializes and runs threads for tracker, distancfinder,
# motorcontrol, motorlogger, and retracer classes
class Runner(object):
    # Declare Static Variables Here
    # lSpeed and rSpeed are integers that set motor control
    lSpeed = 0
    rSpeed = 0
    # laserPos provides an x and y coordinate of the laser from the camera frame
    laserPos = (-1,-1) # 480 x 640 (h, w)
    # relPos provides x and y coordinates of the robots relative position from
the laser
    relPos = (-1,-1) # Dist (d, x) from robot
    # array that logs motor controls
    log = [] # format: (time, lspeed, rspeed)
    motor = Motor()
    initialPath = True

    # lock1, lock2, and condVar are used manage shared data between
    # tracker, distancefinder, motorcontrol, and motorlogger classes
    lock1 = threading.RLock()
    lock2 = threading.RLock()
    condVar = threading.Condition(lock1)
    pointReady = False

    # position servos head camera into specific orientation
    def __init__(self):
```

```
        self.servo = Servo()
        self.servo.setServoPwm('0', 90)
        self.servo.setServoPwm('1', 10)
        print("Created Runner: Waiting 3 seconds...")
        time.sleep(3)

    # initializes and runs threads of the classes relevant for tracking laser
    def run(self):
        trackingThread = Tracker()
        distanceThread = DistanceFinder()
        controlThread = MotorControl()
        loggingThread = MotorLogger()

        trackingThread.start()
        distanceThread.start()
        controlThread.start()
        loggingThread.start()

        trackingThread.join()
        distanceThread.join()
        controlThread.join()
        loggingThread.join()
        retracerThread = Retracer()
        retracerThread.start()

        retracerThread.join()
```

C. tracker.py

```
import threading
import cv2
import numpy as np
import runner
import time
from picamera2 import Picamera2, Preview
import logging

# Tracker class finds position of laser ptr from camera frame and returns its x,y
coordinates
class Tracker(threading.Thread):

    # Starts camera
    def __init__(self,group=None, target=None, name=None, args=(), kwargs=None,
*, daemon=None):
        super().__init__(group=group, target=target, name=name, daemon=daemon)
        self.camera = Picamera2()
        self.camera.start()
        print("Created Laser Tracker")

    # run loop to continously find x,y coordinates of laser
    def run(self):
        while runner.Runner.initialPath:
            with runner.Runner.condVar:
                runner.Runner.lock1.acquire()
                x,y = self.find_laser()
                #print("Laser found at: ", x, y)
                if x == -1 and y == -1:
                    found = False
                    time.sleep(0.05)
                    for i in range(0,5):
                        x,y = self.find_laser()
                        if not (x == -1 and y == -1):
                            found = True
                            break
                        time.sleep(0.05)
                    if not found:
```

```
                        runner.Runner.initialPath = False
                runner.Runner.laserPos = (x,y)
                runner.Runner.pointReady = True
                #time.sleep(1) #TODO: REMOVE THIS
                runner.Runner.condVar.notify()
                runner.Runner.lock1.release()
            time.sleep(0.02)

    # applies color filter to camera feed to find laser ptr and return its x,y
coordniates
    def find_laser(self):
        filename = "tempRedLaser.jpg"
        self.camera.capture_file(filename)
        image = cv2.imread(filename)

        # red color boundaries [B, G, R]

        lower = [240, 0, 0]
        upper = [255, 240, 240]



        # create NumPy arrays from the boundaries
        lower = np.array(lower, dtype="uint8")
        upper = np.array(upper, dtype="uint8")

        # find the colors within the specified boundaries and apply
        # the mask


        mask = cv2.inRange(image, lower, upper)

        output = cv2.bitwise_and(image, image, mask=mask)

        ret, thresh = cv2.threshold(mask, 100, 255, cv2.THRESH_BINARY)
        contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)

        if len(contours) != 0:
```

```
            # print(len(contours))
            # draw in blue the contours that were founded
            cv2.drawContours(output, contours, -1, -1, 150)

            # find the biggest countour (c) by the area
            c = max(contours, key=cv2.contourArea)
            x, y, w, h = cv2.boundingRect(c)

