Robotics Lab 4: Server Client Programming

Authors: Stuart Harley, Nathan Chapman

Date: 9/28/2021

Abstract

In robotics, programming a robot can either be done offline or following a server/client protocol. Offline programming is what we have done in previous labs, where we develop machine code on a PC and compile it and upload it to the robot to run without further intervention. Server/Client programming is when a robot is encoded with a program that allows messages and commands to be sent back and forth from a client to enable remote control of the robot. In this lab, we explore with building a server/client programming framework to control our robot. To begin with, we have the robot travel and rotate by sending commands that the robot receives and performs. Next, we have the client query the sensors on the bot to ensure that we have access to read the environment surrounding the robot. Then we add in functionality to move and turn on button presses. Finally, we implement a line-follow algorithm with all logic taking place on the client and only query, move, and turn commands sent to the robot. Overall, our lab showed no statistical difference of server/client programming from offline control, moves as deemed appropriate, and is capable of following a line.

Methods

Task 1

In task 1 we are simply running the provided server-client demo code to confirm that the messages are being sent by the client and received by the robot correctly. The robot was sent a "playNote" message with an argument corresponding to the pitch of the note to be played. The robot would then play a note at the specified pitch.

Task 2

In task 2, we were asked to program the robot to receive move signals from a client. Using these commands, the robot was supposed to either move 25 cm forward or rotate 90 degrees. To do this, we created two methods, a drive_dist command, and a rotate command. The drive_dist command sends one argument, distance in centimeters, specifying how far to move the robot forward or backward. The speed of the wheels is set at 90 degrees per second because this speed was found to deliver the most consistent results in Lab 1. It then uses the equation below to calculate the time it needs to run in order to travel the given distance.

t = d x .235sec/cm

In this equation, t is the time the robot will sleep for, d is the distance we want to travel, and sec/cm is a constant since we declared the speed a constant 90 degrees per second. This sec/cm constant calculates to be .235 sec/cm. This logic is computed on the robot itself because we know exactly how far we want the robot to move. Therefore, we don't want to send both a

start and stop command which could run into issues while transmitting and cause the robot to be inaccurate.

The turning equation is similar, where the command turn has two arguments, direction (clockwise, counterclockwise) and degrees. We use the same concept as before, where we calculate runtime using degrees times a constant.

t = d x .023sec/deg

In this equation, t is the time the robot will sleep while the wheels turn in opposite directions. d is the number of degrees the robot will turn. And since the wheel speed is set at a constant 90 degrees per second, the sec/degree constant calculates to be .023 sec/degree.

For measuring the distance the robot moved in task 2, a ruler was placed next to the robot and the robot was placed with the center of the wheel at 0cm. The robot moved forward until stopping and the distance at the center of the wheel was recorded. Example picture below shown in Image 1.

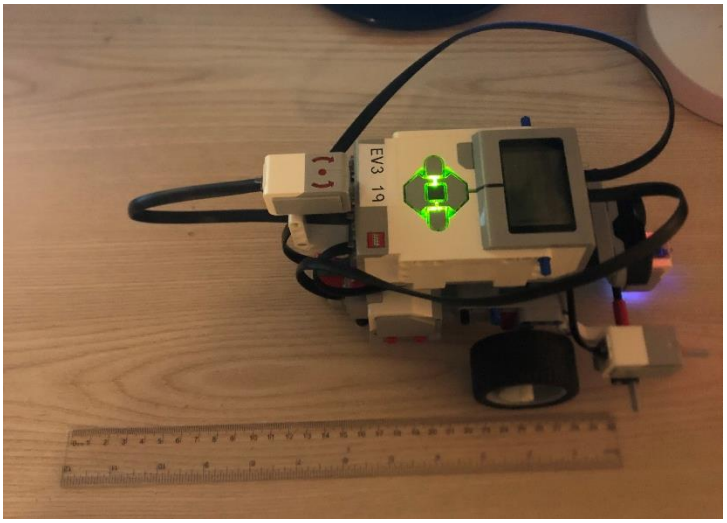

Image 1: Setup for measuring forward movement

For measuring the amount the robot rotated in task 2, a sheet of paper was marked with a circle and starting locations for the wheels. The robot rotated until stopping and the actual degrees rotated were recorded. Example picture below shown in Image 2.

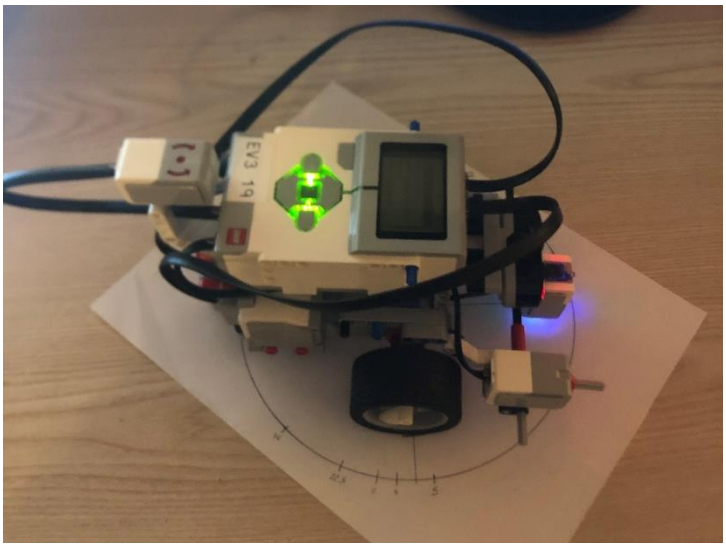

Image 2: Setup for measuring rotational movement

After completing this task and recording the results we performed a two-sided T-test to determine if the robot's movements matched the movements that we recorded back in Lab 1. The null hypothesis for each test was that the robot's movements in the offline (Lab 1) and server/client (Lab 4) tests were not significantly different. The alternative hypothesis is that the robot's movements were significantly different. In this case we are rejecting the null hypothesis when the p-value of the test is less than or equal to 0.05 ($p <= 0.05$).

