Spockbots

Release 1.0

Spockbots

Nov 15, 2019

CONTENTS

1 Coaches Disclaimer **1**
- 1.1 Team background . . . 1
- 1.2 Choice of not using Scratch or Java . . . 2
- 1.3 Choice of Python . . . 2
- 1.4 Choice of the OS . . . 2
- 1.5 Pitfalls of Mindtstorm and Python . . . 3

2 Python Essentials **7**
- 2.1 Variables . . . 7
- 2.2 Lists . . . 7
- 2.3 Functions . . . 7
- 2.4 Classes . . . 7
- 2.5 Conditions . . . 8
- 2.6 Loops . . . 8
- 2.7 Exceptions . . . 9
- 2.8 Function as a parameter . . . 10
- 2.9 Import . . . 10
- 2.10 Program . . . 10

3 Lessons **13**
- 3.1 Python Programming Templates and Activities . . . 13
- 3.2 Color Sensor . . . 13
- 3.3 Three color sensors . . . 17
- 3.4 Driving The Robot . . . 19
- 3.5 Gyro Sensor . . . 27

4 Spockbots City **33**

5 Summary **35**
- 5.1 Summary of Robot Features . . . 35

6 Spockbots Robot **37**
- 6.1 Design . . . 37
- 6.2 MECAHNICS TBD . . . 39
- 6.3 Menu . . . 39
- 6.4 City Runs . . . 39
- 6.5 Spockbots API . . . 45
- 6.6 Spockbots Code . . . 53
- 6.7 lego . . . 83

7 Indices and tables **85**

Python Module Index **87**

Index **89**

CHAPTER

ONE

COACHES DISCLAIMER

We have included in this document some information provided to the kids in order to be transparent with the help and educational material that has been issued to the team and to clearly distinguish what was done by coaches and what was done by the team.

We apologize for this long document, but we want to make sure we communicate our activities properly to all. However, the judges can skip this section and go directly to the *Spockbots City* section if they wish.

The information in the coaches disclaimer has been collected by the coaches, with some input from the team but in large part it has been conducted just by the coaches as what we have done here is only needed due to the evolving python mindstorm ecosystem. Naturally, in the mindstorm GUI you do not have these issue as the GUI is very stable and a proven way of doing competitions.

1.1 Team background

Most of the team members have extensive experience in the Mindstorm EV3 GUI programming language. They developed for over more than 2 years a library that includes:

- Gyro sensor based
  - turn
  - forward
  - reset with wait
- Color sensor based
  - calibration of multiple sensors while storing the failures in a file
  - using the calibrated minimum and maximum values for decisions if it is black and white
  - line following on black
  - line following on color
  - drive to black
  - drive to white
  - align to black
  - align to white
- One of the other strengths of the team was to build fancy attachments to their box robot they improved over the last two years.

Note: THus, he team has an existing library for minstorm GUI and they could have made it easy on themselves to just reuse this Mindstorm Myblocks library.

1.2 Choice of not using Scratch or Java

We shortly discussed using scratch and some of the kids do not like it and instead wanted to use Python, as we used Python in the past on the projects. There was also no need to use Java as we could leverage the kid's experience in python and focus on FLL learning tasks.

1.3 Choice of Python

The team came together and at discussed if to use Python or Mindstorm GUI and it was decided as we already did Mindstorms to try out Python. The team had in addition a vote a month back to decide if we should switch back to the GUI as the python programming did not progress as smoothly as expected. This is actually not related to the kid's ability to learn and explore new things, but to subtle differences between the GUI and Python, providing some unbalanced advantage at this time still towards the GUI do to some administrative functions that are included in the GUI that are not yet exposed or corrected in the Python API.

1.4 Choice of the OS

During the project we have, with little input from the kids, experimented with different operating systems and python versions.

Ev3dev, Visual Code and Python3

> Initially we selected ev3dev.org as the coach used that about 4-5 years ago and contributed to the community a use case to run jupyter notebook services directly on the lego brick so students could use jupyter on their laptops to interactively use the robots via jupyter directly from the Laptops. We however did not use this feature and instead started with Visual Code due to their ability to use a proper IDE instead of using notebooks.
>
> The student explored the features of interfacing in Python with the brick including focus on the build in line follower
>
> However the coaches made an executive decision not to use this version even after they updated to the snapshot releases due to two main issues: (1) we regularly observed that when something goes wrong the motor continue to operate; and due to the long start up time of python3 that made development too slow and cumbersome.
>
> The coaches explored advanced features such as a kill button via a thread and the use of an absolute drive function that is default in ev3dev but requires threading and python3 was needed.
>
> After however seeing the slow progress the students made while waiting too long for the programs to start. The coaches mad the decision to use pybricks-micropython. At this time the team was asked if we should switch back to Mindstorm GUI, but it was decided to stick with Python as it a more educational goal for this team.
>
> In addition to these versions we also experimented with the releases for "stretch" and "buster" we used "buster", but learned later in the forums we should have used "stretch" due to issues with the overall reliability. However at that time we were more focused on speed of startup of python.

pybricks-micropython

After experimenting with pybricks-micropython we found that the python loading time were significantly reduced. We saw a significant increase in productivity as the kids could focus on programming and not on waiting for the program to get started. However, this OS and the python version is far less than ideal as it does not provide the more advanced features than the ev3dev OS provides. We also found that the micropython version of pybricks is faster than the one on ev3dev.

1.5 Pitfalls of Mindtstorm and Python

In this section we document some of the pitfalls that we believe should be considered to be fixed.

Runaway Motors

(+) GUI: This does not happen in Mindstorm GUI giving it an advantage. Also the backspace button interrupts the program.

(-) Python: We observed that in some cases when using ev3dev the motors simply run away and can not be made to stop. This seems to be discussed online as one of the open bugs. A solution is posted in the ev3dev documentation but uses threads and can not be applied to all micropython versions. However, when applying this solution we sometimes still ended up in runaway motors.

