Spockbots

Release 1.0

Spockbots

Nov 12, 2019

CONTENTS

CHAPTER

ONE

SUMMARY

1.1 Summary of Robot Features

1.1.1 Mechanical Design

Durability - Robot designed to maintain structural integrity and have the ability to withstand rigors of competition

- We build a box robot that contaons a frame around it that is stable
- The robot is durable and can withstand drops
- The cables are out of the way

Mechanical - Efficiency Robot designed to be easy to repair, modify, and be handled by technicians

- The robot does not need to be repaired a lot due to its sturdy design

Mechanization - Robot mechanisms designed to move or act with appropriate speed, strength and accuracy for intended tasks (propulsion and execution)

- The robot has 63.6 mm wheel which allows to go fast, but also precise.
- The robot has 2 medium motors built in that allow for easy attachments.

1.1.2 Programming

Programming Quality Programs are appropriate for the intended purpose and should achieve consistent results, assuming no mechanical faults

- We tested several runs and they were 100% reliable given our goals for the mission.
- We use functions to describe the runs so they can be quickly developed

Programming Efficiency Programs are modular, streamlined, and understandable

- We use classes for Colorsensors and motors
- We use a lot of functions and methods
- We use seperate programs for runs
- The code is very modular
- The code is documented
- We use a python document generator to create the documentation.

Automation/Navigation - Robot designed to move or act as intended using mechanical and/or sensor feedback (with minimal reliance on driver intervention and/or program timing)

- We use 3 color sensors
- We use 1 gyro sensor
- We use line following
- We programmed our own left, right, based on angle rotations for the motor
- We use Gyro sensor left, right, forward
- We use color and reflective mode to identify markers to react
- We make sure the robot is not running too fast if we run into the crane
- We make sure the robot is fast when we turn over the lift
- We can use our line following forwards and backwards, by inverting the motors
- All reflective light values are independently calibrated and mapper between 0 to 100

1.1.3 Strategy and Innovation

Design Process Developed and explained improvement cycles where alternatives were considered and narrowed, selections tested, designs improved (applies to programming as well as mechanical design)

- Programming - we focused on python and explored if we can replicate our library which we previously developed in

Mindtsorm. This was new to us and we were not sure if we can do the missions in Pythoin. We found out we can.

- Mechanical design - we designed a number of robots. we found that its bets to place the brick over the wheels as otherwise the wheels slip
- We tried mechanical attachments that showed they were to heavy and the robot spipped. We even tried wider wheels but they also slipped.
- We made all atatchments very light and relatively small.
- The attachments are custom designed for the missions
- We can drive backwards to drive up the ramp. The light sensor that is than facing the ramp is high up so its possible to go up the ramp.

Mission Strategy Clearly defined and described the team's game strategy

- Our main goal was to learn python and test various functionssuch as line following, gyro, movements
- We picked missions that were easy to do but give us some number of points

TODO: list of missions with their points

Innovation - Team identifies their sources of inspiration and creates new, unique, or unexpected feature(s) (e.g. designs, programs, strategies or applications) that are beneficial in performing the specified tasks

- The bets feature is the calibration of the light sensor that drives ove a line and stores the minimum black and maximum white value. Than we use a special sensor value function to always return values mapper between 0 and 100.
- We reimplemented turn, left and right with motor angle measuremnts
- We have implemented line following
- We have implemented a gyro go straight
- All of the modular code is reusable and could be used by others.

CHAPTER

TWO

PYTHON

We show how simple it is to program in python with some examples.

2.1 Variables

Variables allow storing of data values. This is the same as the EV3 GUI variable

Example:

```
x = 5
y = 10
```

2.2 Lists

This is the same as the array in EV3 GUI.

Lists store multiple data values:

```
vector = [x, y]
vector = [5, 10]
```

2.3 Functions

A function is a block of code which only runs when it is called. It may return a value and can have parameters. This is the same as a myblock, but easier to write and modify:

```
def add(a,b):
    return a + b

def PRINT(message):
    print("Message", message)
```

2.4 Classes

With classes we can group functions and variables conveniently into an object. an object is just like a variable that uses the class as template. we can call all variables and functions on this object. Functions in a class are called methods. A special method is __init__ which is called once when we declare an object from the template:

```
class Person:

    def __init__(name, age, weight):
        self.name = name
        self.age = age
        self.height = height

    def grow(amount):
        self.height = height + amount

    def how_tall():
        return self.height

seric = Person("Seric", 14, 150)
seric.grow(1)
print (seric.height)      #  151
print (seric.how_tall())  #  151
```

2.5 Conditions

Conditions allow is to react if a value is tru or false. It is the same as in EV3 GUI but easier to write:

```
if seric.height > 180:
    print("He is tall")
elif seric.height < 180:
    print("He is still growing")
else:
    print("he is exactly 180cm")
```

2.6 Loops

We used while and for loops the repeat an indented block of code. While loops can also loop over elements in a list easily.

2.6.1 Loop forever

```
while True:
    print("I loop forever")
```

2.6.2 Loop with condition

```
counter = 1
while counter <= 3:
    print (counter)
    counter = counter + 1

# 1
# 2
# 3
```

2.6.3 Loop with break

```
counter = 1
while True:
    print (counter)
    counter = counter + 1
    if counter > 3:
       break # break leaves the loop
# 1
# 2
# 3
```

2.6.4 Loop through a list

for counter in [1,2,3]: print (counter)

1 # 2 # 3

2.7 Import

When we create code in separate files they can be made known within a program while importing the functions, classes, or variables. This allows us to organize the code while grouping topical code into a file.

```
from spockbots.motor import SpockbotsMotor
from time import sleep
```

2.8 Program

A program can be executed in a terminal on teh EV3 brick. It must be executable. Let us assume the following core it in the file *run_led.py*. we make it executable with:

```
chmod a+x run_led.py
```