Task 3

In task 3, we had to query sensors and send the output back to the client. We implemented a query command for this, and it returns the value of the requested port's sensor at that point in time. On the client side, we also have a query_all command that sends a query command to each sensor once a second until the program is shut down. Because the sensor functions are specific to the type of sensor that is attached to the port, we had to standardize which sensors are plugged into which port. Accordingly, we chose port 1 to be the color sensor, port 2 to be the ultrasonic sensor, port 3 to be the gyroscopic sensor, and port 4 to be the touch sensor.

Task 4

Task 4 asked us to implement WASD control, where the robot would move forward if W was pressed, left if A is pressed, right if S is pressed, backwards if D is pressed, and stop if space is pressed. On the client side, a user must enter a w, a, s, d, or space command, and the robot will be sent a drive or turn command that will make the robot continue the specified movement until a new command is sent.

Task 5

In task 5, we had to implement out line following algorithm from lab 4 using client-host based programming. To do this, we simply replaced all the offline movement commands in our

previous lab with server/client movement commands (drive, turn). We also query the color sensor at .25 second intervals to check if we are on or off the line. All of the logic is computed by the client, with the robot only being sent the specific movement and query commands.

For testing, a line jagged was drawn on the ground using masking tape that the robot would follow as shown below in Image 3. The robot starts at one end of the line with the color sensor overtop of the tape line. Whatever color is sensed under the color sensor when the line following algorithm is started is the color that the robot will assume the line is. In the setup below, our line is black.

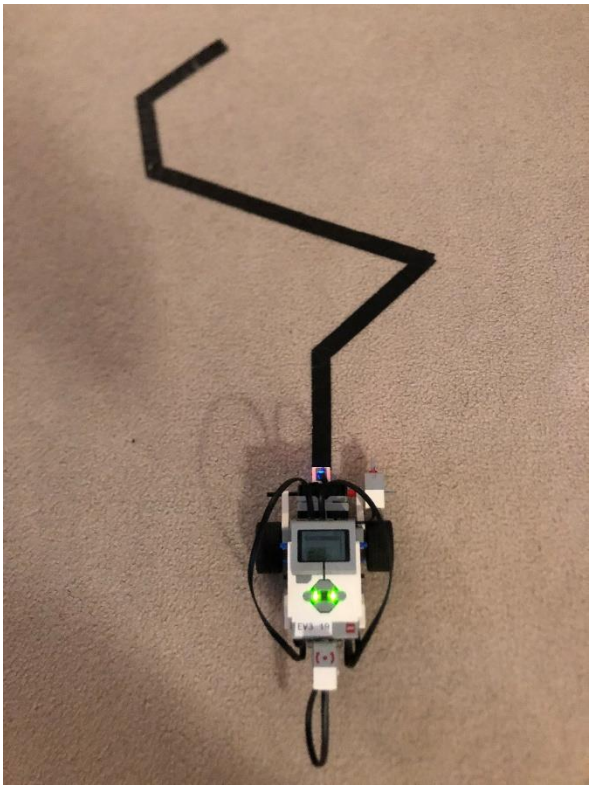

Image 3: Setup for robot following a line

Results

Task 1

The setup task has no real results associated with it. Our robot functioned as expected with the server/client commands functioning appropriately.

Task 2

In task 2, we compared the distance the robot moved straight and turning to the distances the robot moved from Lab 1. Since we had set the speed of the robot at 90 degrees per second for this lab, we used the test cases from lab 1 where the robot's speed was also 90 degrees per second. The results from these trials as well as the p-values from out two sample t-tests are shown below in tables 1, 2, and 3.

Trial	Distance	Angle (CW)	Angle (CCW)
1	24.9	94.2	92.7
2	24.9	93.6	84.3
3	25.0	93.2	93.9
4	24.8	97.7	83.7
5	25.6	92.8	88.2
Mean	25.04	94.3	88.56
Std Dev	0.287054	1.761817	4.184543

Table 1: Movement results at 90 degrees per second from Lab 1

Trial	Distance	Angle (CW)	Angle (CCW)
1	25.1	92.7	91.7
2	25.0	84.3	89.1
3	25.1	93.9	87.8
4	24.9	83.7	89.9
5	24.9	88.2	91.2
Mean	25.0	88.56	89.94
Std Dev	0.089442719	4.184543	1.412232

Table 2: Movement results at 90 degrees per second using server/client commands

Two-Sample t-test results of lab 1 vs current lab results			
	Distance	CW Turn	CCW Turn
P-value	0.39845	0.01767	0.27470

Table 3: P-value results from two-sided t-tests

As shown above, the T-test values show no statistical difference between live control (this lab) and offline control (lab 1) for the straight distance and CCW turn. However, the results show a significant difference for the CW turn. However, by inspecting the results it appears that the live control (this lab) results were more accurate than the offline control (lab 1) results. This was likely due to minor differences in the robot build being used in these tests.

Task 3

For task 3, our query command ran smoothly. It took little time to implement and works well. Image 4 below shows an example of Phase 1 of the query protocol: Receiving a sensor request for a specified port. The query message is received by the server and logged to the console as requesting the value of the sensor at port 1.