Gyro Hardware Differences

(+) GUI: The forums in the Internet have plenty of documentation on resetting the Gyros into a workable framework. This includes switching sensor modes, introducing timed loops and check for angles. Today it is easy for students to find them and copy them into their programs. Our team simply used a delay of 0.1 seconds which was in most cases sufficient for our previous FLL participations.

(-) Python: Due to the newness of python the reset is not properly discussed, furthermore, the reset into different senor modes although possible in the GUI requires elevated permissions in Python which gives the GUI an advantage as they do not have to learn how to become a system administrator in Linux ;-)

(-) Problem for both: We had more than one robot and we found that we had some hardware issues with one of our Gyro sensors, as the reset did not function well. Without input from the kids we replaced this broken sensor with a new one. However the kids struggled for a long time trying to get that sensor to go until the coaches took a closer look at it and identified a hardware difference/fault. If we would not have had more than one robot we would not have been able to identify this and the team would still try to get the gyro to get working ;-) The interesting part was that just switching to a different sensor it worked much more reliable.

In retrospect we found a significant set of documentation by one coach that discusses the difference between the many Gyro sensors. I think in python we see the same issue as discussed for the GUI version.

Motor Stall on Angle:

(+) GUI: this isse does not occur in the GUI version and the motor returns after using a number of degrees or rotations within wait blocks.

(-): Python: While driving forward for an angle or given rotation we find that sometimes the robot does not reach the given distance. Thus it may happen that the while loop may never end. What we found out is that the motors may gat stalled and never finish the loop. The fix to this is to also terminate if the motor has reached the angle minus a delta or if the gyro angle speed is 0. However, again we noticed that many times the gyro is not returning 0, but instead -1, so when we checked for the angle it also would not reliably terminate.

We would like to see a better discussion of this issue in the manual as this is a feature that is not commonly discussed.

Light Sensor Blackout:

(+) GUI: Although this error occurs also on the GUI version it seems that most programs that use a light sensor can recover from it quite easily

(-) Python: Python is more strict and when we expect an integer but recieve an error during the reading of an unknown type, programs will no longer work – we must write a special light sensor function that ignors this error and instead return a previous value

Program Loading Time:

(-) GUI: the loading time is slow when the programs are big

(-) Python3: The loading times seem even slower than using the GUI

(-+) ev3dev micropython: Loading times seems slow, but ok

(+) pybricks-micropython: Loading times are reasonable

Times (need to be verified):

- python3: 35 seconds
- EV3DEVr micropython: 15 seconds
- pybricks micropython: 10 seconds
- Bluetooth copy: 15 seconds
- wireless copy: 3-5 seconds

Thread Support:

(+) GUI: Threads are clearly better supported in the GUI via myblocks. Alone the graphical representation helps.

(-) Python micropython: Threads do not seem to be properly supported. The official version of micropython points this out in its release notes. This should be made available.

OSX Bluetooth:

(-) GUI: We had issues with reliability of the initial connection in macOS making bluetooth unusable for us. We verified this on different bricks and computers

(+) Python: other than sometimes having to reboot the brick multiple times, bluetooth works much better in ev3dev

Documenting the Code:

(+) GUI: The gui has some advanced features for documenting the code that are not available in Visual Code

(++) Python: code documentation can be done in the source code and is easy to do

(-) Python Sphinx: Python has superior functionality while using for example to document the code in Sphinx. However to enable this no proper documentation is provided or discussed in detail as far as we can tell. The coach wrote a code and Makefiles that allow the creation of the library in Sphinx. However there was not enough time to teach the team how to do advanced features such as autodoc, code highlighting and inclusion and how to structure the document. We plan to do this in a future activity. However the programs and the contents have been created from templates that were provided as educational component given to the team. Using Sphinx did provide an advantage as the documentation nad code can be *snapshotted* easily and updates can be communicated quickly. We recommend that LEGO provides time to integrate such documentation feature ability into their upcoming documentation

(-) Python Sphinx on the brick: The coaches experimented with generating the documentation on the brick directly, but it was just too slow, so a way was developed on how to generate them on a laptop. This is beyond the need to know for the team.

However the documentation can easily be created with

make html make pdf

These commands are executed by the coach on regular basis and not the team at this time. The commands create html and pdf documents

(-) Google docs for python code documentation: Due to the advanced features of sphinx it seems cumbersome to use google docs if a system such as sphinx is able to generate a sophisticated documentation that fosters easier learning achievements.

SSH Key Management and Config:

As the robots are in a secure area, they were not allowed to be put on WIFI. However, as we used Bluetooth we could overcome this issue. All robots were set up with ssh keys and ssh configs have been created to more easily log into the robots and identify them by color. This has been set up by the coach without input or knowledge of the team.

This allows the team member to simply type

ssh blue

to log into the blue robot for example

The resources provided by LEGO do not adequately describe how to change hostnames or how to setup ssh configurations while levaraging ssh-add. Naturally at this time this is a feature that is beyond the scope of a team. Instead LEGO could contribute programs that make the management of such tools trivial such as a commandline tool

mindstorm secure setup

or a button in Visual code that does this so inexperienced teams can also leverage this.

CHAPTER

TWO

PYTHON ESSENTIALS

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 refer to 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

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

2.5 Conditions

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

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

2.6 Loops

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

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 Exceptions

When working with the Mindstorm sensors we sometimes find that the sensors do not work properly and return no result. Python has a special mechanism for this that is called try/except. Let us illustrate this.

Let us simulate a sensor with a fault that we can set that returns an error if we pass the parameter value 1 but returns its value for all other inputs.

```
def sensor(number):
    if number == 1:
        rasie ValueError # this just creates an error
    else
        return number
```

Now we can simulate a faulty sensor and deal with its exceptions. Let us test the sensor in a loop such as

last_value=0 # we set a last value for number in [0,1,2]:

try: here we try to see if the function works value = sensor(number) last_value = value # stores the last value and when

an exception occurs we read that

print("Success:", number)

except: value = last_value print("Error:", value)

The nice thing with this loop is that not only do we know when there is an error, but we correct the error with just the last value we found

The result is

:: Success: 0 Error: 1 Success: 2

This is naturaly helpful in cases of the Light sensors, when once in a while the light sensor value dos not return properly.