Here an example:

```
#!/usr/bin/env pybricks-micropython

from spockbots.output import flash
import time


def run_led():
    """
    Flashes the LEDs on the brick
    """

    flash()


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

The first line tells us to use python to run the program.

The if __name__ line tells us to run the next lines (e.g. the function) as functions are not run when we define them.

CHAPTER

THREE

SPOCKBOTS ROBOT

3.1 Design

3.1.1 Goals for using Python

Our main goal this year was to learn Python.

Previously we used Mindstorms GUI and developed a sophisticated library with many myblocks.

Questions

1. Can we convert this library into python?
2. Would python easier than the mindstrom GUI?
3. Would it be easier to define missions with Python?
4. Would the robot perform well with the Python program?
5. What general reasons are there for and against Python or Mindstorm?

Previous Mindstorm GUI programs

- We had an library developed in Mindstorm GUI with many myBlocks that we used previously to develop code. Can they be redeveloped in Python?
- We had some issues with using Bluetooth between Mac and the robot in Mindstorm. Is this improved in Python
- In contrast to Windows, the GUI on Mac seems slower. Is the development in python faster?
- We often ran out of screen space as the programs were long. Does using python help?
- We had some issues with the Gyro and light sensors in the mindstor GUI. Do these issues occur also in python?

Table 1: Python and Mindtsorm GUI comparison

Task	Winner	Mindstorm	Python
Bluetooth on Mac	Python	Connection could often not be done easily	No issues
MyBlocks definition	Python	Complicated to define, if you make an error in the parameters one needs to start over.	With functions easy to define and correct
Gyro Drift	Neither	Gyro needs to be plugged in and out to avoid drift	Gyro needs to be plugged in and out to avoid drift
Gyro Reset	Neither	Reset requires a time delay	Reset requires a time delay.
Color Sensor No value	Python	Sometimes the color sensor has no value. We did not have a fix for that.	Easy to fix in python while using the previous value. Comes back quickly
Color Sensor reset with mor than one sensor	Python	Not available	We implemented this so that all color sensors return always values between 0 -100, this helps line following.
Editor	Equal	GUI is intuitive to understand but has issues with myblock. Limited space, myblocks do not have names under them but just icons	Visual code is easy but a bit mor complex to understand, but once you know it writing programs is easy. Upload and run code with F5 is convenient
Interrupting a wrong program with the backspace button	Mindstorm	This can easily be done in mindstorm	This does not work in python. We need to add code for that
Younger Kids	Mindstorm	Mindstorm GUI is intuitive, but best for small programs	Python is a more difficult to learn by younger kids
Older Kids	Python	Mindostom GUI becomes cumbersome when we develop more complex programs	Python is easy to learn by older kids

Overall winner:

Python

What should be added to Python:

- Color Sensor calibration and value code we developed that returns values between 0-100 for all sensors
- A test to see if the gyro is drifting
- A solution to avoid the unplugging of the gyro sensor

Observations and answers:

Gyro:

1. The gyro needs to be plugged in and out at the beginning to avoid drifting.

   This could not be solved in Python but we implemented a function that detects better if the Gyro drifts.

2. We need to have a wait till the Gyro is calm

   We reimplemented this not with time delay, but a counter to see if the angle has changed. This could also be implemented in mindstorm GUI

Light Sensor:

1. sometimes the light sensor did not return a result

2. We developed a calibration that drove over a line to calibrate our sensors. However, the reset block is only designed to use one Gyro and not 2

Fig. 1: Robot 1a. Crane setup does not have to be precise

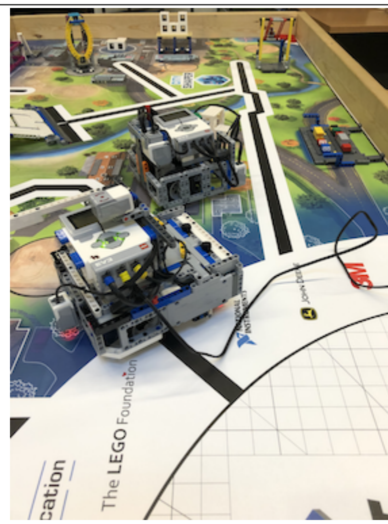

Fig. 2: Robot 2a. Crane setup still works well when the peg points to the line.

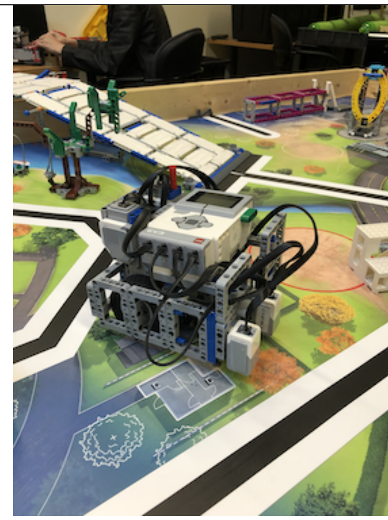

Fig. 3: Robot 3. The robot is too likght as there is no weight on the back wjeels

3.2 Menu

We named it *0_menu.py* so it shows up on the top in the brick program:

```
    Crane
>>> Swing
    Calibrate
    ....
```

Displays a menu in which we muve with the UP DOWN keys up and down. We leave with the left key and select a program with the right key.

3.3 City Runs

3.3.1 run.red_circle

run.red_circle.**run_red_circle**()

TBD

3.3.2 run.tan_circle

run.tan_circle.**run_tan_circle**()

TBD

3.3.3 run.black_circle

`run.black_circle.run_black_circle()`
TBD

3.3.4 run.crane

`run.crane.run_crane()`
TBD

```
#!/usr/bin/env pybricks-micropython

from spockbots.motor import SpockbotsMotor
from time import sleep


def run_crane():
    """
    TBD
    """
    robot = SpockbotsMotor()
    robot.debug = True

    robot.setup()
    robot.color.read()

    print(robot)

    """
    robot.forward(50, 10)
    robot.turn(25, 45)
    robot.forward(50, 30)

    robot.turn(25, -45)

    robot.gotowhite(25, 3)
    robot.gotoblack(10, 3)
    robot.gotowhite(10, 3)

    #robot.forward(5, 2)
    #robot.forward(-20, 20)
    #robot.right(20, 45)
    #robot.forward(-75, 60)
    """

    dt = 0.0

    robot.forward(50, 20)

    robot.gotowhite(25, 3)
    robot.turntoblack(25, direction="right", port=3)

    robot.forward(50, 5)

    robot.turntowhite(15, direction="left", port=2)
```

(continues on next page)

(continued from previous page)

```
    robot.followline(speed=10, distance=13,
                     port=2, right=True,
                     delta=-35, factor=0.4)

    robot.forward(50, -5)

    robot.gotowhite(10, 3)
    robot.gotoblack(10, 3)
    robot.gotowhite(10, 3)

    robot.forward(2, 4)
    robot.forward(10, 1)

    # sleep(0.2)

    # back to base

    robot.forward(5, -5)  # backup slowly
    robot.forward(75, -20)
    robot.turn(25, 45)
    robot.forward(75, -30)
    robot.turn(25, 45)
    robot.forward(75, -20)


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

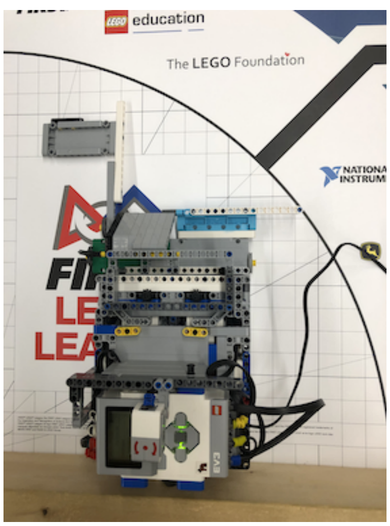

Fig. 4: Crane 1. Crane setup does not have to be precise

Fig. 5: Crane 2. Crane setup still works well when the peg points to the line.

Fig. 6: Crane 4. Successful placement

Reliability:

Setup The setup for the crane mission is important, but it does not have to be precise. There is a balck line that we can allign the blue peg with.

Run We use the balck line to align the robot so that the attachment is working well.

Mechanical We have an attachment designed that pushes the block and activates the blue levers at the righ time. The drive must not be fast into the crane. The crane block must not swing wne we start.

Mission Order To avoid the swinging of the line (by other teams moving the table), this is our first mission.

3.3.5 run.swing

`run.swing.run_swing()`
TBD

```
#!/usr/bin/env pybricks-micropython

from spockbots.motor import SpockbotsMotor
from time import sleep


def run_crane():
    """
    TBD
    """
    robot = SpockbotsMotor()
    robot.debug = True

    robot.setup()
    robot.color.read()

    print(robot)

    """
    robot.forward(50, 10)
    robot.turn(25, 45)
    robot.forward(50, 30)

    robot.turn(25, -45)

    robot.gotowhite(25, 3)
    robot.gotoblack(10, 3)
    robot.gotowhite(10, 3)

    #robot.forward(5, 2)
    #robot.forward(-20, 20)
    #robot.right(20, 45)
    #robot.forward(-75, 60)
    """

    dt = 0.0

    robot.forward(50, 20)

    robot.gotowhite(25, 3)
    robot.turntoblack(25, direction="right", port=3)

    robot.forward(50, 5)

    robot.turntowhite(15, direction="left", port=2)

    robot.followline(speed=10, distance=13,
```

(continues on next page)