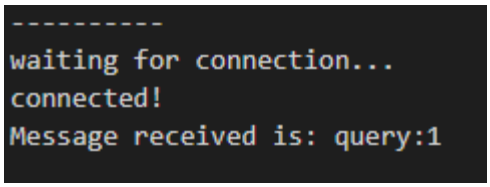

Image 4: Screenshot of output for query phase 1

Image 5 below shows Phase 2 of the query protocol: Looking up what sensor type is attached to the port. There was not output to either the client or server consoles for this phase of the protocol so shown below is the code snippet from the server code where it calls the appropriate sensor function.

```
if arg is '1': # Color sensor
    data = s1_sensor.color()
elif arg is '2': # Ultrasonic sensor
    data = s2_sensor.distance()
elif arg is '3': # Gyroscope sensor
    data = s3_sensor.speed()
elif arg is '4': # Touch sensor
    data = s4_sensor.pressed()
```

Image 5: Screenshot of code for query phase 2

Image 6 below shows Phase 3 of the query protocol: sending the result of the query back to the client. Since we queried port 1 (color sensor), we got back a color reading. These results are printed to the client console when received.

```
Enter a cmd (play, buttons, drive, drive_dist, turn, query, query_all): >? query
Enter Sensor # 1(Color) 2(Ultrasonic) 3(Gyro) 4(Touch)>? 1
Sensor  1 :  Color.WHITE
```

Image 6: Screenshot of output for query phase 3

Image 7 below shows the output of a few seconds of query all, where we query each sensor every second. As you can see, the values all change as the sensors detect changes in the environment or position of the robot.

```
Enter a cmd (play, buttons, drive, drive_dist, turn, query, query_all): >? query_all
    Sensor 1: Color.WHITE   Sensor 2: 213   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 213   Sensor 3: -5    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 213   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 249   Sensor 3: 54    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 235   Sensor 3: -6    Sensor 4: True
    Sensor 1: Color.WHITE   Sensor 2: 228   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 243   Sensor 3: -12   Sensor 4: False
    Sensor 1: Color.BROWN   Sensor 2: 72    Sensor 3: -8    Sensor 4: False
    Sensor 1: Color.BROWN   Sensor 2: 91    Sensor 3: -2    Sensor 4: False
    Sensor 1: None  Sensor 2: 328   Sensor 3: -46   Sensor 4: False
    Sensor 1: Color.BLACK   Sensor 2: 340   Sensor 3: 9 Sensor 4: False
    Sensor 1: Color.BLUE    Sensor 2: 199   Sensor 3: -24   Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
    Sensor 1: Color.WHITE   Sensor 2: 199   Sensor 3: -6    Sensor 4: False
```

Image 7: Screenshot of output of live sensor query

Task 4

Task 4 was the live control task to be performed by using the WASD buttons and commands. We were able to maneuver the robot almost instantaneously using these commands. Shown below, we have screenshots of different points of the submitted task 4 video where we demonstrate each movement capability.

First, shown in Image 8 below, we have the robot moving forward after a W command.

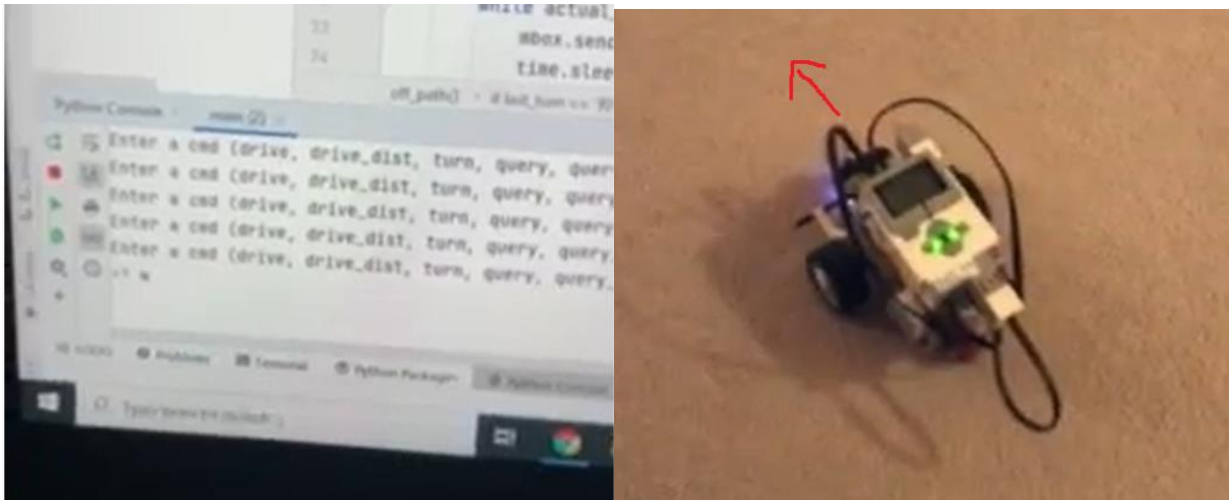

Image 8: Screenshot of task 4 video. W key input. Robot moves forward.

Second, shown in Image 9 below, we have an A input causing the robot to turn left.

