CTRL Z

TEAM 5031

Program Documentation

Contents

Main Selection Program (main.py)

```
# The Sound class creates a new instance that is assigned to the variable
created.

soundGenerator = Sound()

# buttonListener generates an event when the user presses the button

buttonListener = Button()


selectedProgram = 0

# These are our missions

missionNames = ["craney", "trafficJam", "tree", "blockDelivery1", "redCircle",
"calibration", "showGyro"]

missions = [craney, trafficJam, tree, blockDelivery1, redCircle, calibration,
showGyro]

numMissions = len(missionNames)


# buttonListener is called when the user presses the left button

def left(pressed):

    global selectedProgram

    if pressed and selectedProgram > 0:

        selectedProgram = selectedProgram - 1

        print(missionNames[selectedProgram])


# buttonListener is called when the user presses the right button

def right(pressed):

    global selectedProgram

    if pressed and selectedProgram < numMissions - 1:

        selectedProgram = selectedProgram + 1

        print(missionNames[selectedProgram])


# buttonListener is called when the user presses the enter (middle) button

def enter(pressed):

    global selectedProgram
    if pressed:
        soundGenerator.beep()
        try:
            missions[selectedProgram].run()
            if selectedProgram < numMissions-1:
                selectedProgram = selectedProgram + 1
        except  Exception as e:
            soundGenerator.beep()
            robot.debug("EXCEPTION")
            robot.debug(e)
            selectedProgram = selectedProgram + 1
        robot.afterMission()
        print(missionNames[selectedProgram])

# Register the buttonListener
buttonListener.on_left = left
buttonListener.on_right = right
buttonListener.on_enter = enter

soundGenerator.beep()

# Read the calibrated values and test if the Gyro is drifting

robot.init()
testoPesto.run()

print(missionNames[selectedProgram])
# Our Main Loop
while True:
    # Check if any buttons are pressed, and call the corresponding event handler
    buttonListener.process()
    # Debounce; Make sure the user has time to release the button
    sleep(0.1)
```

FollowWall (FollowWall.py):

```
def run(powerB, powerC, condition, exitSensor, val1, val2=0, stop=True):
    '''Controls each motor power separately, used for driving against the wall.
We arc against the wall and drive into it in order to have a reference point that
 is stationary and effective.'''
    robot.resetStartTime()
    robot.resetMotors()
    robot.safeMotorsOn(powerB, powerC)
    while not condition(exitSensor, val1, val2):
        pass
    if stop:
        robot.driveBase.stop()
```

Gear Lash (gearLash.py):

```
def run(direction, delay = 0.2, power = 5):
    '''Moves the motors at a very low power for a short time
    in order to minimize the slippage of the gears inside the motors of the robot
.'''
    robot.resetStartTime()
    robot.resetMotors()
    motorPower = power*direction*robot.motorDirection
    robot.safeMotorsOn(motorPower, motorPower)
    # Check if the time is equal to the amount set at the beginning of the functi
on
    while not condition(time, robot.timer, delay, 0):
        pass
    robot.driveBase.stop()
```

Gyro Turn (gyroTurn.py)

```
def run(targetAngle, motor = [1,-1], power = 80, adjust = 30):
    '''turns the robot using gyro sensor. Compares current angle with target angl
e
    then turns until target angle is reached. Since we use a very high power,
    we will turn to a value less than our desired angle, and turn slowly towards
our target angle.
    If we do exceed our target angle, we are still able to turn back towards it a
t a slow power.'''
    robot.resetAngle()
    # Moves the motor at a high power while the gyro sensor angle reading is less
 than the target angle subtracted from a specified value.
    while abs(robot.getAngle()) < targetAngle - adjust:
        # try:
            robot.driveBase.on(power*motor[0], power*motor[1])
            robot.checkAbort()
    # Prints the error between the target angle and the gyro sensor angle reading
 to the VSCode output window for debugging purposes.
    robot.driveBase.stop()
    robot.sleep(0.1)
    robot.debug("before error")
    robot.debug(targetAngle-abs(robot.getAngle()))
    # Checks whether the gyro sensor angle reading is less than or greater than t
he set target angle.
    if abs(robot.getAngle()) < targetAngle:
        # Moves the motors in one direction at a low power while the gyro sensor
angle reading is less than the target angle.
        while abs(robot.getAngle()) < targetAngle:
            robot.driveBase.on(5*motor[0], 5*motor[1])
            robot.checkAbort()
        robot.driveBase.stop()
    else:
        # Moves the motors in the opposite direction at a low power while the gyr
o sensor angle reading is less than the target angle.
        while abs(robot.getAngle()) > targetAngle:
            robot.driveBase.on(-5*motor[0], -5*motor[1])
            robot.checkAbort()
        robot.driveBase.stop()
    robot.debug(targetAngle-abs(robot.getAngle()))
```