2.8 Function as a parameter

The Mindtsorm GUI has a convenient Wait method and loop exits that probe certain conditions. Python does not directly provide them, but allows you to create loops.

Instead of just testing for a condition such as introduced in the previous sections, we can also use a functionname as a parameter.

Let us demonstrate and assume that the function

- motor.angle() - returns the angle of the angle of the gyro

we can now create a test function such as

```
def run_for_a_distance(): # is true for running
    return motor.angle() <= 1000
```

This allows us now to define a function that contains a loop to which we pass the running() condition:

```
def followline(speed, until=None):

    while until():
        print("I am following the line ")
        time.sleep(0.1)
```

Now we can call it just as follows

```
followline(25, run_for_a_distance)
```

The convenient thing now is that we can create other functions so we do not have to rewrite the function that loops but just change the termination function such as

```
def run_till_black(): # is true for running
    return colorsensor.reflection() > 10
```

and run it with

```
followline(25, run_till_black)
```

2.9 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
```

2.10 Program

A program can be executed in a terminal on the 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 is 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 simply define them.

CHAPTER

THREE

LESSONS

Note: PLEASE NOTE THAT ALL LIBRARY CLASSES METHODS AND FUNCTIONS HAVE PREVIOUSLY BEEN DEVELOPED OVER A PERIOD OF 2 YEARS BY THE

SPOCKBOTS TEAM

USING THE MINDSORM GUI. THIS DOCUMENT HAS BEEN CREATED TO DEMONSTRATE HOW THIS LIBRARY WAS CONVERTED.

Note: However, some ideas were discussed and explicitly targeted as lessons, such as how to design a more reliable Gyro reset, and how to deal with the different black/white values when using more than one color sensor.

We found that through structured activities such enhancements are possible to be developed by the team. Guidance from the coach was integrated in the solutions. Just as a teacher points out wrong problems to solutions it is important for the coach to provide suggestions for debugging the code. Which we have done. However, as explained in the pitfalls of python some aspects are currently beyond the needed scope for the teams using Python and we encourace LEGO to improve their library with feedback from those having used Python on the EV3 extensively

3.1 Python Programming Templates and Activities

The activities conducted by the team were centered around a number of coordinated educational interactions with a subset of team members. Due to extensive travel and time constraints of some team members and focus on other portions they could not participate in these activities.

As a coach I put together this material so that they or even other teams can benefit form this information. For example I am planning to visit the regional Elementary School to discuss expansion and potential adoption of the material into local schools. The reason we target elementary schools is that we believe this material can be taught on that grade level with the help of a qualified teacher.

3.2 Color Sensor

Goals:

- Learn about classes
- Learn how to read and write files
- Learn how to use try/except

- Learn how to pass paramaters to an method on initialization
- Leran how to use vriables start with *self.* in other methods
- Learn how to map values between 0 and 100

Problem: In our first exercise we are exploring the color sensor. Often you only have a single color sensor, but when looking at the different sensors in your kits, they may all behave slightly differently. What registers as black 8 on one sensor may register as black 15 on another. The same is valid with white, we may see values such as 80, 100, and so on.

However we know that the color values are always between 0 - 100. With one significant issue. Once in a while no value is returned and if we were to expect an integer we need to make sure that Python can react towards such errors.

As we also want you to learn about classes in Python we provide you with a template that you need to complete. Please revisit our Python material and look at how we define classes. Now let us define a class for a *better* Color sensor and we call this class

SpockbotsColorSensor

We do the usual things such as importing the needed abstractions from the pybricks and python libraries. We have provided this as an example for you.

We also demonstrate how to initialize the color sensor while you can pass along simply the port number so you can reflect once more on which conditions work. The keyword *self* can be most simply understood while knowing that programmers do not like to always write the class name, so they use self in this particular case.

The init class also contains a number of variables that we can access within other methods defined as part of this class. We also showcase yohow to define simple methods while leveraging other variables such as the *self.sensor* we define in the init method. In the class methods *reflection()* and *color()*. What would happen if the color sensor cuts out or does not read a value. To avoid this case we simply put a loop around the method that returns the value. We simply try if it works and if it does, then return a value if not it tries it again.

We store the minimum black and maximum white values that we are detected under the current light conditions.

You have the following tasks:

1. Understand the set_white function and implement an equivalent black function
2. Implement a function value() that always returns the mapped color values
3. In case we have more than one color sensor its sometimes interesting to see which one is which, thus this function just flashes the sensor by switching to different modes
4. As we have not yet talked about files, we would like to take this example to teach you about how to write files in Python. This is demonstrated in the write function and it shows how to write values to a file. We also provide you with a read function so you know how to read a file. The read function also shows how easy it is to use try and except. In case the file would not be there, and we call read we get an error. When reading the values we must make sure that the order is the same when we write it.

Now after you have written the better sensor class let us see how we use it

The sensor is stored in a directory called *spockbots* that has a file __init__.py in it this allows us to import the sensor class into a program as follows, while reading a value and wait for a while until we read the next value and print it out.

Experiment and move the sensor over various different places to measure the difference.