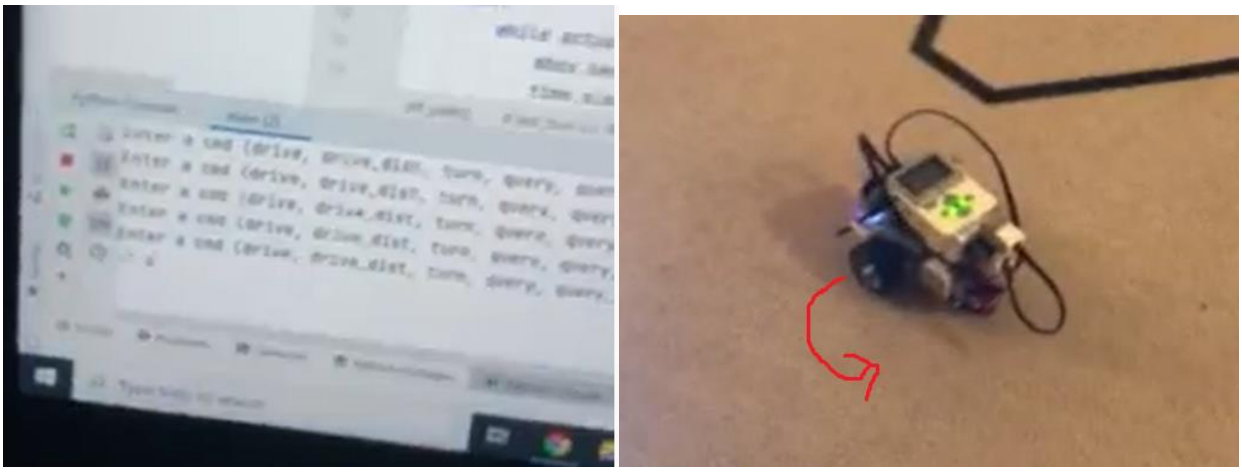

Image 9: Screenshot of task 4 video. A key input. Robot turns left.

Third, shown in Image 10 below, the robot turns right after a D input.

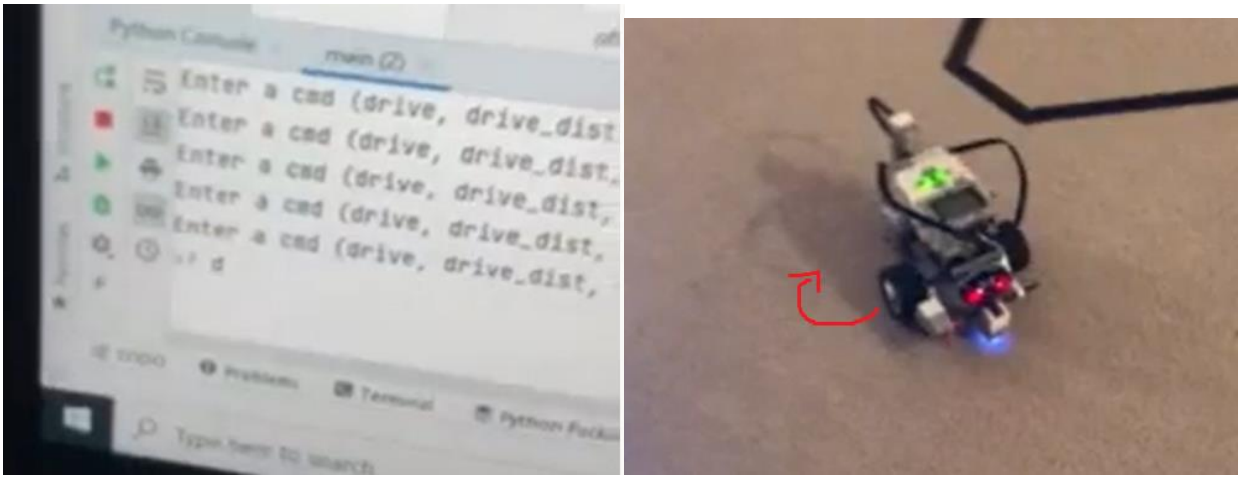

Image 10: Screenshot of task 4 video. D key input. Robot turns right.

Fourth, shown in Image 11 below, the robot moves backwards once it receives an S command.

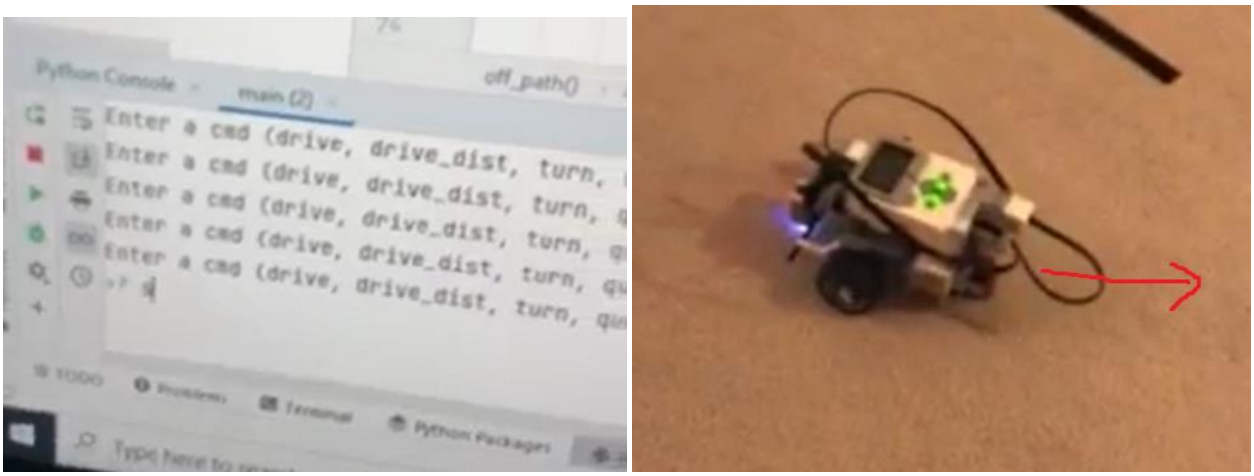

Image 11: Screenshot of task 4 video. S key input. Robot moves backwards.

Finally, shown in Image 12 below, the robot stops moving after receiving the space command.