            #print(x + w / 2, y + h / 2)
            return x + w / 2, y + h / 2

            # draw the biggest contour (c) in green
            # cv2.rectangle(output, (x, y), (x + w, y + h), (0, 255, 0), 2)

        # show the images

        # cv2.imshow("Result", np.hstack([image, output]))

        # cv2.waitKey(0)
        return -1, -1
```

D. distancefinder.py

```
import threading
import runner
import time
import logging

# DistanceFinder class uses the x,y coordniates of the laser ptr from thr Tracker
class
# to determine the relative position the laser ptr is from the robot in x,y
coordinates
class DistanceFinder(threading.Thread):

    def __init__(self, group=None, target=None, name=None, args=(), kwargs=None,
*, daemon=None):
        super().__init__(group=group, target=target, name=name, daemon=daemon)
        print("Created Distance Finder")

    # run loop that continously finds the relPos coordinates
    def run(self):
        while runner.Runner.initialPath:
            with runner.Runner.condVar:
                while not runner.Runner.pointReady:
                    runner.Runner.condVar.wait()
                runner.Runner.lock1.acquire()
                x,y = self.find_distance()
                #print("Distance Away: ", x, y)
                runner.Runner.relPos = (x,y)
                runner.Runner.lock1.release()
            time.sleep(0.02)

    # Algorithm that determins the relPos coordinates
    def find_distance(self):
        a, b = runner.Runner.laserPos
        if a == -1 and b == -1:
            return -1, -1
        d = 40017 * pow(b, -1.376)

        ######################################
```

```
        # Bounds to find off-center distance #
        ######################################
        fL10 = lambda x: -0.11 * x * x + 12.86 * x - 164.4
        fL5 = lambda x: -0.0864 * x * x + 9.3043 * x + 16.8
        fCenter = lambda x: 314
        fR5 = lambda x: 0.0764 * x * x - 8.3843 * x + 603.4
        fR10 = lambda x: 0.1064 * x * x - 12.784 * x + 811.9

        if a < fL10(d):
            # TODO: Fix this for linearization
            # print("Greater than L10")
            return d, -10
        elif a < fL5(d):
            # print("Between 10 and 5 L")
            k = a - fL10(d)
            j = fL5(d) - a
            return d, (-10) * (j / (k + j)) + (-5) * (k / (k + j))
        elif a < fCenter(d):

            # print("Between 5 and 0 L")
            k = a - fL5(d)
            j = fCenter(d) - a
            return d, (-5) * (j / (k + j)) + (0) * (k / (k + j))
        elif a < fR5(d):

            # print("Between 0 and 5 R")
            k = a - fCenter(d)
            j = fR5(d) - a
            return d, (0) * (j / (k + j)) + (5) * (k / (k + j))

        elif a < fR10(d):
            # print("Between 5 and 10 R")
            k = a - fR5(d)
            j = fR10(d) - a
            return d, (5) * (j / (k + j)) + (10) * (k / (k + j))
        else:
            # TODO: Fix this for linearization
            # print("Greater than R10")
            return d, 10
```

E. motorcontrol.py

```
import threading
import runner
import math
import time
import logging

# MotorControl class uses relPos x,y coorndinates to detemine motor controls to
follow laser ptr
class MotorControl(threading.Thread):

    def __init__(self, group=None, target=None, name=None, args=(), kwargs=None,
*, daemon=None):
        super().__init__(group=group, target=target, name=name, daemon=daemon)
        print("Created Motor Controller")

    # run loop that continously generates motor controls
    def run(self):
        while runner.Runner.initialPath:
            runner.Runner.lock1.acquire()
            runner.Runner.lock2.acquire()

            r, theta = self.to_polar(runner.Runner.relPos[0],
runner.Runner.relPos[1])

            baseSpeed = 0
            if r < 10:
                baseSpeed = 0
            elif r < 30:
                baseSpeed = (int) (50 * (r-10))
            else:
                baseSpeed = 1000

            #TODO: Potentially Change to a Polynomial Function for motor offsets
            offset = 2 * math.atan(theta)  # returns value on range [-1,1]
            lSpeed = (int)(baseSpeed * (1 + 1.6 * offset))
            rSpeed = (int)(baseSpeed * (1 - 1.6 * offset))
```

```
            self.set_motors(lSpeed, rSpeed)

            #time.sleep(1) #TODO: REMOVE THIS

            runner.Runner.lock2.release()
            runner.Runner.lock1.release()
            time.sleep(0.02)
        self.set_motors(0,0)