(continued from previous page)

```
                      port=2, right=True,
                      delta=-35, factor=0.4)

    robot.forward(50, -5)

    robot.gotowhite(10, 3)
    robot.gotoblack(10, 3)
    robot.gotowhite(10, 3)

    robot.forward(2, 4)
    robot.forward(10, 1)

    # sleep(0.2)

    # back to base

    robot.forward(5, -5)   # backup slowly
    robot.forward(75, -20)
    robot.turn(25, 45)
    robot.forward(75, -30)
    robot.turn(25, 45)
    robot.forward(75, -20)


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

Reliability:

Setup To help the setup we are using a jig.

Run We use the balck lines and that allow us to be more precise.

Mechanical We have an attachment designed that pushes the house block, a lever that starts the swing, a lever that allows us to flip a blue stand and a lever tu turn the elevator.

Mission Order THis mission would be best to be strarte at the begining as it is mor complex to set up the the crane, but the crane may swing so we decided to run the swing first.

3.3.6 run.led

`run.led.`**`run_led`**`()`

TBD

3.3.7 run.turn_to_black module

`run.turn_to_black.`**`run_turn_to_black`**`()`

TBD

3.3.8 run.calibrate

`run.calibrate.`**`run_calibrate`**`()`

Run the calibration

Returns a file called calibrate.txt that contains the minimum black and the maximum white value for the sensors

3.3.9 run.check

`run.check.run_check()`

Checks the robot by driving the large and medium motors and flashing the color sensors

Order:

- Large Motor left
- Large Motor left
- Medium Motor left
- Medium Motor left
- Color Sensor left
- Color Sensor right
- Color Sensor back

3.4 Spockbots API

3.4.1 spockbots.output

`spockbots.output.PRINT(*args, x=None, y=None)`

prints message on screen at x and y and on the console. if x and y are missing prints on next position on lcd screen this message prints test messages.

The sceensize is maximum x=177, y=127)

Parameters

- **args** – multible strings to be printed in between them
- **x** – x value
- **y** – y value

`spockbots.output.beep()`

The robot will make a beep

`spockbots.output.clear()`

clears display

`spockbots.output.flash(colors=['RED', 'BLACK', 'RED', 'BLACK', 'GREEN'], delay=0.1)`

The robot will flash the LEDs and beep twice

`spockbots.output.led(color)`

changes color of led light

Parameters

- **color** – light color
- **brightness** – light brightness

Returns

`spockbots.output.readfile(name)`
Reads the file with the name and returns it as a string.

Parameters **name** – The file name

Returns The data in teh file as string

`spockbots.output.sound(pitch=1500, duration=300)`
plays a sound

Parameters

- **pitch** – sound pitch
- **duration** – how long the sound plays

Returns

`spockbots.output.voltage()`
prints voltage of battery

`spockbots.output.writefile(name, msg)`
Writes a new file with the name. If it exists the old file will be deleted.

Parameters

- **name** – The name of the file
- **msg** – The message to be placed in the file

Returns

3.4.2 spockbots.check

`spockbots.check.check(speed=100, angle=360)`
do a robot check by

a) turning on the large motors one at a time

b) turning on the medium motors one at a time

c) turning on the light sensors one at a time

Parameters

- **speed** –
- **angle** –

Returns

3.4.3 spockbots.motor

`class spockbots.motor.SpockbotsMotor(direction=None)`
Bases: `object`

`aligntoblack(speed, port_left, port_right, black=10)`
aligns with black line while driving each motor.

Parameters

- **speed** – speed of robot
- **port_left** – port of left color sensor

- **port_right** – port of right color sensor
- **black** – value of black

aligntowhite (*speed*, *port_left*, *port_right*, *white=80*)
aligns with white line while driving each motor.

Parameters

- **speed** – speed of robot
- **port_left** – port of left color sensor
- **port_right** – port of right color sensor
- **white** – value of white

angle_to_distance (*angle*)
calculation to return the distance in cm given an angle.

Parameters **angle** – the angle

Returns distance in cm for turning an angle

beep ()
robot will beep.

calibrate (*speed, distance=15, ports=[2, 3, 4], direction='front'*)
calibrates color sensors by driving over black and white line.

Parameters

- **speed** – speed of robot
- **distance** – distance that robot travels
- **ports** – ports of color sensors
- **direction** – direction of calibration

check_kill_button ()
This will stop all motors and finish the program. It can be used in the programs to check if the program should be finished early du to an error in the runs.

distance_to_angle (*distance*)
calculation to convert the distance from cm into angle.

Parameters **distance** – The distance in cm

Returns The degrees traveled for the given distance

distance_to_rotation (*distance*)
calculation to convert the distance from cm into rotations.

Parameters **distance** – The distance in cm

Returns The rotations to be traveled for the given distance

followline (*speed=25*, *distance=None*, *t=None*, *port=3*, *right=True*, *stop_color_sensor=None*, *stop_values=None*, *stop_color_mode=None*, *delta=-35*, *factor=0.4*)
follows line for either a distance or for time.

Parameters

- **speed** – speed of robot
- **distance** – distance that robot follows line
- **t** – time that robot follows line for

- **port** – port of color sensor
- **right** – whether the robot is following the right or left side of line
- **black** – black value
- **white** – white value
- **delta** – adjustment value to convert from color sensor values (0 to 100) to motor steering (-100 to 100)
- **factor** – factor of adjustment, controls smoothness

followline_pid(*debug=False*, *distance=None*, *t=None*, *right=True*, *stop_color_sensor=None*, *stop_values=None*, *stop_color_mode=None*, *port=3*, *speed=25*, *black=0*, *white=100*, *kp=0.3*, *ki=0.0*, *kd=0.0*)

forward(*speed*, *distance*, *brake=None*)
the robot drives forward for a given distance.

Parameters

- **speed** – speed of robot
- **distance** – distance that robot goes forward (in cm)
- **brake** – one of the values brake, hold, coast

gotoblack(*speed*, *port*, *black=10*)
robot moves to the black line while using the sensor on the given port.

Parameters

- **speed** – speed of robot
- **port** – port of color sensor
- **black** – value of black

gotocolor(*speed, port, colors=[0]*)
robot moves to the black line while using the sensor on the given port.

Parameters

- **speed** – speed of robot
- **port** – port of color sensor
- **black** – value of black

gotowhite(*speed*, *port*, *white=90*)
robot moves to the white line while using the sensor on the given port.

Parameters

- **speed** – speed of robot
- **port** – port of color sensor
- **white** – value of white

on(*speed*, *steering=0*)
turns the large motors on while using steering.

Parameters

- **speed** – the speed of the robot
- **steering** – an angle for the steering

on_forever (*speed_left*, *speed_right*)
turns motors on with left and right speed.

Parameters

- **speed_left** – speed of left motor
- **speed_right** – speed of the right motor

reset ()
resets the angle in the large motors to 0.

setup (*direction=None*)
setup the direction, the motors, and the tank with the appropriate direction.

Parameters **direction** – if the direction is 'forward' the robot moves forward, otherwise backwards.

Returns left, right motors and tank

still ()
waits till the motors are no longer turning.

stop (*brake=None*)
stops all motors on all different drive modes.

Parameters **brake** – None, brake, coast, hold

turn (*speed*, *angle*)
takes the radius of the robot and dives on it for a distance based on the angle.

Parameters

- **speed** – speed of turn
- **angle** – angle of turn

turntoblack (*speed*, *direction='left'*, *port=3*, *black=10*)
turns the robot to the black line.

Parameters

- **speed** – speed of turn
- **direction** – left or right
- **port** – port of color sensor
- **black** – value of black

turntowhite (*speed*, *direction='left'*, *port=3*, *white=80*)
turns the robot to the white line.

Parameters

- **speed** – speed of turn
- **direction** – left or right
- **port** – port of color sensor
- **white** – value of white

value (*port*)
return the reflective color sensor value.

Parameters **port** – the port number of the color sensor

Returns the reflective color value

3.4.4 spockbots.gyro

class spockbots.gyro.**SpockbotsGyro**(*robot*, *port=1*)

Bases: object

improved gyro class

angle()

Gets the angle

Returns The angle in degrees

drift()

tests if robot drifts and waits until its still

forward(*speed=25*, *distance=None*, *t=None*, *port=1*, *delta=-35*, *factor=0.4*)

Moves forward

Parameters

- **speed** – The speed
- **distance** – If set the distance to travle
- **t** – If set the time to travel
- **port** – The port number of the Gyro sensor
- **delta** – controlling the smoothness of the line
- **factor** – controlling the smoothness of the line

Returns

left(*speed=25*, *degrees=90*, *offset=0*)

The robot turns left with the given number of degrees

Parameters

- **speed** – The speed
- **degrees** – The degrees
- **offset** –

Returns

reset()

safely resets the gyro

right(*speed=25*, *degrees=90*, *offset=0*)

The robot turns right with the given number of degrees

Parameters

- **speed** – The speed
- **degrees** – The degrees
- **offset** –

Returns

status (*count=10*)
: tests count times if robot is still and returns if its still or drifts :param count: number of times tested if its still :return: still,drift which are true/false

still ()
: tests if robot dosent move :return: True if robot dosent move

turn (*speed=25*, *degrees=90*)
: uses gyro to turn positive to right negative to left :param speed: speed it turns at :param degrees: degrees it turns :return:

zero ()
: set the gyro angle to 0 :return:

3.4.5 spockbots.colorsensor

class `spockbots.colorsensor.`**SpockbotsColorSensor** (*port=3*)
Bases: `object`

defines a Colorsensor with values between 0 and 100

color ()
: returns the color value

  Returns the color value

flash ()
: flashes the color sensor by switching between color and reflective mode

info ()
: prints the black and white value read form the sensor

read ()
: reads the color sensor data form the file :return:

reflection ()
: gets the reflection from the sensor

  Returns the original reflective lit value without

set_black ()
: sets the current value to black

set_white ()
: sets the current value to white :return:

value ()
: reads the current value mapped b<etween 0 and 100 :return: returns the reflective light mapped between 0 to 100

write ()
: append the black and white value to a file

class `spockbots.colorsensor.`**SpockbotsColorSensors** (*ports=[2, 3, 4], speed=5*)
Bases: `object`

This is how we create the sensors:

colorsensor = SpockbotsColorSensors(ports=[2,3,4]) colorsensor.read()

Now you can use

colorsensor[i].value()

to get the reflective value of the colorsensor on port i. To get the color value we can use

colorsensor[i].color()

clear()
removes the file calibrate.txt

color(*i*)
returns the color value between 0-100 after calibration on the port i

Parameters **i** – number of the port

Returns The color value, blue = 2

flash(*ports=[2, 3, 4]*)
Flashes the light sensor on teh ports one after another

Parameters **ports** – the list of ports to flash

info(*ports=[2, 3, 4]*)
prints the information for each port, e.g. the minimal black and maximum while values :param ports: the list of ports to flash

read(*ports=[2, 3, 4]*)
reads the black and white values to the file calibrate.txt

The values must be written previously. If the file does not exists a default is used.

2: 0, 100 3: 0, 100 4: 4, 40 # because it is higher up so white does

not read that well

test_color(*ports=[2, 3, 4]*)
prints the color value of all senors between 0-100

Parameters **ports** – the list of ports

test_reflective(*ports=[2, 3, 4]*)
prints the reflective value of all senors between 0-100

Parameters **ports** – the list of ports

value(*i*)
returns the reflective value between 0-100 after calibration on the port i

Parameters **i** – number of the port

Returns the reflective color value

write(*ports=[2, 3, 4]*)
writes the black and white values to the file calibrate.txt

Parameters **ports** – the ports used to write

3.5 Spockbots Code

3.5.1 Spockbots Output