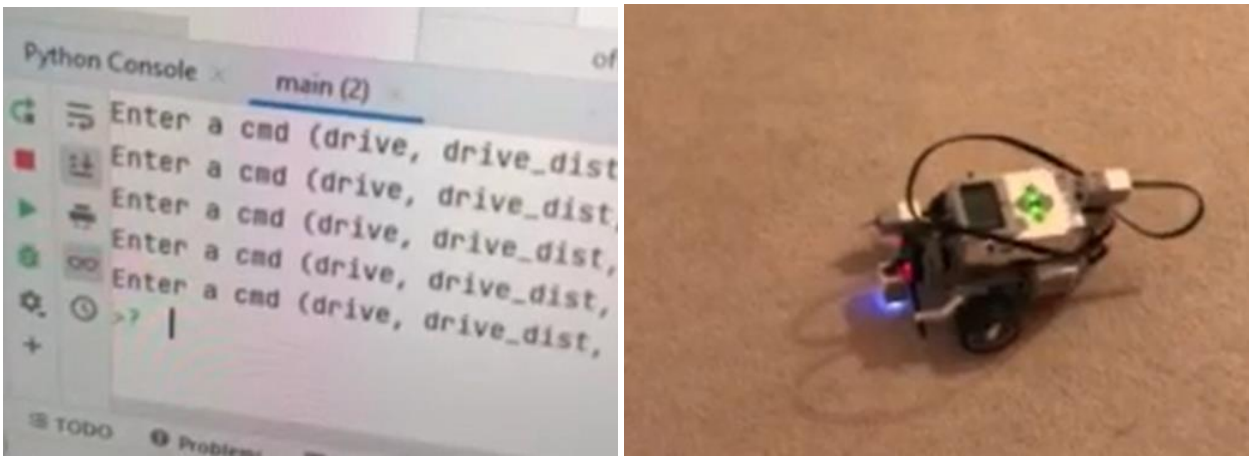

Image 12: Screenshot of task 4 video. 'Space' key input. Robot stops.

Task 5

Task 5 asked us to implement a client-host based line follow. Shown below, we have screenshots of different points of the submitted task 5 video where we demonstrate different stages of the robot completing the line following task.

First, shown in Image 13 below, we see the robot begin its journey on the line and continue forward until it no longer senses the line.

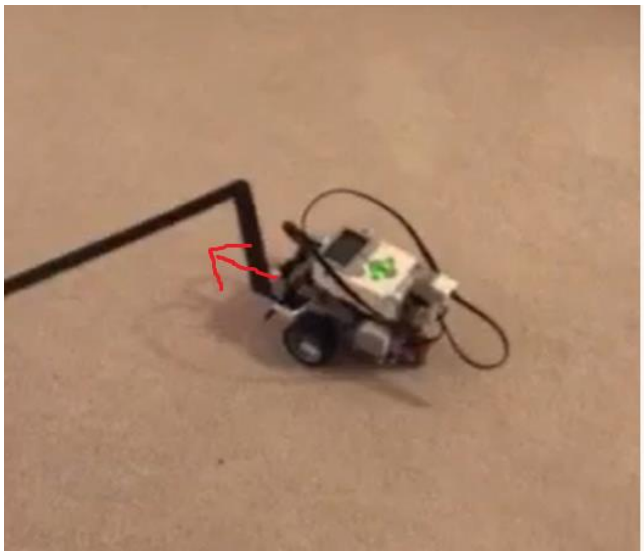

Image 13: Screenshot of task 5 video. Robot is starting to traverse the line.

Next, shown in Image 14 below, we see the robot initially turn right looking for the line. Once it turns approximately 150 degrees and doesn't find the line, it rotates back to the left to find the line.

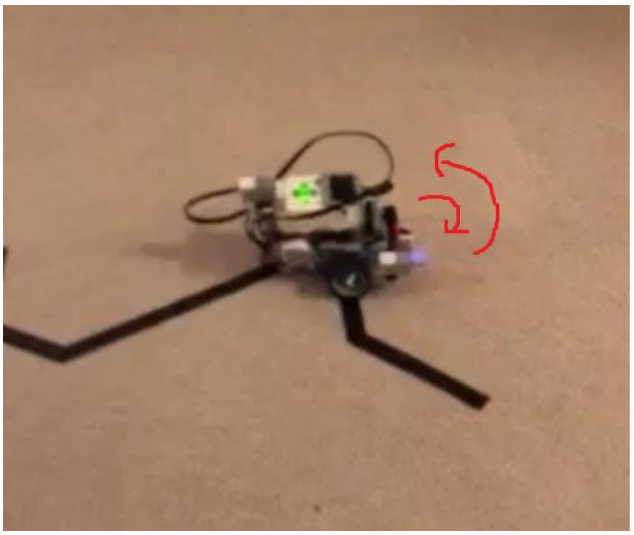

Image 14: Screenshot of task 5 video. Robot turned right in an attempt to find the line. It didn't and is now turning back left.

At a later turn, shown in Image 15 below, we see the robot initially turn left looking for the line. Once it turns approximately 150 degrees and doesn't find the line, it rotates back to the right to find the line.