```
from spockbots.colorsensor import SpockbotsColorSensor
from time import sleep

colorsensor = SpockbotsColorSensor(2) # we assume you put the sensor on port 2

colorsensor.black = 8     # finding the minimum black value you need to do, use port␣
↪view
```

(continues on next page)

(continued from previous page)

```
colorsensor.white = 90   # finidng the maximum white value you need to do, use port␣
↪view

while True:
    value = colorsensor.value()
    print(value)
    sleep(0.5)
```

Next, modify the program to print th color instead of the reflective value.

Here is the template for this assignment to complete the file *colorsensor.py*:

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

        :return: the original reflective lit value without
        """
        while True:
            try:
                return self.sensor.reflection()
            except:
                pass
```

(continues on next page)

(continued from previous page)

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

        :return: the color value
        """
        #
        # how would you write a function for returning always a color value
        # even if the sensor cuts out. see the reflection() method for an example.
        #

    def set_white(self):
        """
        sets the current value to white if its higher than what is stored
        :return:
        """
        value = self.sensor.reflection()
        if value > self.white:
            self.white = value

    def set_black(self):
        """
        sets the current value to black if it is smaller than what is stored
        """
        #
        # PLEASE PUT YOUR CODE HERE
        #

    def value(self):
        """
        reads the current value mapped between 0 and 100.
        :return: returns the reflective light mapped between 0 to 100
        """

        # read the current color value
        # map the value between 0 to 100 while using the minimum black and maximum␣
↪white value
        # Make sure to only return values between 0 and 100 while testing it
        #
        # use the variable v and return it at the end. Remember functions can return␣
↪values

        return v

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

    def write(self):
        """
        append the black and white value to a file
```

(continues on next page)

(continued from previous page)

```
        """
        f = open("/home/robot/calibrate.txt", "w+")
        f.write(str(self.sensor.black) + "\n")
        f.write(str(self.sensor.white) + "\n")
        f.close()

    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")

    def info(self):
        """
        prints the black and white value read form the
        sensor
        """
        #
        # write a print statement that prints out the information for this color␣
↪sensor such as
        # port, black, and white
        #
```

3.3 Three color sensors

Now we have a beautiful example for a Python class in our color sensor. The next lesson will introduce you to how you can use the same class to define a new one that includes a number of colorsensors. We specify the ports simply as a list at time of creation. So our goal is to do something like

```
colorsensors = SpockbotsColorSensors(port=[2,3,4])
# drive over the black line
# and find the black white values for all sensors
colorsensor.calibrate(port=[2,3,4])
colorsensors.write(port=[2,3,4])
```

Now we can use it in a program as follows to print repeatedly the values from all sensors every half second

```
colorsensors = SpockbotsColorSensors(port=[2,3,4])
colorsensors.read(port=[2,3,4])

while True:
    print (colorsensors.value(2),
           colorsensors.value(3),
           colorsensors.value(4))
    time.sleep(0.5)
```

Here is the template for the multi color sensor class

```
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 = [None, None, None, None, None]
            # in python lists start from 0 not 1
            # so we simply do not use the first element in the list
        # 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:
            print("SETUP COLORSENSOR", i)
            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 the reflective value form the port i

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

(continues on next page)

(continued from previous page)

```
        :param i: number of the port
        :return: The color value, blue = 2
        """
        # return the color value from the port i

    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
        """
        # write the min black and maximum white to a file


    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.
            2: 0, 100
            3: 0, 100
            4: 4, 40    # because it is higher up so white does
                          not read that well
        """
        #
        # loop over the ports and read in the values from the file
        #

    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
        """
        #
        # loop over the porst and flash the color sensor
        #
```

3.4 Driving The Robot

Now it's time to drive around with our robot and our improved color sensors. So what we have to do is simple create a class that includes all the Robot motors and Sensors. So lets get started. First, you must import all the needed classes from pybrics and Python. This includes a long list and you can find them in our template

We simply call the class *SpockbotsMotor*. We define in that calss basic parameters such as wheel size Naturally, we need a left and right motor, but also want to access the motor as part of a tank to do steering just the same way as we do it in the GUI version. In addition we need to create as many color sensors as your robot has, in case of the Spockbots team they decided to use three.

One function that is not provided by Python is a kill button when something goes wrong. To achieve this we simply create a kill method, that sets a variable called *self.running* to false. This function returns True if the LEFT_UP button

is pressed.

we can then use it in functions in an if condition such as

```
def forward(speed, direction):

    if check_kill_button(self):
        return
```

And if the button is pressed the program running variable is set. Within the function we first check if running is False, we know the button has previously been pressed and thus the check button will be True. The return in the function simply means that you leave the function once it reaches the return. We know this function is not ideal but is good enough for us to try things out and if things do not go well we can at least try to stop the robot. To demonstrate its use we like you to take a look at the sleep function. naturally we do not like to sleep if the button has been pressed. This is just how we use it elsewhere. We even can use the check_kill_button in loops to leave the loops when the button is pressed.

The setup method includes all the motor variables so we have values such as self.left, self.right, and self.tank That we can use in the robot.

Sometimes programmers like to make things simple. As writing *self.colorsensors.value(port)* to get the refelctive value on the given port it seems more convenient to create a method that can abbreviate things such as *self.value(port)*

So insead of writing

```
robot = SpockbotsMotor(direction="backwards")
light = self.colorsensors.value(2)
light = self.colorsensors.color(2)
```

we can simply write

:: light_value = self.value(2) color_value = self.color(2)

Next write a function on how to reset the angle in the left and right motors to 0. This will be useful when we measure the distance traveled.

In our next tasks we will calculate which distance we traveled given an angle from the motor or the rotations. We use the circumferance for this and apply the formula that you need to research.

There are various methods that the spockbots team developed in previous years to be found useful. Reimplement these methods in Python.

```
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
from spockbots.output import led

#########################################################
# Robot
```

(continues on next page)

(continued from previous page)

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


class SpockbotsMotor(object):


    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.running = True
        led("GREEN")
        self.diameter = round(62.4, 3)  # mm
        self.width = 20.0  # mm
        self.circumference = round(self.diameter * math.pi, 3)
        self.axle_track = 140.0 # not used, width between middle of tires
        self.direction = "forward"

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

        self.colorsensors = SpockbotsColorSensors(ports=[2, 3, 4])

        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)




    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_UP in brick.buttons(): # backspace
            self.running = False
            led("RED")
            print("KILL")
            self.beep()
            self.beep()
            self.beep()
            self.beep()
```

(continues on next page)

(continued from previous page)

```
            self.stop()
            self.left_medium.stop(Stop.BRAKE)
            self.right_medium.stop(Stop.BRAKE)
        return not self.running

    def sleep(self,seconds):
        if self.check_kill_button():
            return

        time.sleep(seconds)


    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

        """
        if self.check_kill_button():
            return

        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

        """
        return self.colorsensors.value(port)

    def color(self, port):
```

(continues on next page)

(continued from previous page)

```
        """
        return the reflective color sensor value.