```
#!/usr/bin/env pybricks-micropython

import os
from time import sleep
```

(continues on next page)

(continued from previous page)

```
from pybricks import ev3brick as brick
from pybricks.parameters import Color

########################################################
# READ AND WRITE FILES
########################################################

debug = True

########################################################
# READ AND WRITE FILES
########################################################

def readfile(name):
    """
    Reads the file with the name and returns it as a string.

    :param name: The file name
    :return: The data in teh file as string
    """
    try:
        # print ("READ", name)
        f = open(name)
        data = f.read().strip()
        f.close()
        return data
    except:
        return None


def writefile(name, msg):
    """
    Writes a new file with the name. If it exists the
    old file will be deleted.

    :param name: The name of the file
    :param msg: The message to be placed in the file
    :return:
    """
    # print ("WRITE", name, msg)
    # try:
    #     f = open(name)
    #     f.write(msg)
    #     f.close()
    # except Exception as e:
    #     print("FILE WRITE ERROR")
    #     print(e)

    command = 'echo \"' + msg + '\" > ' + name
    # print("COMMNAD:", command)
    os.system(command)


########################################################
# Sound
########################################################
```

(continues on next page)

(continued from previous page)

```
def beep():
    """
    The robot will make a beep
    """
    brick.sound.beep()


def sound(pitch=1500, duration=300):
    """
    plays a sound

    :param pitch: sound pitch
    :param duration: how long the sound plays
    :return:
    """
    brick.sound.beep(pitch, duration)


#######################################################
# LED
#######################################################

def led(color):
    """
    changes color of led light

    :param color: light color
    :param brightness: light brightness
    :return:
    """
    if color == "RED":
        led_color = Color.RED
    elif color == "GREEN":
        led_color = Color.GREEN
    elif color == "YELLOW":
        led_color = Color.YELLOW
    elif color == "BLACK":
        led_color = Color.BLACK
    elif color == "ORANGE":
        led_color = Color.ORANGE
    else:
        led_color = None
    brick.light(led_color)


def flash(colors=["RED", "BLACK", "RED", "BLACK", "GREEN"],
          delay=0.1):
    """
    The robot will flash the LEDs and beep twice
    """
    beep()
    for color in colors:
        led(color)
        sleep(delay)
    beep()
```

(continues on next page)

(continued from previous page)

```
#######################################################
# LCD
#######################################################

def clear():
    """
    clears display

    """
    brick.display.clear()


def PRINT(*args, x=None, y=None):
    """
    prints message on screen at x and y and on the console.
    if x and y are missing prints on next position on lcd screen
    this message prints test messages.

    The sceensize is maximum x=177, y=127)

    :param args: multible strings to be printed in between them
    :param x: x value
    :param y: y value
    """
    text = ""
    for a in args:
        if a is not None:
            text = text + str(a) + " "
    if x is not None and y is not None:
        brick.display.text(text, (x, y))
    else:
        brick.display.text(text)
        print(text)


#######################################################
# Voltage
#######################################################

def voltage():
    """
    prints voltage of battery
    """
    value = brick.battery.voltage() / 1000
    PRINT("Voltage: " + str(value) + " V", 80, 10)
```

3.5.2 Spockbots Check

```
from spockbots.motor import SpockbotsMotor
from spockbots.output import PRINT, led


def check(speed=100, angle=360):
    """
```

(continues on next page)

(continued from previous page)

```
    do a robot check by

    a) turning on the large motors one at a time
    b) turning on the medium motors one at a time
    c) turning on the light sensors one at a time

    :param speed:
    :param angle:
    :return:
    """
    robot = SpockbotsMotor()
    robot.debug = True

    robot.setup()

    speed = speed * 10

    print(robot)

    robot.beep()

    PRINT('Light')

    PRINT('Left')
    led("RED")
    robot.colorsensor[2].flash()

    PRINT('Right')
    led("GREEN")
    robot.colorsensor[3].flash()

    PRINT('Back')
    led("YELLOW")
    robot.colorsensor[4].flash()

    led("GREEN")

    PRINT('Large Motors')

    PRINT('Left')
    led("RED")
    robot.left.run_angle(speed, angle)

    PRINT('Right')
    led("GREEN")

    robot.right.run_angle(speed, angle)

    PRINT('Medium Motors')

    PRINT('Left')
    led("RED")
    robot.left_medium.run_angle(speed, angle)

    PRINT('Right')
    led("GREEN")
    robot.right_medium.run_angle(speed, angle)
```

(continues on next page)

(continued from previous page)

```
    PRINT('finished')
```

3.5.3 Spockbots Colorsensor

```
#!/usr/bin/env micropython

from time import sleep

from pybricks import ev3brick as brick
from pybricks.ev3devices import ColorSensor
from pybricks.parameters import Port


class SpockbotsColorSensor:
    """
    defines a Colorsensor with values between 0 and 100
    """

    def __init__(self, port=3):
        """