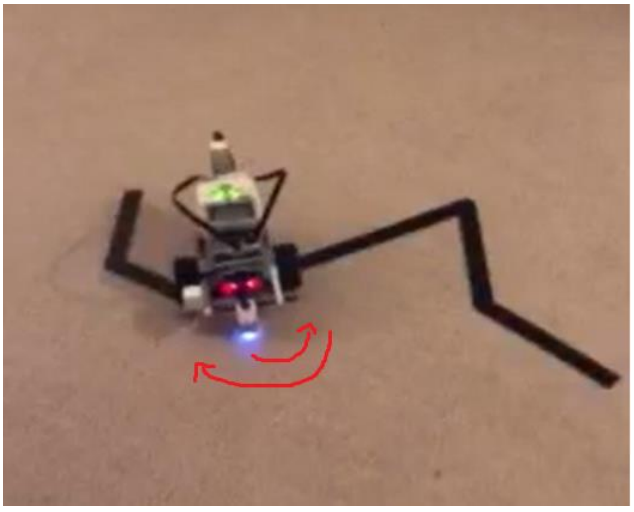

Image 15: Screenshot of task 5 video. Robot turned left in an attempt to find the line. It didn't and is now turning back right.

Finally, shown in Image 16 below, the robot reaches the end of the line.

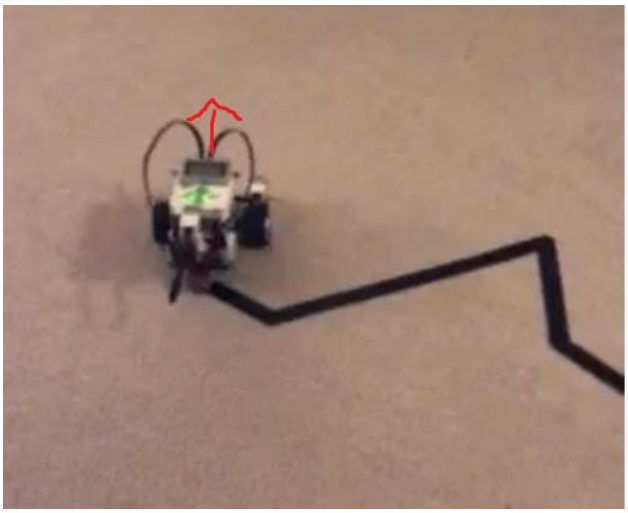

Image 16: Screenshot of task 5 video. Robot finishes traversing the line.

Discussion

1. Explain the difference between a 1-**sample** and 2-**sample** t-test? What are we using a 2-**sample** test here instead of 1-**sample**?
    a. A one sample test compares a set of values from what is considered acceptable, whereas a two-sample test compares two sets of values to each other and determines if they are statistically different or not.

2. Was there a statistical difference between telling the robot to move or turn with the offline control from Lab 1 and the server-client control you created? If there was no difference, what factors could lead to a difference? If there was a difference, where do you think it came from? Hint, think about the cons for server-client control.
    a. There was no difference for the straight and CCW turn cases. However, there was a difference for the CW turn case. But when examining the results, it appears that the server/client turn was more accurate than the offline turn. This was likely due to minor differences in the robot build for the two labs. Because we calculated the time the robot needed to move on the server side instead of sending a start and stop message from the client, the robot was able to achieve the same accuracy as the offline control tests from Lab 1 since there was no possibility of lag or delay in the stop message.

3. What were some of the design considerations you thought about when coming up with the message protocol for motor commands and sensor query?
    a. For the motor commands, the main consideration was what the robot needed to know in order to move. For straight movement, the user can specify if they want

just one or both of the motors to spin, the speed they want the robot to move, and the amount of time to move. However, by leaving the time as 0 we created an option to make the robot move continuously until a different command was received which we utilized for tasks 4 and 5. We also made a separate drive method for task 2 called drive_dist where the speed was set at a constant and the only input was the distance to move in cm. This message protocol allowed us to be very accurate with our precise distance movements. The combination of both of these movement messages allowed us to be able to succeed at tasks requiring different aspects of straight movement. Similarly to this movement message, we created the turn message to take in a direction and the total of degrees to turn. The speed was set at a constant so this method would turn the robot very accurately.

When considering the sensor query method, we arbitrarily picked which sensor ports would be connected to which sensors since there is not impact on performance. We also did not see a need to format the responses in any way. We simply sent back the exact results of the query to the client. In this way, the client can see exactly what the sensors recorded and use that data accordingly.

4. What are some of the applications of controlling a robot live like you did with the WASD program?
    a. One example would be navigating a maze or obstacle course, which has its own applications. You could be taking part in a race, navigating a house that is on fire to find and rescue survivors, or even moving towards an active bomb to disable it.

5. What was the most difficult part of this Lab?
    a. The most difficult part was implementing timed sleeps between the motor commands and the query commands for the line following task. Because the robot can only perform one operation at a time (either moving or querying) in this client/server environment, if we didn't implement a sleep between the movement command and the query command the query command would start immediately stopping the movement command and the robot would never move. Therefore, we had to implement small delays between the movement commands and query commands so that the robot would have time to move, but also wouldn't be able to move too far in that time such that it would mess up the rest of the task.

Supplementary Materials

Contained in this section are code screenshots for tasks 1 through 5. Videos of task 4 and 5 are also included in the submission.

Python Code

The python files included in the submission are client_main.py and server_main.py which contain the client and server code for tasks 1 through 5. Screenshots of the python code are shown below.

server_main.py is shown below. This code controls the server-side functions of the robot.