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

        """
        return self.colorsensors.color(port)

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

        """
        #
        # write the function that resets the motor
        #

    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

        """
        # switch on the motor, but use a speed between 0 t 100. as the ev3 function␣
↪require
        # values from 0 to 1000 we simply multiply the speed by 10

    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

        """

        #
        # what is the rotation traveled using a given circumferance in cm
        # 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

        """
        #convert a distance to the angle travelde.
        return distance

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

(continues on next page)

(continued from previous page)

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

        """
        convert  the angle to a distance
        return d

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

        :param brake: None, brake, coast, hold

        """
        #
        # This function just stops all the large motors and waits until the robot no␣
↪longer moves
        #
        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 until the motors are no longer turning.
        """
        # Implement a function that tells if the robot is still, you can use
        # the motor angle or the gyro sensor

        PRINT("Still Stop")

    def turntocolor(self,
                    speed,
                    direction="left",
                    port=2,
                    colors=[6]):
        """
        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

        """
```

(continues on next page)

(continued from previous page)

```
        #
        # write a function that turns while only spinning the right motor till it
        # finds any of the colors in the list
        #

    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

        """
        #
        # write a function that truns while only spinning the right motor till it␣
↪finds a black line
        #

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

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

        """
        #
        # write a function that turns while only spinning the right motor till it␣
↪finds a white line
        #

    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

        """
        #
        # write a method that drives up to a black line while using the front color␣
↪sensors
        #
```

(continues on next page)

(continued from previous page)

```
    def aligntowhite(self, speed, port_left, port_right, white=80):
        """
        aligns with white 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 white: value of white

        """
       #
        # write a method that drives up to a black line while using the front color␣
↪sensors
        #

    def alignonblackline(self, speed, port_left, port_right, black, white):
        # Sandra contribt=uted this code
        # as we drive up to a line, we slighty my drive over it.
        # This method drives bacb and forth to find a better allignment

        self.aligntoblack(speed, port_left, port_right, black)
        self.aligntoblack(-speed, port_left, port_right, black)
        self.aligntowhite(speed/2, port_left, port_right, white)
        self.aligntoblack(-speed/2, port_left, port_right, black)




    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

        """
        #
        # drive forward till the light sensor on the given port returns black
        #

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

        """
        #
        # drive forward till the light sensor on the given port returns white
        #
```

(continues on next page)

(continued from previous page)

```
    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

        """
        #
        # drive forward till the light sensor on the given port returns a color for␣
↪the given list
        #


    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

        """
        #
        # you have decided to have 3 color sensors, write a program that drives over␣
↪the black line to
        calibrate it for balck and white and write the values to a file
        #
```

3.5 Gyro Sensor

Goal:

- Learn about passing functions as parameter (Advanced Python concept)
- Learn how to turn the Gyro more precisely while making corrections
- Learn how to drive forward while minimizing the "jump" when using high speeds in forward
- Learn how to write a simple Gyro straight function similar to a line following function
- Learn how to more reliably reset the Gyro
- Learn how to deal with values missing from the Gyro (same as color values)

Going forward with the robot and turning is an elementary task that needs to be implemented. The robot has two different ways of accomplishing this.

First it can be achieved while probing the motors that store an angle reporting back how much the motor has turned. However, what the Spockbots team found is although the motor forward is convenient, it often does not return the desired result, e.g. when the robot caries a heavy unbalanced load it may turn to the one or other side.

Second, you can use the Gyro sensor that measures the angle and speed the robot turns. While the gyro sensor is not very precise, it allows adequate results. When you experiment with the Gyro sensor you will notice the following

issues

1. when turning it may turn too much as you turn with a speed and braking takes time
2. when starting the robot with highspeed to go forward the robot "jumps" and when dropping often looses its orientation.

So let us discuss how we deal with the issue while using a Gyro Server template that you will gradually improve.

In Python we have these issues

1. Sensor value is not 0 after reset
2. Sensor value drifts after reset as it takes time to settle down
3. Sensor drifts forever and never settles
4. Sensor value is not returned as no value is available from the sensor

You are expected to write a code that fixes this

Tasks and lessons

1. Conceptualize that the robot can go forward and backwards, for this reason the Gyro can count clockwise or counter clockwise. The direction is the same as your robot's direction
2. Conceptualize the angle function and compare it with the Color sensor. The while loop with the try except deals with missing values.
3. (Optional) Please change the code so that instead of looping use the last previous valid angle.
4. Define a reset method that waits till the gyro is still and the angle is 0
5. Develop methods for turning left
6. Develop methods for turning right
7. Integrate the left and right method in a better turn method. This method checs at the end if its at the expected angle, and if not corrects it while moving at a slow speed.
8. Test out your robot to see how accurate the turn is
9. Define a move forward function that avoids the "jump" and making the gyro start problematic Remember sometimes if we move slow we are more precise. Can you accelerate your robot from slow to fast. Use a proportional line following algorithm. You developed that as part of your previous mindstorm GUI library

```
import sys
import time
from time import sleep

from pybricks.ev3devices import GyroSensor
from pybricks.parameters import Direction
from pybricks.parameters import Port
from spockbots.output import led, PRINT, beep, sound, signal


class SpockbotsGyro(object):

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

        :param robot: robot variable that includes robot.tank so we can use steering
        :param port: port number for gyro sensor 1,2,3,4
```

(continues on next page)

(continued from previous page)