        :param port: the port
        :param speed: teh speed for calibration
        """
        """
        :param: number  number of color sensor on ev3
        """
        if port == 1:
            self.sensor = ColorSensor(Port.S1)
        elif port == 2:
            self.sensor = ColorSensor(Port.S2)
        elif port == 3:
            self.sensor = ColorSensor(Port.S3)
        elif port == 4:
            self.sensor = ColorSensor(Port.S4)

        self.port = port
        self.black = 100
        self.white = 0

    def set_white(self):
        """
        sets the current value to white
        :return:
        """
        value = self.sensor.reflection()
        if value > self.white:
            self.white = value

    def set_black(self):
        """
        sets the current value to black
        """
        value = self.sensor.reflection()
```

(continues on next page)

(continued from previous page)

```
        if value < self.black:
            self.black = value

    def reflection(self):
        """
        gets the reflection from the sensor

        :return: the original reflective lit value without
        """
        return self.sensor.reflection()

    def color(self):
        """
        returns the color value

        :return: the color value
        """
        return self.sensor.color()

    def value(self):
        """
        reads the current value mapped b<etween 0 and 100
        :return: returns the reflective light mapped between 0 to 100
        """
        val = self.sensor.reflection()
        b = self.black
        t1 = val - b
        t2 = self.white - self.black
        ratio = t1 / t2
        c = ratio * 100
        if c < 0:
            c = 0
        if c > 100:
            c = 100
        return int(c)

    def flash(self):
        """
        flashes the color sensor by switching between
        color and reflective mode
        """

        brick.sound.beep()
        light = self.sensor.rgb()
        sleep(0.3)
        light = self.sensor.reflection()
        sleep(0.3)

    def write(self):
        """
        append the black and white value to a file
        """
        f = open("/home/robot/calibrate.txt", "w+")
        f.write(str(self.sensor.black) + "\n")
        f.write(str(self.sensor.white) + "\n")
        f.close()
```

(continues on next page)

(continued from previous page)

```
    def info(self):
        """
        prints the black and white value read form the
        sensor
        """
        print("colorsensor",
              self.port,
              self.black,
              self.white)

    def read(self):
        """
        reads the color sensor data form the file
        :return:
        """
        try:
            f = open("/home/robot/calibrate.txt", "r")
            self.colorsensor[port].black = int(f.readline())
            self.colorsensor[port].white = int(f.readline())
            f.close()
        except:
            print("we can not find the calibration file")


class SpockbotsColorSensors:
    """

    This is how we create the sensors:

        colorsensor = SpockbotsColorSensors(ports=[2,3,4])
        colorsensor.read()

    Now you can use

        colorsensor[i].value()

    to get the reflective value of the colorsensor on port i.
    To get the color value we can use

        colorsensor[i].color()

    """

    def __init__(self, ports=[2, 3, 4], speed=5):
        """
        Creates the color sensors for our robot.
        Once calibrated, the sensor values always return 0-100,
        where 0 is black and 100 is white

        :param ports: the list of ports we use on the robot for color sensors
        :param speed: The speed for the calibration run
        """
        self.ports = ports
        self.speed = speed
        self.colorsensor = [0, 0, 0, 0, 0]
            # in python lists start from 0 not 1
            # so we simply do not use the firts element in the list
```

(continues on next page)

(continued from previous page)

```
        # our robot uses only
        #   colorsensor[2]
        #   colorsensor[3]
        #   colorsensor[4]
        #   the ports are passed along as a list [2,3,4]
        self.ports = ports
        for i in ports:
            self.colorsensor[i] = SpockbotsColorSensor(port=i)

    def value(self, i):
        """
        returns the reflective value between 0-100 after
        calibration on the port i

        :param i: number of the port
        :return: the reflective color value
        """
        return self.colorsensor[i].value()

    def color(self, i):
        """
        returns the color value between 0-100 after
        calibration on the port i

        :param i: number of the port
        :return: The color value, blue = 2
        """
        return self.colorsensor[i].color()

    def write(self, ports=[2, 3, 4]):
        """
        writes the black and white values to the file
        calibrate.txt

        :param ports: the ports used to write
        """

        f = open("/home/robot/calibrate.txt", "w")
        for i in ports:
            f.write(str(self.colorsensor[i].black) + "\n")
            f.write(str(self.colorsensor[i].white) + "\n")
        f.close()

    def clear(self):
        """
        removes the file calibrate.txt
        """
        f = open("/home/robot/calibrate.txt", "w")
        f.close()

    def read(self, ports=[2, 3, 4]):
        """
        reads the black and white values to the file
        calibrate.txt

        The values must be written previously. If the file
        does not exists a default is used.
```

(continues on next page)

(continued from previous page)

```
            2: 0, 100
            3: 0, 100
            4: 4, 40    # because it is higher up so white does
                          not read that well
        """
        try:
            f = open("/home/robot/calibrate.txt", "r")
            for i in ports:
                self.colorsensor[i].black = int(f.readline())
                self.colorsensor[i].white = int(f.readline())
            f.close()
        except:
            print("we can not find the calibration file")
            self.colorsensor[2].black = 9
            self.colorsensor[2].white = 100
            self.colorsensor[3].black = 10
            self.colorsensor[3].white = 100
            self.colorsensor[4].black = 4
            self.colorsensor[4].white = 48
            print("Using the following defaults")
            self.info()

    def flash(self, ports=[2, 3, 4]):
        """
        Flashes the light sensor on teh ports one after another

        :param ports: the list of ports to flash
        """
        for port in ports:
            self.colorsensor[port].flash()

    def info(self, ports=[2, 3, 4]):
        """
        prints the information for each port, e.g.
        the minimal black and maximum while values
        :param ports: the list of ports to flash
        """
        print("")
        print("Color sensor black and white values")
        print("")

        for port in ports:
            self.colorsensor[port].info()
        print()

    def test_reflective(self, ports=[2, 3, 4]):
        """
        prints the reflective value of all senors between 0-100

        :param ports: the list of ports
        """
        print("")
        print("Color sensor tests")
        print("")

        for port in ports:
            v = self.colorsensor[port].value()
```

(continues on next page)

(continued from previous page)

```
            print("Color sensor", port, v)
        print()

    def test_color(self, ports=[2, 3, 4]):
        """
        prints the color value of all senors between 0-100

        :param ports: the list of ports
        """
        print("")
        print("Color sensor tests")
        print("")

        for port in ports:
            v = self.colorsensor[port].value()
            print("Color sensor", port, v)
        print()
```

3.5.4 Spockbots Gyro

```
import sys
import time
from time import sleep

from pybricks.ev3devices import GyroSensor
from pybricks.parameters import Direction
from pybricks.parameters import Port


########################################################
# Gyro
########################################################

class SpockbotsGyro(object):
    """
    improved gyro class
    """
    # The following link gives some hints why it does not
    # work for the Gyro in mindstorm
    # http://ev3lessons.com/en/ProgrammingLessons/advanced/Gyro.pdf

    # in python we have three issues

    # sensor value is not 0 after reset
    # sensor value drifts after reset as it takes time
    #        to settle down
    # sensor value is not returned as no value is available
    #        from the sensor

    # This code fixes it.

    def __init__(self, robot, port=1):
        """
        Initializes the Gyro Sensor
```

(continues on next page)

(continued from previous page)