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.ev3devices import (Motor, TouchSensor, ColorSensor,
                                 InfraredSensor, UltrasonicSensor, GyroSensor)
from pybricks.parameters import Port, Stop, Direction, Button, Color
from pybricks.tools import wait, StopWatch, DataLog
from pybricks.robotics import DriveBase
from pybricks.media.ev3dev import SoundFile, ImageFile
from pybricks.messaging import BluetoothMailboxServer, TextMailbox

from _thread import allocate_lock

import time

DEG_PER_SEC = 90
RUN_TIME_PER_CM = .235
TURN_TIME_PER_DEG = .023

# Create your objects here.
ev3 = EV3Brick()
motor1 = Motor(Port.B)
motor2 = Motor(Port.C)
s1_sensor = ColorSensor(Port.S1)
s2_sensor = UltrasonicSensor(Port.S2)
s3_sensor = GyroSensor(Port.S3)
s4_sensor = TouchSensor(Port.S4)

# Write your program here.
ev3.speaker.beep()

#Note the timeout parameter doesn't seem to work
#This wait checks for a message and then just returns
#Compared to wait() which will sit there and wait
#for a message
def wait_with_timeout(mbox,name):
        """Waits until ``mbox`` receives a value."""
        lock = allocate_lock()
        lock.acquire()
        with mbox._connection._lock:
            mbox._connection._updates[name] = lock
        try:
            #timeout parameter doesn't work as
            #expected so I'm just setting
            #the wait flag to be zero
            return lock.acquire(0,1)
        finally:
            with mbox._connection._lock:
                del mbox._connection._updates[name]

def ping():
    server = BluetoothMailboxServer()
    mbox = TextMailbox('greeting', server)

    # The server must be started before the client!
    print('waiting for connection...')
    server.wait_for_connection()
    print('connected!')
    # In this program, the server waits for the client to send the first message
    # and then sends a reply.
    mbox.wait()
    print(mbox.read())
    mbox.send('hello to you!')
```

```
def setupConnection():
    server = BluetoothMailboxServer()
    mbox = TextMailbox('greeting', server)

    # The server must be started before the client!
    print('waiting for connection...')
    server.wait_for_connection()
    print('connected!')
    return mbox

def receiveMsg():
    mbox = setupConnection()
    while 1:
        mbox = setupConnection()
        mbox.wait()
        msg = mbox.read()
        print("Message received is: "+msg)
        if msg == "q":
            break

def playNotes():
    mbox = setupConnection()

    while 1:
        wait_with_timeout(mbox,mbox.name)
        msg = mbox.read()
        if msg != None:
            print("Message received is: "+msg)
            if msg == "q":
                break
            elif msg == "a":
                ev3.speaker.beep(880,100)
            elif msg == "c":
                ev3.speaker.beep(523,100)
            elif msg == "g":
                ev3.speaker.beep(783,100)
            mbox._connection._mailboxes={}
            msg = None

def server():
    mbox = setupConnection()
    pressed = ""
    while 1:
        wait_with_timeout(mbox,mbox.name) #non-blocking
        #wait() #blocking
        msg = mbox.read()
        #if pressed != []:
        #    print("pressed  is "+str(pressed))
        if msg != None:
            print("Message received is: "+msg)
            #assumes there are only two tokens
            cmd,arg = mbox.read().split(":")
            if cmd == "play":
                #print("arg is "+arg)
                if arg == "q":
                    break
                elif arg == "a":
                    ev3.speaker.beep(880,100)
                elif arg == "c":
                    ev3.speaker.beep(523,100)
                elif arg == "g":
                    ev3.speaker.beep(783,100)
            elif cmd == "buttons":
                #print("pressed is "+str(pressed))
                mbox.send(pressed)
```

```
        elif cmd == "drive":
            motor, speed, runtime = arg.split(',')
            speed = float(speed)
            runtime = float(runtime)
            if motor is '1':
                motor1.run(speed)
            elif motor is '2':
                motor2.run(speed)
            elif motor is '3':
                motor1.run(speed)
                motor2.run(speed)
            if int(runtime) is not 0:
                time.sleep(runtime)
                motor1.brake()
                motor2.brake()
        elif cmd == "drive_dist":
            dist = float(arg)
            motor1.run(DEG_PER_SEC)
            motor2.run(DEG_PER_SEC)
            time.sleep(dist * RUN_TIME_PER_CM)
            motor1.brake()
            motor2.brake()
        elif cmd == "turn":
            direction, degrees = arg.split(',')
            runtime = float(degrees) * TURN_TIME_PER_DEG
            if direction is 'cw':
                motor1.run(DEG_PER_SEC)
                motor2.run(-DEG_PER_SEC)
            elif direction is 'ccw':
                motor1.run(-DEG_PER_SEC)
                motor2.run(DEG_PER_SEC)
            if int(degrees) is not 0:
                time.sleep(runtime)
                motor1.brake()
                motor2.brake()
        elif cmd == "query" or "query_all":
            data = None
            if arg is '1': # Color sensor
                data = s1_sensor.color()
            elif arg is '2': # Ultrasonic sensor
                data = s2_sensor.distance()
            elif arg is '3': # Gyroscope sensor
                data = s3_sensor.speed()
            elif arg is '4': # Touch sensor
                data = s4_sensor.pressed()
            mbox.send(data)
        else:
            print("Received invalid cmd "+cmd)
        #clears msg and the mailbox
        msg = None
        mbox._connection._mailboxes={}

    buttons = ev3.buttons.pressed()
    #gets the buttons pressed if any
    if buttons != []:
        pressed = "" #clears pressed
        for i in range(len(buttons)):
            pressed+=str(buttons[i])
            if i!=len(buttons)-1:
                pressed+=":"
```

```
def printButtons():
    pressed = ""
    while True:
        buttons = ev3.buttons.pressed()
        print(buttons)
        #gets the buttons pressed if any
        if buttons != []:
            pressed = "" #clears pressed
            for i in range(len(buttons)):
                pressed+=str(buttons[i])
                if i!=len(buttons)-1:
                    pressed+=":"
        print("Pressed is "+pressed)
def main():
    #receiveMsg()
    #playNotes()
    server()
    #printButtons()