PID Line Follower (pidLineFollower.py)

```
def run(targetPower, pidSensor, side, condition, exitSensor, val1,
val2 = 0, kp = constants.kp, stop = True):

    '''Follows the edge of the black line smoothly with the specified light
     sensor. Follow the edge at the speed, sensor, and side pecified by the user
in the first three parameters.
     The condition can be chosen as distance, light, or time, and the values on w
hich these exit can be modified.
     The second last parameter allows the user to change the constant that change
s the error to a value to add to the motors, which allows for the choice of sharp
er or
     smoother turns.'''

    robot.resetMotors()
    robot.resetStartTime()

    direction = 1 if targetPower > 0 else -1

    # Checking if the set exit condition is met
    while not condition(exitSensor, val1, val2):
        # Calculates the error by taking the current reflected light intensity an
d subtracting it by the optimal value (half black, half white)
        error = robot.calibratedValue(pidSensor) - constants.optimal
        # Calculates the pTerm by multiplying the error by a constant less than o
ne to lower the value.
        # It then multiplies by the side and direction, which can change which si
de or which direction it follows the line.
        pTerm = error * kp * side * direction
        robot.safeMotorsOn(
            # Add the pTerm to one motor power and subtract it from the other
            robot.motorDirection * targetPower + pTerm,
            robot.motorDirection * targetPower - pTerm
            )
    if stop:
        robot.driveBase.stop()
```

Square Up (squareUp.py)

```
def run(startPower, duration, side = 1):
    '''makes the robot perpendicular to the edge of the black line. Starts at hig
h power
     and moves each motor back then forward based on if the light sensor sees
     greater or less than 50 while decreasing the power, until the power is 0
     and the duration is reached. The rate at which we deccelerate at is
     calculated by dividing our duration by the start power.'''

    startTime = robot.timer.time()
    currentPower = startPower
    # The amount of power we decrease by per second
    powerDecay = startPower / duration

    while currentPower >= 0:
        elapsedTime = robot.timer.time() - startTime
        # Changes the current power by subtracting the initial power from the tim
e passed multiplied by the amount of power decreased per second.
        currentPower = startPower - elapsedTime * powerDecay
        currentPower = currentPower * side
        # Checks if the back left light sensor reading is greater than 50 (readin
g white)
        # If it is, move the motor until the light sensor reads black.
        if robot.calibratedValue(robot.BACK_LEFT) > 50:
            robot.motorB.on(currentPower)
        else:
            robot.motorB.on(-currentPower)

        # Checks if the back right light sensor reading is greater than 50 (readi
ng white)
        # If it is, move the motor until the light sensor reads black.
        if robot.calibratedValue(robot.BACK_RIGHT) > 50:
            robot.motorC.on(currentPower)
        else:
            robot.motorC.on(-currentPower)

    robot.driveBase.stop()
```