```
        :param robot: robot varible
        :param port: port number for gyro sensor 1,2,3,4
        :param direction: if front if we drive forward
                          otherwise backwards
        """

        self.robot = robot
        if self.robot.direction == "forward":
            sensor_direction = Direction.CLOCKWISE
        else:
            sensor_direction = Direction.COUNTERCLOCKWISE

        found = False
        while not found:
            print("FINDING GYRO")
            try:
                if port == 1:
                    self.sensor = GyroSensor(Port.S1,
                                             sensor_direction)
                elif port == 2:
                    self.sensor = GyroSensor(Port.S2,
                                             sensor_direction)
                elif port == 3:
                    self.sensor = GyroSensor(Port.S3,
                                             sensor_direction)
                elif port == 4:
                    self.sensor = GyroSensor(Port.S4,
                                             sensor_direction)

                print("SENSOR:", self.sensor)
                sleep(0.1)
                self.sensor.reset_angle(0)
                found = True
            except Exception as e:
                print(e)
                if "No such sensor on Port" in str(e):
                    print()
                    print("ERROR: THe Gyro Sensor is disconnected")
                    print()
                    sys.exit()

        # bug should be = 0
        self.last_angle = -1000  # just set the current value
                                 # to get us started
        print("GYRO INITIALIZED")

    def angle(self):
        """
        Gets the angle

        :return: The angle in degrees
        """
        try:
            s = self.sensor.speed()
            a = self.sensor.angle()
            print("AS", a, s)
```

(continues on next page)

(continued from previous page)

```
            self.last_angle = a
        except:
            print("Gyro read error")
            a = self.last_angle

        return a

    def zero(self):
        """
        set the gyro angle to 0
        :return:
        """
        self.sensor.reset_angle(0)

        angle = 1000
        while angle != 0:
            # sleep(0.1)
            angle = self.angle()

    def still(self):
        """
        tests if robot dosent move
        :return: True if robot dosent move
        """
        return not self.drift()

    def drift(self):
        """
        tests if robot drifts and waits until its still

        """
        # loop in case we get a read error from the gyro speed
        while True:
            try:
                speed = self.sensor.speed()
                if speed == 0:
                    return False  # no drift if the speed is 0
                else:
                    return True  # DRIFT IF THE SPEED IS NOT 0
            except:
                print("ERROR: DRIFT no value found")
                # No speed value found, so repeat

    def status(self, count=10):
        """
        tests count times if robot is still and returns if its still or drifts
        :param count: number of times tested if its still
        :return: still,drift which are true/false
        """
        last = self.angle()
        i = 0
        still = 0
        drift = 0
        while i <= count:
            angle = self.angle()
            if angle == last:
                still = still + 1
```

(continues on next page)

(continued from previous page)

```
                drift = 0
            else:
                drift = drift + 1
            i = i + 1
        print("GYRO STATUS", i, still, drift)
        return still >= count, drift >= count

    def reset(self):
        """
        safely resets the gyro
        """

        self.sensor.reset_angle(0)
        try:
            self.last_angle = angle = self.sensor.angle()
        except:
            print("Gyro read error")
            self.last_angle = angle = -1000

        count = 10
        while count >= 0:
            # sleep(0.1)
            try:
                angle = self.sensor.angle()
            except:
                print("Gyro read error", angle)
            print(count, "Gyro Angle: ", angle)
            if abs(angle) <= 2:
                count = count - 1
        self.last_angle = angle
        print("Gyro Angle, final: ", angle)

    def turn(self, speed=25, degrees=90):
        """
        uses gyro to turn positive to right negative to left
        :param speed: speed it turns at
        :param degrees: degrees it turns
        :return:
        """
        if degrees < 0:
            self.left(speed=speed, degrees=abs(degrees))
        elif degrees > 0:
            self.right(speed=speed, degrees=abs(degrees))

    def left(self, speed=25, degrees=90, offset=0):
        """
        The robot turns left with the given number of degrees

        :param speed: The speed
        :param degrees: The degrees
        :param offset:
        :return:
        """
        self.reset()
        if speed == 25:
            offset = 8
```

(continues on next page)

(continued from previous page)

```
        # self.zero()

        self.robot.on_forever(speed, -speed)
        angle = self.angle()

        print(angle, -degrees + offset)

        while angle > -degrees + offset:
            # print(angle, -degrees + offset)
            angle = self.angle()
        self.robot.stop()

    def right(self, speed=25, degrees=90, offset=0):
        """
        The robot turns right with the given number of degrees

        :param speed: The speed
        :param degrees: The degrees
        :param offset:
        :return:
        """
        self.reset()

        if speed == 25:
            offset = 8

        # self.zero()

        self.robot.on_forever(-speed, speed)
        angle = self.angle()
        print(angle, degrees - offset)
        while angle < degrees - offset:
            # print(angle, degrees - offset)
            angle = self.angle()
        self.robot.stop()


    def forward(self,
                speed=25,  # speed 0 - 100
                distance=None,  # distance in cm
                t=None,
                port=1,  # the port number we use to follow the line
                delta=-35,  # control smoothness
                factor=0.4):  # parameters to control smoothness
        """
        Moves forward

        :param speed: The speed
        :param distance: If set the distance to travle
        :param t: If set the time to travel
        :param port: The port number of the Gyro sensor
        :param delta: controlling the smoothness of the line
        :param factor: controlling the smoothness of the line
        :return:
        """

        if right:
```

(continues on next page)

(continued from previous page)

```
            f = 1.0
        else:
            f = - 1.0

        if distance is not None:
            distance = 10 * distance

        current = time.time()  # the current time
        if t is not None:
            end_time = current + t  # the end time

        self.reset()

        while True:
            value = self.angle()  # get the Gyro angle value

            # correction = delta + (factor * value)
            # calculate the correction for steering
            correction = f * factor * (value + delta)

            self.on(speed, correction)  # switch the steering on
                               # with the given correction and speed

            # if the time is used we set run to false once
            #        the end time is reached
            # if the distance is greater than the
            #        position than the leave the
            distance_angle = self.left.angle()

            traveled = self.angle_to_distance(distance_angle)

            current = time.time()  # measure the current time
            if t is not None and current > end_time:
                break  # leave the loopK
            if distance is not None and distance < traveled:
                break  # leave the loop

        self.stop()  # stop the robot
```

3.5.5 Spockbots Motor Code

```
import math
import time

from pybricks import ev3brick as brick
from pybricks.ev3devices import Motor
from pybricks.parameters import Port, Button
from pybricks.parameters import Stop, Direction
from pybricks.robotics import DriveBase
# from pybricks.ev3devices import ColorSensor
# from spockbots.colorsensor import SpockbotsColorSensor
from spockbots.colorsensor import SpockbotsColorSensors
from spockbots.output import PRINT
from threading import Thread
import sys
```

(continues on next page)

(continued from previous page)

```
########################################################
# Motor
########################################################


class SpockbotsMotor(object):

    def check_kill_button(self):
        """
        This will stop all motors  and finish the program.
        It can be used in the programs to check if the program should be
        finished early du to an error in the runs.
        """
        if Button.LEFT in brick.buttons():
            print("KILL")
            self.beep()
            self.beep()
            self.beep()
            self.beep()

            self.stop()
            self.left_medium.stop(Stop.BRAKE)
            self.right_medium.stop(Stop.BRAKE)

    def __init__(self, direction=None):
        """
        defines the large motors (left and right),
        the tank move, and the medium motors.

        :param direction: if the direction is 'forward'
                          the robot moves forward, otherwise
                          backwards.

        """

        self.diameter = round(62.4, 3)  # mm
        self.width = 20.0  # mm
        self.circumference = round(self.diameter * math.pi, 3)
        # self.axle_track = round(8.0 * 14, 3)
        self.axle_track = 140.0
        self.direction = "forward"

        self.left, self.right, self.tank = \
            self.setup(direction=direction)

        self.color = SpockbotsColorSensors(ports=[2, 3, 4])
        self.colorsensor = [None, None, None, None, None]

        for port in [2, 3, 4]:
            self.colorsensor[port] = self.color.colorsensor[port]

    def beep(self):
        """
        robot will beep.
```

(continues on next page)

(continued from previous page)

```
        """
        brick.sound.beep()

    def __str__(self):
        PRINT()
        PRINT("Robot Info")
        PRINT("============================")
        PRINT("Tire Diameter:", self.diameter)
        PRINT("Circumference:", self.circumference)
        PRINT("Tire Width:    ", self.width)
        PRINT("Axle Track:    ", self.axle_track)
        PRINT("Angle Left:    ", self.left.angle())
        PRINT("Angle Right:   ", self.right.angle())
        PRINT("Direction:     ", self.direction)

        PRINT()
        return ""

    def setup(self, direction=None):
        """
        setup the direction, the motors, and the tank with the appropriate direction.