```
        :param direction: if front if we drive forward
                          otherwise backwards
        """

        self.robot = robot
        if 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)

            except Exception as e: # the gyro is not attched, please plug it in and␣
↪out
                signal()
                beep()
                if "No such sensor on Port" in str(e):
                    print()
                    print("ERROR: The Gyro Sensor is disconnected")
                    print()
                    sys.exit()

        print("GYRO INITIALIZED")

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

        :return: The angle in degrees
        """
        while True:
            try:
                    a = self.sensor.angle()
                    self.last_angle = a
            return a
                except:
                    print("Gyro read error")
            pass

    def zero(self):
        """
        set the gyro angle to 0
        :return:
```

(continues on next page)

(continued from previous page)

```
        """
        self.sensor.reset_angle(0)

    def still(self, count=10):
        """
        tests if robot does not move for maximum count times and returns when it␣
↪reaches 0
        :return: True if robot does not move
        """
        #
        # write a code that tests if the speed of the sensor is 0
        #

    def reset(self, count=10):
        """
        safely resets the gyro
        """
        #
        # resets the gyro and
        # waits till it is still
        # if it is not still it repeats this maximum count times

    def turn(self, speed=25, degrees=90, offset=None):
        """
        uses gyro to turn positive to right negative to left. As it may turn too much,
↪ it
        will correct itself at a lower speed and turn. As the sensor is accurate to 2
        degrees, we only do the correction if the robot is more than two degrees off.

        :param speed: speed it turns at
        :param degrees: degrees it turns
        :return:
        """
    #
    # Implement this function
    #
    # use the left and right function to make it easier for you

    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:
        """
        #
    # Implement this method
    #

    # remember the function to run is self.robot.on_forever(speed, -speed)

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

(continues on next page)

(continued from previous page)

```
        :param speed: The speed
        :param degrees: The degrees
        :param offset:
        :return:
        """
    # Implement this method
    # compare it to what you implemented in left


    def forward(self,
                speed=10,  # speed 0 - 100
                distance=None,  # distance in cm
                t=None,
                finished=None,
                min_speed=1,
                acceleration=2,
                port=1,  # the port number we use to follow the line
                delta=-180,  # control smoothness
                factor=0.01):  # 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
        :paran finished: a function name passes as parameter that returns True if it␣
↪is
                         supposed to run and False if it is finished.
        Examples:

            gyro.forward(50, distance=30, factor=0.005)

        """

        def forever():
        """
        In case we do not pass a finish function by name we
        just run forever.
            """

        return False

        if finish == None:
            finish = forever

        self.robot.reset()
        self.reset()
        while not finished():

          #
          # complete the body of the loop
          #

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

CHAPTER

FOUR

SPOCKBOTS CITY

CHAPTER

FIVE

SUMMARY

5.1 Summary of Robot Features

5.1.1 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.

5.1.2 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
- We have a special interrupt

5.1.3 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

SIX

SPOCKBOTS ROBOT

6.1 Design

6.1.1 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

6.2 MECAHNICS TBD

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

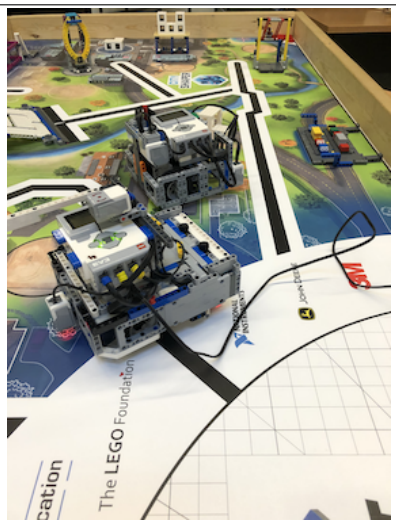

Fig. 2: Robot 2a. Crane setup still works well when the peg points to the line.

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

6.3 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 move with the UP DOWN keys up and down. We leave with the left key and select a program with the right key.

6.4 City Runs

6.4.1 run.red_circle

`run.red_circle.`**`run_red_circle`**`()`
Drive the red peces in the red circle

6.4.2 run.tan_circle

run.tan_circle.**run_tan_circle**()
TBD

6.4.3 run.black_circle

run.black_circle.**run_black_circle**()
TBD

6.4.4 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.colorsensors.read()

    print(robot)

    #
    # setup gyro
    #
    gyro = Gyro(robot)
    gyro.setup()

    """
    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)
    """
```

(continues on next page)

(continued from previous page)

```
    dt = 0.0

    gyro.forward(50, 20)

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

    gyro.forward(50, 5)

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

    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)

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

    # sleep(0.2)

    # back to base

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


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

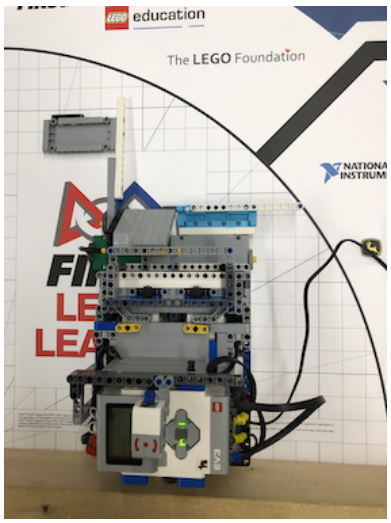

Fig. 4: Crane 1. Crane setup does not have to be precise

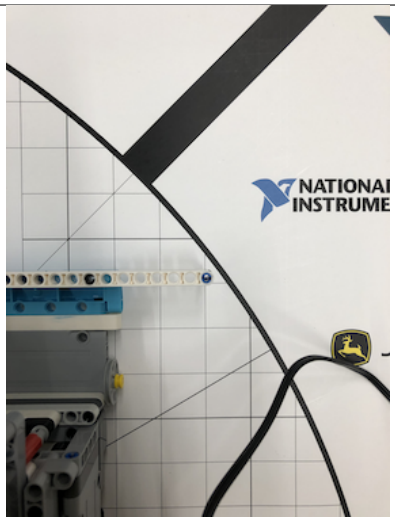

Fig. 5: Crane 2. Crane setup still works well when the peg points to the line.

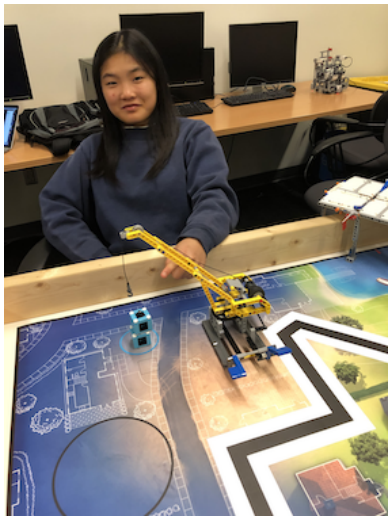

Fig. 6: Crane 4. Successful placement

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.

6.4.5 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.colorsensors.read()

    print(robot)
```