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

client_main.py is shown below. This code controls the client-side functions of the robot.

```
#!/usr/bin/env python3
from pybricks.messaging import BluetoothMailboxClient, TextMailbox

import time

last_turn = 'RIGHT'

#client code to be run on a pc

SERVER = '24:71:89:4a:f5:db'
def ping():
    client = BluetoothMailboxClient()
    mbox = TextMailbox('greeting', client)

    print('establishing connection...')
    client.connect(SERVER)
    print('connected!')

    # In this program, the client sends the first message and then waits for the
    # server to reply.
    mbox.send('hello!')
    mbox.wait()
    print(mbox.read())

def sendMsg():
    client = BluetoothMailboxClient()
    mbox = TextMailbox('greeting', client)

    print('establishing connection...')
    client.connect(SERVER)
    print('connected!')
    while 1:
        cmd = input("Enter a letter: ")
        print("Sent command was: "+cmd)
        mbox.send(cmd)
```

```
def on_path(mbox, line_color):
    actual_color = line_color
    while actual_color == line_color:
        mbox.send('drive:3,90,0')
        time.sleep(.25)
        mbox.send('query:1')
        mbox.wait()
        actual_color = mbox.read()
    mbox.send('drive:3,0,.01')
    time.sleep(.1)
    off_path(mbox, line_color, actual_color)

def off_path(mbox, line_color, actual_color):
    global last_turn
    start = time.perf_counter()
    elapsed = time.perf_counter() - start
    if last_turn == 'RIGHT':
        while actual_color != line_color and elapsed < 3.5:
            mbox.send('turn:cw,0')
            time.sleep(.05)
            elapsed = time.perf_counter() - start
            mbox.send('query:1')
            mbox.wait()
            actual_color = mbox.read()
        if actual_color == line_color:
            last_turn = 'RIGHT'
        else:
            while actual_color != line_color:
                mbox.send('turn:ccw,0')
                time.sleep(.05)
                mbox.send('query:1')
                mbox.wait()
                actual_color = mbox.read()
            last_turn = 'LEFT'
    else:
        while actual_color != line_color and elapsed < 3.5:
            mbox.send('turn:ccw,0')
            time.sleep(.05)
            elapsed = time.perf_counter() - start
            mbox.send('query:1')
            mbox.wait()
            actual_color = mbox.read()
        if actual_color == line_color:
            last_turn = 'LEFT'
        else:
            while actual_color != line_color:
                mbox.send('turn:cw,0')
                time.sleep(.05)
                mbox.send('query:1')
                mbox.wait()
                actual_color = mbox.read()
            last_turn = 'RIGHT'
    mbox.send('drive:3,0,.01')
    time.sleep(.1)
    on_path(mbox, line_color)
```

```python
def client():
    client = BluetoothMailboxClient()
    mbox = TextMailbox('greeting', client)
    print('establishing connection...')
    client.connect(SERVER)
    print('connected!')
    while 1:
        cmd = input("Enter a cmd (drive, drive_dist, turn, query, query_all, wsad or space, line_follow): ")
        if cmd == "play":
            freq = input("Enter a frequency")
            msg = cmd+":"+freq
            print("Sending message "+msg)
            mbox.send(cmd+":"+freq)
        elif cmd =="buttons":
            msg = cmd + ":" + "dummy"
            print("Sending message " + msg)
            mbox.send(msg)
            mbox.wait() #wait for a response
            b = mbox.read()
            print("lasted button pressed was "+str(b))
        elif cmd == 'drive':
            msg = cmd + ':'
            motors = input("Which motor (1 for L, 2 for R, 3 for both): ")
            msg = msg + motors
            speed = input("Enter a speed ")
            msg = msg + ',' + speed
            sec: str = input("Enter a time ")
            msg = msg + ',' + sec
            print("Sending Message: " + msg)
            mbox.send(msg)
        elif cmd == 'drive_dist':
            msg = cmd + ':'
            dist = input("Enter a distance (cm): ")
            msg = msg + dist
            print("Sending message " + msg)
            mbox.send(msg)
        elif cmd == 'turn':
            msg = cmd + ':'
            motor = input("Enter Clockwise (cw) or CounterClockwise (ccw): ")
            msg = msg + motor + ','
            degrees = input('Enter a turn angle: ')
            msg = msg + degrees
            print("Sending message: " + msg)
            mbox.send(msg)
        elif cmd == 'query':
            msg = cmd + ':'
            snum = input("Enter Sensor # 1(Color) 2(Ultrasonic) 3(Gyro) 4(Touch)")
            msg = msg + snum
            mbox.send(msg)
            mbox.wait()
            data = mbox.read()
            print("Sensor ", snum, ": ", data)
```

```
        elif cmd == 'query_all':
            while True:
                out = ''
                for i in range(4):
                    mbox.send(cmd + ':' + str(i+1))
                    mbox.wait()
                    out = out + '\tSensor ' + str(i+1) + ': ' + mbox.read()
                print(out)
                time.sleep(1)
        elif cmd == 'w':
            mbox.send('drive:3,180,0')
        elif cmd == 's':
            mbox.send('drive:3,-180,0')
        elif cmd == 'a':
            mbox.send('turn:ccw,0')
        elif cmd == 'd':
            mbox.send('turn:cw,0')
        elif cmd == ' ':
            mbox.send('drive:3,0,.01')
        elif cmd == 'line_follow':
            mbox.send('query:1')
            mbox.wait()
            line_color = mbox.read()
            on_path(mbox, line_color)
        else:
            print("Unrecognized cmd: "+cmd)


def main():
    #ping()
    #sendMsg()
    client()
if __name__ == '__main__':
    main()
```

Videos

Task 4 video: This video show the robot reacting to the WSAD and space commands that are sent to it by moving accordingly.

Task 5 video: This video shows the robot following the jagged line on the ground.