    # converts relPos x,y coordniates to polar form for analysis
    def to_polar(self, d, x):
        radius = math.sqrt(d*d + x*x)
        theta = math.atan(x/d)
        return radius, theta

    # determines lSpeed and rSpeed based off polar coordinates
    def set_motors(self, l, r):
        runner.Runner.motor.setMotorModel(l,l,r,r)
        runner.Runner.lSpeed = l
        runner.Runner.rSpeed = r
```

E. motorlogger.py

```
import threading
import runner
import time
import logging

# MotorLogger class logs motor controls generated by MotorControl class
class MotorLogger(threading.Thread):

    def __init__(self, group=None, target=None, name=None, args=(), kwargs=None,
*, daemon=None):
        super().__init__(group=group, target=target, name=name, daemon=daemon)
        print("Created MotorLogger")


    # run loop that logs lSpeed and rSpeed which sets motor Controls
    def run(self):
        while runner.Runner.initialPath:
            runner.Runner.lock2.acquire()
            runner.Runner.log.append((time.time(), runner.Runner.lSpeed,
runner.Runner.rSpeed))
            #print("Motor Speeds: ", runner.Runner.lSpeed, runner.Runner.rSpeed)
            #time.sleep(1) #TODO: REMOVE THIS
            runner.Runner.lock2.release()
            time.sleep(0.005)
        runner.Runner.log.append((time.time(), runner.Runner.lSpeed,
runner.Runner.rSpeed))
```

F. retracer.py

```
import threading
import runner
import time
import logging

# Retracer class uses motor logs to retrace path taken by following laser ptr
class Retracer(threading.Thread):
    def __init__(self, group=None, target=None, name=None, args=(), kwargs=None,
*, daemon=None):
        super().__init__(group=group, target=target, name=name, daemon=daemon)
        print("Created Retracer")

        i = 0
        self.compressedLog = []

        '''
        print("Initial List:")
        for item in runner.Runner.log:
            print(item)

        '''
        # compression algorithm to reduce number of motor controls in motor logs
to increase retracing precision
        while i < len(runner.Runner.log):
            j = i
            start = runner.Runner.log[i][0]
            while j < len(runner.Runner.log) and (runner.Runner.log[i][1] ==
runner.Runner.log[j][1] and runner.Runner.log[i][2] == runner.Runner.log[j][2]):
                j += 1
            if j == len(runner.Runner.log):
                end = runner.Runner.log[j-1][0] + runner.Runner.log[j-1][0] -
runner.Runner.log[j-2][0]
            else:
                end = runner.Runner.log[j][0]
            self.compressedLog.append((end - start, runner.Runner.log[i][1],
runner.Runner.log[i][2]))
            i = j
```

```
        print("Log Compression Complete")
        time.sleep(10)

    # run loop that sets motor controls from motor logs
    def run(self):


        for item in self.compressedLog:
            self.set_motors(item[1],item[2])
            time.sleep(item[0]/1.15)
        self.set_motors(0,0)


    def set_motors(self, l, r):

        runner.Runner.motor.setMotorModel(l,l,r,r)
```

G. Servo.py

Provided Via the Freenove Open Source GitHub Repository
https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi

```
import time
from PCA9685 import PCA9685
class Servo:
    def __init__(self):
        self.PwmServo = PCA9685(0x40, debug=True)
        self.PwmServo.setPWMFreq(50)
        self.PwmServo.setServoPulse(8,1500)
        self.PwmServo.setServoPulse(9,1500)
    def setServoPwm(self,channel,angle,error=10):
        angle=int(angle)
        if channel=='0':
            self.PwmServo.setServoPulse(8,2500-int((angle+error)/0.09))
        elif channel=='1':
            self.PwmServo.setServoPulse(9,500+int((angle+error)/0.09))
        elif channel=='2':
            self.PwmServo.setServoPulse(10,500+int((angle+error)/0.09))
        elif channel=='3':
            self.PwmServo.setServoPulse(11,500+int((angle+error)/0.09))
        elif channel=='4':
            self.PwmServo.setServoPulse(12,500+int((angle+error)/0.09))
        elif channel=='5':
            self.PwmServo.setServoPulse(13,500+int((angle+error)/0.09))
        elif channel=='6':
            self.PwmServo.setServoPulse(14,500+int((angle+error)/0.09))
        elif channel=='7':
            self.PwmServo.setServoPulse(15,500+int((angle+error)/0.09))