        :param direction: if the direction is 'forward' the robot moves forward,␣
↪otherwise backwards.
        :return: left, right motors  and tank

        """
        self.check_kill_button()

        if direction is None:
            self.direction = "forward"
        else:
            self.direction = direction

        if self.direction == "forward":

            self.left = Motor(Port.A, Direction.COUNTERCLOCKWISE)
            self.right = Motor(Port.B, Direction.COUNTERCLOCKWISE)
        else:
            self.left = Motor(Port.A, Direction.CLOCKWISE)
            self.right = Motor(Port.B, Direction.CLOCKWISE)

        self.tank = DriveBase(self.left, self.right,
                              self.diameter, self.axle_track)

        self.left_medium = Motor(Port.D, Direction.CLOCKWISE)
        self.right_medium = Motor(Port.C, Direction.CLOCKWISE)

        return self.left, self.right, self.tank

    def value(self, port):
        """
        return the reflective color sensor value.

        :param port: the port number of the color sensor
        :return: the reflective color value
```

(continues on next page)

(continued from previous page)

```
        """
        return self.colorsensor[port].value()

    def reset(self):
        """
        resets the angle in the large motors to 0.

        """

        self.left.reset_angle(0)
        self.right.reset_angle(0)

    def on(self, speed, steering=0):
        """
        turns the large motors on while using steering.

        :param speed: the speed of the robot
        :param steering: an angle for the steering

        """

        self.tank.drive(speed * 10, steering)

    def distance_to_rotation(self, distance):
        """
        calculation to convert the distance from cm into rotations.

        :param distance:  The distance in cm
        :return: The rotations to be traveled for the given distance

        """

        rotation = distance / self.circumference
        return rotation

    def distance_to_angle(self, distance):
        """
        calculation to convert the distance from cm into angle.

        :param distance:  The distance in cm
        :return: The degrees traveled for the given distance

        """

        rotation = self.distance_to_rotation(distance) * 360.0
        return rotation

    def angle_to_distance(self, angle):
        """
        calculation to return the distance in cm given an angle.

        :param angle: the angle
        :return: distance in cm for turning an angle

        """

        d = self.circumference / 360.0 * angle
```

(continues on next page)

(continued from previous page)

```
        return d

    def stop(self, brake=None):
        """
        stops all motors on all different drive modes.

        :param brake: None, brake, coast, hold

        """

        if not brake or brake == "brake":
            self.left.stop(Stop.BRAKE)
            self.right.stop(Stop.BRAKE)
            self.tank.stop(Stop.BRAKE)
        elif brake == "coast":
            self.left.stop(Stop.COAST)
            self.right.stop(Stop.COAST)
            self.tank.stop(Stop.COAST)
        elif brake == "hold":
            self.left.stop(Stop.HOLD)
            self.right.stop(Stop.HOLD)
            self.tank.stop(Stop.HOLD)

        self.still()

    def still(self):
        """
        waits till the motors are no longer turning.
        """

        PRINT("Still Start")

        count = 10
        angle_left_old = self.left.angle()
        angle_right_old = self.right.angle()
        while count > 0:
            angle_left_current = self.left.angle()
            angle_right_current = self.right.angle()
            if angle_left_current == angle_left_old and \
                    angle_right_current == angle_right_old:
                count = count - 1
            else:
                angle_left_old = angle_left_current
                angle_right_old = angle_right_current

        PRINT("Still Stop")

    def forward(self, speed, distance, brake=None):
        """
        the robot drives forward for a given distance.

        :param speed: speed of robot
        :param distance: distance that robot goes forward (in cm)
        :param brake: one of the values brake, hold, coast

        """
        self.check_kill_button()
```

(continues on next page)

(continued from previous page)

```
        PRINT("Forward", speed, distance, brake)

        if distance < 0:
            speed = -speed
            distance = -distance

        self.reset()
        angle = abs(self.distance_to_angle(10 * distance))
        self.on(speed)

        run = True
        while run:
            current = abs(self.left.angle())
            run = current < angle

        self.stop(brake=brake)

        PRINT("Forward Stop")

    def on_forever(self, speed_left, speed_right):
        """
        turns motors on with left and right speed.

        :param speed_left: speed of left motor
        :param speed_right: speed of the right motor

        """
        self.check_kill_button()

        PRINT("on_forever", speed_left, speed_right)
        self.reset()

        self.left.run(speed_left * 10)
        self.right.run(speed_right * 10)

    def turn(self, speed, angle):
        """
        takes the radius of the robot and dives on it
        for a distance based on the angle.

        :param speed: speed of turn
        :param angle: angle of turn

        """
        self.check_kill_button()

        PRINT("Turn", speed, angle)

        self.reset()

        c = self.axle_track * math.pi
        fraction = 360.0 / angle
        d = c / fraction
        a = self.distance_to_angle(d)

        self.left.run_angle(speed * 10, -a, Stop.BRAKE, False)
```

(continues on next page)

(continued from previous page)

```
        self.right.run_angle(speed * 10, a, Stop.BRAKE, False)

        count = 10
        old = abs(self.left.angle())
        while abs(self.left.angle()) < abs(a) or \
                abs(self.right.angle()) < abs(a):

            PRINT("TURN CHECK", count, old,
                  abs(self.left.angle()),
                  abs(self.right.angle()))
            if old == abs(self.left.angle()):
                count = count - 1
            else:
                old = abs(self.left.angle())

            if count < 0:
                PRINT("FORCED STOP IN TURN")
                self.beep()
                break
        self.stop()

        PRINT("Turn Stop")

    def turntoblack(self,
                    speed,
                    direction="left",
                    port=3,
                    black=10):
        """
        turns the robot to the black line.

        :param speed: speed of turn
        :param direction: left or right
        :param port: port of color sensor
        :param black: value of black

        """
        self.check_kill_button()

        PRINT("turntoblack", speed, direction, port, black)

        if direction == "left":
            self.left.run(speed * 10)
        else:
            self.right.run(speed * 10)

        while self.value(port) > black:
            pass
        self.stop()

    def turntowhite(self,
                    speed,
                    direction="left",
                    port=3,
                    white=80):
        """
        turns the robot to the white line.
```

(continues on next page)

(continued from previous page)

```
        :param speed: speed of turn
        :param direction: left or right
        :param port: port of color sensor
        :param white: value of white

        """
        self.check_kill_button()

        PRINT("turntowhite", speed, direction, port, white)

        if direction == "left":
            self.left.run(speed * 10)
        else:
            self.right.run(speed * 10)

        while self.value(port) < white:
            pass
        self.stop()

    def aligntoblack(self, speed, port_left, port_right, black=10):
        """
        aligns with black line while driving each motor.

        :param speed: speed of robot
        :param port_left: port of left color sensor
        :param port_right: port of right color sensor
        :param black: value of black

        """
        self.check_kill_button()

        PRINT("aligntoblack", speed, port_left, port_right, black)

        self.on_forever(speed, speed)
        left_finished = False
        right_finished = False

        while not (left_finished and right_finished):
            left_light = self.value(port_left)
            right_light = self.value(port_right)
            PRINT("Light", left_light, right_light)
            if left_light < black:
                self.left.stop(Stop.BRAKE)
                left_finished = True
            if right_light < black:
                self.right.stop(Stop.BRAKE)
                right_finished = True
        self.stop()

        PRINT("aligntoblack Stop")

    def aligntowhite(self, speed, port_left, port_right, white=80):
        """
        aligns with white line while driving each motor.