ZMove (zMove.py)

```
def run(targetPower, turn, turnDirection, condition, exitSensor, val1, val2 = 0,
    slowDownDistance = 10000, startPower = 30, endPower = 15, stop = True, kUp =
constants.kUp, kDown = constants.kDown):

    '''moves the robot forward or on an arc with error correction by finding
     the difference in the distance traveled by each motor multipled by the arc
     (variable called turn). It also uses acceleration and deceleration
     by adding a constant until the target power is reached, then the robot moves
     at the target power until the slow down distance is reached, then the power
decrases by
     subtracting the constant until the end power is reached, in which case the r
obot
     keeps moving until the end distance is reached.'''

    robot.resetMotors()
    robot.resetStartTime()
    direction = 1 if targetPower > 0 else -1

    targetPower = abs(targetPower)

    currentPower = startPower
    accelerating = True

    # Checking if the set exit condition is met
    while not condition(exitSensor, val1, val2):

        distanceB = abs(robot.motorB.position)
        distanceC = abs(robot.motorC.position)
        # Start decelerating if the distance is greater than the slow down distan
ce
        if distanceB > slowDownDistance and currentPower > endPower:
            currentPower = currentPower - constants.kDown
        elif accelerating:
            currentPower = currentPower + constants.kUp
            if currentPower >= targetPower:
                accelerating = False

        if turnDirection == 1:
            error = constants.kWiggle * (distanceB - distanceC * turn)
        else:
            error = constants.kWiggle * (distanceB * turn - distanceC)
        powerB = currentPower - error
        powerC = currentPower + error
        robot.safeMotorsOn(powerB * robot.motorDirection * direction, powerC * ro
bot.motorDirection * direction)

    if stop:
        robot.driveBase.stop()
```

ZPivot (zPivot.py)

```
def run(targetPower, motor, condition, exitSensor, val1, val2 = 0,
    slowDownDistance = 800, startPower = 10, endPower = 10, stop = True):

    '''turns the robot by setting the specified motor (one that is not turning)to
 0.
     Based on the inputed direction the other motor moves backwards or forwards
     by multiplying the power by the direction. It also uses acceleration and dec
eleration
     by adding a constant until the target power is reached, then the robot moves
     at the target power until the slow down distance is reached, then the power
decrases by
     subtracting the constant until the end power is reached, in which case the r
obot
     keeps moving until the end distance is reached.'''

    robot.resetMotors()
    robot.resetStartTime()
    direction = 1 if targetPower > 0 else -1

    targetPower = abs(targetPower)

    currentPower = startPower
    accelerating = True

    while not condition(exitSensor, val1, val2):

        distance = abs(motor.position)

        if distance > slowDownDistance and currentPower > endPower:
            currentPower = currentPower - constants.kDown
        elif accelerating:
            currentPower = currentPower + constants.kUp
            if currentPower >= targetPower:
                accelerating = False

        power = currentPower

        motor.on(power * robot.motorDirection * direction)

    if stop:
        robot.driveBase.stop()
```

Constants (constants.py)

```
# proportional coefficient for line following
kp = 0.08
# integral coefficient for line following
ki = 0.0
# deriviative coeficcient for line following
kd = 0.0
# optimal light reading (half black, half white)
optimal = 50

# distance error to motor power scaling coefficient for zMove
kWiggle = 0.1
# acceleration coefficient
kUp = 1.0
# deceleration coefficient
kDown = 1.0
```

Exit Conditions (exitConditions.py)

```
def distance(motor, value, unused):
    '''returns True if the distance travelled exceeds the set value'''
    #print(motor.position, file = stderr)
    robot.checkAbort()
    return abs(motor.position) > value

def time(timer, value, unused):
    '''returns True if the time elapsed since timerReset exceeds the set value'''
    #print(timer.time() - robot.startTime, file = stderr)
    robot.checkAbort()
    return timer.time() - robot.startTime > value

def light(sensor, value1, value2):
    '''returns True if the light reading is within the set threshold'''
    robot.checkAbort()
    return robot.calibratedValue(sensor) >=value1 and robot.calibratedValue(senso
r) <=value2
```