(continues on next page)

(continued from previous page)

```
#
# setup gyro
#
gyro = Gyro(robot)
gyro.setup()

"""
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

gyro.forward(50, 20)

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

gyro.forward(50, 5)

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

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)

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

# sleep(0.2)

# back to base

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

(continues on next page)

(continued from previous page)

```
    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.

6.4.6 run.led

run.led.**run_led**()
TBD

6.4.7 run.turn_to_black module

run.turn_to_black.**run_turn_to_black**()
TBD

6.4.8 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

6.4.9 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

6.5 Spockbots API

6.5.1 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.**signal**(*t=0.05*, *pitch=1500*, *duration=300*)

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

6.5.2 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

6.5.3 spockbots.motor

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

alignonblackline(*speed*, *port_left*, *port_right*, *black*, *white*)

alignstoblack(*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

alignstowhite(*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.

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

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

Returns the reflective color value

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

sleep(*seconds*)

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

turntocolor(*speed, direction='left', port=2, colors=[6]*)
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

6.5.4 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=10*, *distance=None*, *t=None*, *finish=None*, *min_speed=1*, *acceleration=2*, *port=1*, *delta=-180*, *factor=0.01*)

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

Examples:

gyro.forward(50, distance=30, factor=0.005)

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

setup()

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 does not move :return: True if robot does not move

turn(*speed=25*, *degrees=90*, *offset=None*)
uses gyro to turn positive to right negative to left. As it may turn too much, it will correct itself at a lower speed and turn. As the sensor is accurate to 2 degrees, we only do the correction if the robot is more than two degrees off.

Parameters

- **speed** – speed it turns at
- **degrees** – degrees it turns

Returns

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

6.5.5 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

6.6 Spockbots Code

6.6.1 Spockbots Output

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

import os
from time import sleep

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)
```

(continues on next page)

(continued from previous page)

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


##########################################################
# Sound
##########################################################

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):
```

(continues on next page)

(continued from previous page)

```
    """
    The robot will flash the LEDs and beep twice
    """
    beep()
    for color in colors:
        led(color)
        sleep(delay)
    beep()


def signal(t=0.05, pitch=1500, duration=300):
    for i in [1,2,3]:
        led("YELLOW")
        sleep(t)
        led("RED")
        sound(pitch=pitch, duration=duration)

#########################################################
# 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():
```

(continues on next page)

(continued from previous page)

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

6.6.2 Spockbots Check

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


def 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

    :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")
```

(continues on next page)

(continued from previous page)

```
    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)

    PRINT('finished')
```

6.6.3 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
```

(continues on next page)

(continued from previous page)

```
        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()
        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
        """
```

(continues on next page)

(continued from previous page)

```
        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()

    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()
```

(continues on next page)

(continued from previous page)

```
    """

    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 = [None, None, None, None, None]
            # in python lists start from 0 not 1
            # so we simply do not use the firts element in the list
        # 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:
            print("SETUP COLORSENSOR", i)
            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")
```

(continues on next page)

(continued from previous page)

```
            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.
            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("")
```

(continues on next page)

(continued from previous page)

```
        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()
            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].color()
            print("Color sensor", port, v)
        print()
```

6.6.4 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
from spockbots.output import led, PRINT, beep, sound, signal

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

class SpockbotsGyro(object):
    """
    improved gyro class
    """
```

(continues on next page)

(continued from previous page)

```
    # 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

        :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:
                signal()
                beep()
                if "No such sensor on Port" in str(e):
```

(continues on next page)

(continued from previous page)

```
                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("Gyro",  "angle", a, "speed", s)
            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 does not move
        :return: True if robot does not 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
```

(continues on next page)

(continued from previous page)

```
                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
                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 setup(self):
        self.reset()
        if self.still():
            PRINT("ROBOT STILL")
            led("GREEN")
```

(continues on next page)

(continued from previous page)

```
        else:
            PRINT("ROBOT DRIFT")
            led("RED")
            self.robot.beep()
            self.robot.beep()
            self.robot.beep()

    def turn(self, speed=25, degrees=90, offset=None):
        """
        uses gyro to turn positive to right negative to left. As it may turn too much,
↪ it
        will correct itself at a lower speed and turn. As the sensor is accurate to 2␣
↪degrees,
        we only do the correction if the robot is more than two degrees off.