        :param speed: speed of robot
```

(continues on next page)

(continued from previous page)

```
        :param port_left: port of left color sensor
        :param port_right: port of right color sensor
        :param white: value of white

        """
        self.check_kill_button()

        PRINT("aligntoblack", speed, port_left, port_right, white)

        self.on_forever(speed, speed)
        left_finished = False
        right_finished = False

        while not (left_finished and right_finished):
            left_light = self.value(port_left)
            right_light = self.value(port_right)
            PRINT("Light", left_light, right_light)
            if left_light > white:
                self.left.stop(Stop.BRAKE)
                left_finished = True
            if right_light > white:
                self.right.stop(Stop.BRAKE)
                right_finished = True
        self.stop()

        PRINT("aligntowhite Stop")

    def gotoblack(self, speed, port, black=10):
        """
        robot moves to the black line while using the
        sensor on the given port.

        :param speed: speed of robot
        :param port: port of color sensor
        :param black: value of black

        """
        self.check_kill_button()

        PRINT("gotoblack", speed, port, black)

        # self.left.run_angle(speed * 10, -a, Stop.BRAKE, False)
        # self.right.run_angle(speed * 10, a, Stop.BRAKE, False)

        self.on(speed, 0)
        while self.value(port) > black:
            pass
        self.stop()

        PRINT("gotoblack Stop")

    def gotowhite(self, speed, port, white=90):
        """
        robot moves to the white line while using
        the sensor on the given port.

        :param speed: speed of robot
```

(continues on next page)

(continued from previous page)

```
        :param port: port of color sensor
        :param white: value of white

        """
        self.check_kill_button()

        PRINT("gotowhite", speed, port, white)

        self.on(speed, 0)
        while self.value(port) < white:
            pass
        self.stop()

        PRINT("gotowhite Stop")

    def gotocolor(self, speed, port, colors=[0]):
        """
        robot moves to the black line while using the
        sensor on the given port.

        :param speed: speed of robot
        :param port: port of color sensor
        :param black: value of black

        """
        self.check_kill_button()

        PRINT("gotocolor", speed, port, colors)

        self.on(speed, 0)
        run = True
        while run:
            value = self.colorsensor[port].sensor.color()
            print("COLOR", value)
            run = value not in colors
        self.stop()

        PRINT("gotocolor Stop")

    def followline(self,
                   speed=25,  # speed 0 - 100
                   distance=None,  # distance in cm
                   t=None,
                   port=3,  # the port number we use to follow the line
                   right=True,  # the side on which to follow the line
                   stop_color_sensor=None,
                   stop_values=None,  # [4,5]
                   stop_color_mode=None,  # color, reflective
                   delta=-35,  # control smoothness
                   factor=0.4):  # parameters to control smoothness

        """
        follows line for either a distance or for time.

        :param speed: speed of robot
        :param distance: distance that robot follows line
        :param t: time that robot follows line for
```

(continues on next page)

(continued from previous page)

```
        :param port: port of color sensor
        :param right: whether the robot is following
                      the right or left side of line
        :param black: black value
        :param white: white value
        :param delta: adjustment value to convert from color
                      sensor values (0 to 100) to motor
                      steering (-100 to 100)
        :param factor: factor of adjustment, controls smoothness

        """
        self.check_kill_button()

        if right:
            f = 1.0
        else:
            f = - 1.0

        if distance is not None:
            distance = 10 * distance

        current = time.time()  # the current time
        if t is not None:
            end_time = current + t  # the end time

        self.reset()

        while True:
            self.check_kill_button()
            value = self.value(port)  # get the light value

            # correction = delta + (factor * value)
            # calculate the correction for steering
            correction = f * factor * (value + delta)
            # correction = f * correction
            # if we drive backwards negate the correction

            self.on(speed, correction)
            # switch the steering on with the given correction and speed

            # if the time is used we set run to
            #         false once the end time is reached
            # if the distance is greater than the
            #         position than the leave the
            angle = self.left.angle()

            traveled = self.angle_to_distance(angle)

            current = time.time()  # measure the current time
            if t is not None and current > end_time:
                break  # leave the loopK
            if distance is not None and distance < traveled:
                break  # leave the loop
            if stop_color_sensor is not None:
                if stop_color_mode == "color":
                    value = self.colorsensor[port].sensor.color()
                    # value = self.colorsensor[port].sensor.rgb()
```

(continues on next page)

(continued from previous page)

```
                elif stop_color_mode == "reflective":
                    value = self.colorsensor[port].value()
                print("VALUE", value)
                if value in stop_values:
                    break  # leave the loop

        self.stop()  # stop the robot

    def calibrate(self, speed, distance=15, ports=[2, 3, 4], direction='front'):
        """
        calibrates color sensors by driving over black and white line.

        :param speed: speed of robot
        :param distance: distance that robot travels
        :param ports: ports of color sensors
        :param direction: direction of calibration

        """
        self.check_kill_button()

        self.reset()
        self.on(speed, 0)
        distance = self.distance_to_angle(distance * 10)

        while self.left.angle() < distance:

            for i in ports:
                self.colorsensor[i].set_white()
                self.colorsensor[i].set_black()
                PRINT(i,
                      self.colorsensor[i].black,
                      self.colorsensor[i].white)

        self.stop()

        for i in ports:
            PRINT(i,
                  self.colorsensor[i].black,
                  self.colorsensor[i].white)

    # followline_pid(distance=45, port=3, speed=20, black=0, white=100, kp=0.3, ki=0.01, kd=0.0)
    def followline_pid(self,
                       debug=False,
                       distance=None,  # distance in cm
                       t=None,
                       right=True,  # the side on which to follow the line
                       stop_color_sensor=None,
                       stop_values=None,  # [4,5]
                       stop_color_mode=None,  # color, reflective
                       port=3, speed=25, black=0, white=100, kp=0.3, ki=0.0, kd=0.0):
        self.check_kill_button()

        if right:
            f = 1.0
        else:
            f = - 1.0
```

(continues on next page)

(continued from previous page)

```
        if distance is not None:
            distance = 10 * distance

        current = time.time()  # the current time
        if t is not None:
            end_time = current + t  # the end time

        self.reset()

        integral = 0

        midpoint = (white - black) / 2 + black
        lasterror = 0.0

        loop_start_time = current = time.time()

        print("kp=", kp, "ki=", ki, "kd=", kd)
        while True:
            self.check_kill_button()
            try:
                value = self.value(port)  # get the light value

                error = midpoint - value
                integral = error + integral
                derivative = error - lasterror

                correction = f * kp * error + ki * integral + kd * derivative

                lasterror = error

                self.on(speed, correction)
                # switch the steering on with the given correction and speed

                # if the time is used we set run to
                #         false once the end time is reached
                # if the distance is greater than the
                #         position than the leave the
                angle = self.left.angle()
                traveled = self.angle_to_distance(angle)
                current = time.time()  # measure the current time

                if debug:
                    if correction > 0.0:
                        bar = str(30 * ' ') + str('#' * int(correction))
                    elif correction < 0.0:
                        bar = ' ' * int(30 + correction) + '#' * int(abs(correction))
                    else:
                        bar = 60 * ' '

                    print("{:4.2f} {:4.2f} {:4.2f} {:3d} {}".format(correction,
↪traveled, current - loop_start_time,
                                                                    value, bar))

                if t is not None and current > end_time:
                    break  # leave the loopK
                if distance is not None and distance < traveled:
```

(continues on next page)

(continued from previous page)

```
                break  # leave the loop
            if stop_color_sensor is not None:
                if stop_color_mode == "color":
                    value = self.colorsensor[port].sensor.color()
                    # value = self.colorsensor[port].sensor.rgb()
                elif stop_color_mode == "reflective":
                    value = self.colorsensor[port].value()
                print("VALUE", value)
                if value in stop_values:
                    break  # leave the loop
        except Exception as e:
            print(e)
            break
    self.stop()  # stop the robot
```

3.6 lego

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

S

T

V

W

Z