Robot (robot.py)

```python
# Prevents collision of the time function
# We import time from our exitCondition module as timeLocal
from util.exitConditions import time as timeLocal

# Object that controls the movement of both motors
driveBase = MoveTank(OUTPUT_B, OUTPUT_C)

# When motorDirection is 1, drive forward, if it is -1, drive backward
motorDirection = 1

motorB = LargeMotor(OUTPUT_B)
motorC = LargeMotor(OUTPUT_C)
motorD = LargeMotor(OUTPUT_D)

soundGenerator = Sound()

# Re-export the timer from the time module
timer = time
startTime = 0

FRONT = ColorSensor(INPUT_4)
BACK_LEFT = ColorSensor(INPUT_3)
BACK_RIGHT = ColorSensor(INPUT_2)
GYRO = GyroSensor(INPUT_1)

# List of light sensor inputs
sensors = [
    FRONT,
    BACK_LEFT,
    BACK_RIGHT
]

calibrationMin = []
calibrationMax = []

# Records the current angle when resetAngle is called
# We use it to make Gyro movements relative to the startAngle
startAngle = 0

def resetAngle():
    '''Reset the startAngle so the angle returned by getAngle is relative to the
beginning of the turn'''
    global startAngle
    startAngle = GYRO.angle
```

```
def getAngle():
    '''Gets the current angle from when the last resetAngle was called'''
    global startAngle
    return GYRO.angle - startAngle

def resetMotors():
    '''Resets the rotational sensors to 0'''
    motorB.reset()
    motorC.reset()

def resetStartTime():
    '''Resets the timer to 0'''
    global startTime
    startTime = timer.time()

def getTime():
    '''Gets the time from when the last resetStartTime was called'''
    global startTime
    return timer.time() - startTime

def debug(text):
    '''Print text to the VSCode output window'''
    print(text, file=stderr)

def initCalibration():
    '''Reads the values stored in the calibration file'''
    global calibrationMin
    global calibrationMax
    calibrationMin, calibrationMax = calibration.readValues()

def calibratedValue(sensor):
    '''scales current light sensor readings to values between 0 and 100
    and clamps the value between 0 and 100'''
    index = getIndexFromSensor(sensor)
    rawValue = sensors[index].reflected_light_intensity
    calibratedValue = 100.0 * (rawValue -
 calibrationMin[index]) / (calibrationMax[index] - calibrationMin[index])
    return min(100.0, max(0.0, calibratedValue))

def printSensors():
    '''prints the calibrated values for each sensor (used for debugging)'''
    debug("BACK_LEFT")
    debug(calibratedValue(BACK_LEFT))
    debug("BACK_RIGHT")
    debug(calibratedValue(BACK_RIGHT))
    debug("FRONT")
    debug(calibratedValue(FRONT))
```

```python
def getIndexFromSensor(sensor):
    '''finds the index of the light sensor object'''
    for i in range (len(sensors)):
        if sensor == sensors[i]:
            return i
    return -1

def init():
    '''uses these functions in main program to use light sensors and test gyro dr
ift'''
    initCalibration()
    testGyroDrift()

def afterMission():
    '''stops drive motors and releases tension of attachment motor'''
    driveBase.stop()
    motorD.off(brake=False)

def safeMotorsOn(powerB, powerC):
    '''ensures that motor powers don't exceed -
100 and 100, if they do, scales down values'''
    driveBase.on(min(100,max(powerB, -100)), min(100,max(powerC, -100)))

def testGyroDrift():
    '''checks for drift by comparing currAngle to prevAngle one second ago,
    if gyro is drifting, robot beeps and you unplug and replug sensor to clear dr
ift'''
    firstRead = getAngle()
    print("checking for gyro drift")
    sleep(1)

    if abs(firstRead - getAngle()) > 1:
        print("gyro is drifting")
        soundGenerator.beep()
        raise Exception ("gyro is drifting")
        sleep(2)
    print ("No drift")

def beep():
    '''beeps'''
    soundGenerator.beep()


def sleep(seconds):
    '''special pause that exits when up button is pressed'''
    resetStartTime()
    while not timeLocal(timer, seconds, 0):
        pass
```

```
def checkAbort():
    '''check if the up button is pressed, if it is, raise an exception and exit t
he program'''
    if Button().up:
        raise Exception("button up was pressed")
```

Show Gyro (showGyro.py)

```python
def run():
    currAngle = robot.GYRO.angle
    lastAngle = currAngle
    robot.resetStartTime()
    timeStep = 3
    while robot.getTime() <= 10:
        currAngle = robot.GYRO.angle
        diff = currAngle - lastAngle
        driftRate = diff/timeStep
        print(currAngle - lastAngle)
        lastAngle = currAngle
        robot.sleep(timeStep)
```