# Main program logic follows:
if __name__ == '__main__':
    print("Now servos will rotate to 90°.")
    print("If they have already been at 90°, nothing will be observed.")
    print("Please keep the program running when installing the servos.")
    print("After that, you can press ctrl-C to end the program.")
    pwm=Servo()
```

```
    while True:
        try :
            pwm.setServoPwm('0',90)
            pwm.setServoPwm('1',10)
        except KeyboardInterrupt:
            print ("\nEnd of program")
            break
```

H. Motor.py

Provided Via the Freenove Open Source GitHub Repository
https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for_Raspberry_Pi

```
import time
from PCA9685 import PCA9685
class Motor:
    def __init__(self):
        self.pwm = PCA9685(0x40, debug=True)
        self.pwm.setPWMFreq(50)
    def duty_range(self,duty1,duty2,duty3,duty4):
        if duty1>4095:
            duty1=4095
        elif duty1<-4095:
            duty1=-4095

        if duty2>4095:
            duty2=4095
        elif duty2<-4095:
            duty2=-4095

        if duty3>4095:
            duty3=4095
        elif duty3<-4095:
            duty3=-4095

        if duty4>4095:
            duty4=4095
        elif duty4<-4095:
            duty4=-4095
        return duty1,duty2,duty3,duty4

    def left_Upper_Wheel(self,duty):
        duty = duty *-1
        if duty>0:
            self.pwm.setMotorPwm(0,0)
            self.pwm.setMotorPwm(1,duty)
        elif duty<0:
```

```
            self.pwm.setMotorPwm(1,0)
            self.pwm.setMotorPwm(0,abs(duty))
        else:
            self.pwm.setMotorPwm(0,4095)
            self.pwm.setMotorPwm(1,4095)
    def left_Lower_Wheel(self,duty):
        duty = duty *-1
        if duty>0:
            self.pwm.setMotorPwm(3,0)
            self.pwm.setMotorPwm(2,duty)
        elif duty<0:
            self.pwm.setMotorPwm(2,0)
            self.pwm.setMotorPwm(3,abs(duty))
        else:
            self.pwm.setMotorPwm(2,4095)
            self.pwm.setMotorPwm(3,4095)
    def right_Upper_Wheel(self,duty):
        duty = duty *-1
        if duty>0:
            self.pwm.setMotorPwm(6,0)
            self.pwm.setMotorPwm(7,duty)
        elif duty<0:
            self.pwm.setMotorPwm(7,0)
            self.pwm.setMotorPwm(6,abs(duty))
        else:
            self.pwm.setMotorPwm(6,4095)
            self.pwm.setMotorPwm(7,4095)
    def right_Lower_Wheel(self,duty):
        duty = duty *-1
        if duty>0:
            self.pwm.setMotorPwm(4,0)
            self.pwm.setMotorPwm(5,duty)
        elif duty<0:
            self.pwm.setMotorPwm(5,0)
            self.pwm.setMotorPwm(4,abs(duty))
        else:
            self.pwm.setMotorPwm(4,4095)
            self.pwm.setMotorPwm(5,4095)
```

```
    def setMotorModel(self,duty1,duty2,duty3,duty4):
        duty1,duty2,duty3,duty4=self.duty_range(duty1,duty2,duty3,duty4)
        self.left_Upper_Wheel(duty1)
        self.left_Lower_Wheel(duty2)
        self.right_Upper_Wheel(duty3)
        self.right_Lower_Wheel(duty4)


PWM=Motor()
def loop():
    PWM.setMotorModel(2000,2000,2000,2000)       #Forward
    time.sleep(3)
    PWM.setMotorModel(-2000,-2000,-2000,-2000)   #Back
    time.sleep(3)
    PWM.setMotorModel(-500,-500,2000,2000)       #Left
    time.sleep(3)
    PWM.setMotorModel(2000,2000,-500,-500)       #Right
    time.sleep(3)
    PWM.setMotorModel(0,0,0,0)                   #Stop

def destroy():
    PWM.setMotorModel(0,0,0,0)
if __name__=='__main__':
    try:
        loop()
    except KeyboardInterrupt:  # When 'Ctrl+C' is pressed, the child program
destroy() will be  executed.
        destroy()
```