Calibration (calibration.py)

```
def run():
    '''runs robot for 7 seconds while storing the highest and lowest light sensor
readings,
    then store these in a minimum and maximum value file'''

    robot.driveBase.on(-15, -15)
    startTime = robot.timer.time()

    # arrays are empty because we have not read light values yet
    lightMin = []
    lightMax = []

    # sets min and max to unrealistic values so they are overwritten on the first
reading
    for i in range(len(robot.sensors)):
        lightMin.append(1000)
        lightMax.append(-1000)

    while robot.timer.time() - startTime < 7:
        # read values for all light sensors
         for i in range(len(robot.sensors)):
            lightReading = robot.sensors[i].reflected_light_intensity

            # if current values are lower than previous min, current value is now
the min
            if lightReading < lightMin[i]:
                lightMin[i] = lightReading

            # if current values are higher than previous max, current value is
now the max
            if lightReading > lightMax[i]:
                lightMax[i] = lightReading

    robot.driveBase.stop()
    # open the file and write the min values
    file = open("util/minValues.txt", "w+")
    for i in range(len(robot.sensors)):
        file.write(str(lightMin[i]) + "\n")
    file.close()

    # open the file and write the max values
    file = open("util/maxValues.txt", "w+")
    for i in range(len(robot.sensors)):
        file.write(str(lightMax[i]) + "\n")
    file.close()

    # initializes the min and max lists
    robot.initCalibration()

def readValues():
    '''Reads the file line by line, then strips the enter from the String,
    then converts values to a float and adds them to min array'''
    file = open("util/minValues.txt", "r")
    min = [float(line.rstrip('\n')) for line in file]
    file.close()

    file = open("util/maxValues.txt", "r")
    max = [float(line.rstrip('\n')) for line in file]
    file.close()

    return min, max
```

Crane (craney.py)

```
def run():
    zMove.run(-80, 1, 1, distance, robot.motorB, 800)
    # gyroTurn.run(20, direction = -1, adjust = 10)
    gyroTurn.run(20, [-1, 1], power = 30, adjust = 10)
    zMove.run(-30, 1, 1, distance, robot.motorB, 550)
    # gyroTurn.run(18, 50, 1, adjust = 5)
    gyroTurn.run(18, [1, -1], 30, adjust = 5)
    #drives forward and uses gyro sensor to adjust to be straight on with the box
    zMove.run(-10, 1, 1, distance, robot.motorB, 780)
    # pushes block forward and lowers crane arm
    zMove.run(-5, 1, 1, distance, robot.motorB, 200)
    zMove.run(100, 1, 1, distance, robot.motorB, 750, startPower = 50)
    # gyroTurn.run(90, 60, -1, adjust = 15)
    gyroTurn.run(90, [-1, 1], 60, adjust = 15)
    zMove.run(100, 1, 1, distance, robot.motorB, 1750, startPower = 60)
    #drives back and turns into base
```

Traffic Jam (trafficJam.py)

```
soundGenerator = Sound()


def run():
   #Traffic Jam Mission
   zMove.run(-50, 1, 1, distance, robot.motorB, 550)
   zMove.run(-25, 1, 1, light, robot.BACK_RIGHT, 2, 25)
   sleep(0.25)
   zMove.run(-50, 1, 1, distance, robot.motorB, 400)
   robot.motorD.on_for_degrees(speed=30, degrees=200)
   zPivot.run(-70, robot.motorB, time, robot.timer, 1.5)
   #Swing Mission
   zMove.run(50, 1, 1, light, robot.FRONT, 80, 100)
   zMove.run(100, 1, 1, distance, robot.motorB, 60)
   zPivot.run(25, robot.motorB, light, robot.BACK_RIGHT, 70, 100)
   zMove.run(-20, 1, 1, distance, robot.motorB, 50)
   zPivot.run(25, robot.motorB, light, robot.BACK_RIGHT, 0, 20)
   zPivot.run(25, robot.motorB, light, robot.BACK_RIGHT, 70, 100)
   # zMove.run(-20, 1, 1, light, robot.BACK_RIGHT, 0, 20)
   # zMove.run(-20, 1, 1, light, robot.BACK_RIGHT, 70, 100)
   #zPivot.run(25, robot.motorB, light, robot.BACK_RIGHT, 70, 100)
   pidLineFollower.run(-8, robot.BACK_RIGHT, 1, distance, robot.motorB, 300, kp =
0.11)
   pidLineFollower.run(-15, robot.BACK_RIGHT, 1, distance, robot.motorB, 675 )
   soundGenerator.beep()
   pidLineFollower.run(-15, robot.BACK_RIGHT, 1, light, robot.BACK_LEFT, 0, 20)
   soundGenerator.beep()
   sleep(0.25)
   pidLineFollower.run(-15, robot.BACK_RIGHT, 1, distance, robot.motorB, 325)
   zMove.run(-15, 1, 1, light, robot.BACK_LEFT, 75, 100)
   soundGenerator.beep()
   sleep(0.25)
   pidLineFollower.run(-25, robot.BACK_RIGHT, 1, distance, robot.motorB, 950)
   #Safety Factor Mission
   zMove.run(60, 1, 1, distance, robot.motorB, 550)
   #gyroTurn.run(90, 50, 1, 10)
```

```
    gyroTurn.run(90, [1, -1], 50, 10)
    zMove.run(-80, 1, 1, distance, robot.motorB, 450)
    zMove.run(-30, 1, 1, light, robot.FRONT, 0, 15)
    zMove.run(40, 1, 1, distance, robot.motorB, 125)
    soundGenerator.beep()
    sleep(0.25)
    #gyroTurn.run(90, 40, 1, 40)
    gyroTurn.run(90, [1, -1], 40, 40)
    sleep(0.25)
    zMove.run(35, 1, 1, distance, robot.motorB, 210)
    zMove.run(35, 1, 1, time, robot.timer, 3.25, startPower=35)
    #Elevator Mission
    zMove.run(-60, 1, 1, distance, robot.motorB, 200)
    robot.motorD.on_for_degrees(speed=45, degrees=460)
    #gyroTurn.run(90, 60, -1, 35)
    gyroTurn.run(90, [-1, 1], 60, 35)
    zMove.run(-40, 1, 1, distance, robot.motorB, 15)
    robot.motorD.on_for_degrees(speed=45, degrees=-525)
    zMove.run(60, 1, 1, distance, robot.motorB, 500)
    #gyroTurn.run(90, 55, 1, 25)
    gyroTurn.run(90, [1, -1], 55, 25)
    zMove.run(-100, 1, 1, distance, robot.motorB, 5000, startPower=90, kUp=150,
kDown=0)
```

Block Delivery (blockDelivery1.py)

```python
def run():
    zMove.run(80, 1, 1, distance, robot.motorB, 1250, 0, 950)
    zMove.run(-80, 1, 1, distance, robot.motorB, 350)
    # gyroTurn.run(90, 60, -1)
    gyroTurn.run(90, [-1, 51], 60)
    zMove.run(-90, 1, 1, distance, robot.motorB, 1000)
```

Red Circle and Bridge (redCircle.py)

```
def run():
    zMove.run(100, 1, 1, distance, robot.motorB, 1850, slowDownDistance=1500, sto
p=False)
    robot.soundGenerator.beep()
    zMove.run(100, 1, 1, light, robot.FRONT, 85, 100)
    robot.motorD.on_for_degrees(speed=-15, degrees=300)
    zMove.run(100, 1, 1, light, robot.BACK_LEFT, 85, 100)
    soundGenerator = Sound()
    soundGenerator.beep()
    sleep(0.5)
    squareUp.run(20, 2.5, 1)
    zMove.run(-50, 1, 1, distance, robot.motorB, 150)
    #gyroTurn.run(90, 80, 1)
    gyroTurn.run(90, [1, -1], 40, adjust = 15)
    robot.motorD.on_for_degrees(speed=50, degrees=625)
    #gyroTurn.run(180, 80, 1)
    gyroTurn.run(180, [1, -1], 30, 40)
    zMove.run(-100, 1, 1, distance, robot.motorB, 2040)
    sleep(1)
    zPivot.run(50,robot.motorB, distance, robot.motorB, 200, robot.motorB)
```