        :param speed: speed it turns at
        :param degrees: degrees it turns
        :return:
        """

        if offset is None:
            if speed == 50:
                offset = 45
            elif speed == 25:
                offset = 17
            else:
                offset = 0

        if degrees < 0: # Turn LEFT

            self.left(speed=speed, degrees=abs(degrees), offset=offset)
            # correct if angle is wrong
            angle = self.angle()
            if abs(angle - degrees) > 2:
                if angle < degrees: # correct if angle is wrong
                    self.right(speed=5, degrees=abs(degrees - angle))
                elif angle > degrees:
                    self.left(speed=5, degrees=abs(degrees - angle))

        elif degrees > 0: # Turn RIGHT

            self.right(speed=speed, degrees=abs(degrees), offset=offset)
            # correct if angle is wrong
            angle = self.angle()
            if abs(angle - degrees) > 2:
                if angle > degrees:
                    self.left(speed=5, degrees=abs(degrees - angle))
                elif angle < degrees:
                    self.right(speed=5, degrees=abs(degrees - angle))

        angle = self.angle()
        print(angle)

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

(continues on next page)

(continued from previous page)

```
        :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()
        angle = self.angle()
        print("LEFT", angle)

    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()
        angle = self.angle()
        print("RIGHT", angle)


    def forward(self,
                speed=10,  # speed 0 - 100
                distance=None,  # distance in cm
                t=None,
                finish=None,
                min_speed=1,
                acceleration=2,
```

(continues on next page)

(continued from previous page)

```
            port=1,  # the port number we use to follow the line
            delta=-180,  # control smoothness
            factor=0.01):  # 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

        Examples:

        gyro.forward(50, distance=30, factor=0.005)

        """

        def forever():
            return False

        if finish == None:
            finish = forever

        print ("GGGG")

        current_speed = min_speed

        if self.robot.check_kill_button():
            return

        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.robot.reset()
        self.reset()
        while not finish():
            if self.robot.check_kill_button():
                return
            value = self.angle()  # get the Gyro angle value

            # correction = delta + (factor * value)
            # calculate the correction for steering
            correction = factor * (value + delta) / 180.0 * 100.0

            self.robot.on(current_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
```

(continues on next page)

(continued from previous page)

```
            distance_angle = self.robot.left.angle()

            traveled = self.robot.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

            # accelerate to make the robot start slowly to not effect the angle
            if current_speed < speed:
                current_speed = current_speed + acceleration

        self.robot.stop()  # stop the robot
        print("GYRO FORWARD TRAVELED", traveled)
```

6.6.5 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
from spockbots.output import led

#########################################################
# Motor
#########################################################


class SpockbotsMotor(object):


    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.running = True
        led("GREEN")
```

(continues on next page)

(continued from previous page)

```
        self.diameter = round(62.4, 3)  # mm
        self.width = 20.0  # mm
        self.circumference = round(self.diameter * math.pi, 3)
        self.axle_track = 140.0 # not used, width between middle of tires
        self.direction = "forward"

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

        self.colorsensors = SpockbotsColorSensors(ports=[2, 3, 4])


    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_UP in brick.buttons(): # backspace
            self.running = False
            led("RED")
            print("KILL")
            self.beep()
            self.beep()
            self.beep()
            self.beep()

            self.stop()
            self.left_medium.stop(Stop.BRAKE)
            self.right_medium.stop(Stop.BRAKE)
        return not self.running

    def sleep(self,seconds):

        if self.check_kill_button():
            return

        time.sleep(seconds)

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

        """
        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)
```

(continues on next page)

(continued from previous page)

```
        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

        """
        if self.check_kill_button():
            return

        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

        """
        return self.colorsensors.value(port)

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

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

        """
        return self.colorsensors.color(port)
```

(continues on next page)

(continued from previous page)

```
    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
        return d

    def stop(self, brake=None):
```

(continues on next page)

(continued from previous page)

```
        """
        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

        """
        if self.check_kill_button():
            return

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

(continues on next page)

(continued from previous page)

```
        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

        """
        if self.check_kill_button():
            return

        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

        """
        if self.check_kill_button():
            return

        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 turntocolor(self,
                    speed,
                    direction="left",
                    port=2,
                    colors=[6]):
        """
        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

        """
        if self.check_kill_button():
            return

        PRINT("turntocolors", speed, direction, port, colors)

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

        while self.color(port) not in colors:
            pass
        self.stop()

    def turntoblack(self,
                    speed,
                    direction="left",
                    port=3,
                    black=10):
        """
```

(continues on next page)

(continued from previous page)

```
        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

        """
        if self.check_kill_button():
            return

        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.

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

        """
        if self.check_kill_button():
            return

        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
```

(continues on next page)

(continued from previous page)

```
        :param black: value of black

        """
        if self.check_kill_button():
            return

        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.right.stop(Stop.BRAKE)
                left_finished = True
            if right_light < black:
                self.left.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
        :param port_left: port of left color sensor
        :param port_right: port of right color sensor
        :param white: value of white

        """
        if self.check_kill_button():
            return

        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.right.stop(Stop.BRAKE)
                left_finished = True
            if right_light > white:
                self.left.stop(Stop.BRAKE)
                right_finished = True
        self.stop()
```

(continues on next page)

(continued from previous page)

```
        PRINT("aligntowhite Stop")

    def alignonblackline(self, speed, port_left, port_right, black, white):
        #s
        self.aligntoblack(speed, port_left, port_right, black)
        self.aligntoblack(-speed, port_left, port_right, black)
        self.aligntowhite(speed/2, port_left, port_right, white)
        self.aligntoblack(-speed/2, port_left, port_right, black)




    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

        """
        if self.check_kill_button():
            return

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

        """
        if self.check_kill_button():
            return

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

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

(continues on next page)

(continued from previous page)

```
        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

        """
        if self.check_kill_button():
            return

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

        self.on(speed, 0)
        run = True
        while run:
            value = self.color(port)
            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
        :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

        """
        if self.check_kill_button():
```

(continues on next page)

(continued from previous page)

```
            return

        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:
            if self.check_kill_button():
                return

            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.color(port)
                elif stop_color_mode == "reflective":
                    value = self.value(port)
                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'):
        """
```

(continues on next page)

(continued from previous page)

```
        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

        """
        if self.check_kill_button():
            return

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

        while self.left.angle() < distance:

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

        self.stop()

        for i in ports:
            PRINT(i,
                  self.colorsensors.colorsensor[i].black,
                  self.colorsensors.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):
        if self.check_kill_button():
            return

        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
```

(continues on next page)

(continued from previous page)

```
        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:
            if self.check_kill_button():
                return
            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:
                    break  # leave the loop
                if stop_color_sensor is not None:
                    if stop_color_mode == "color":
                        value = self.color(port)
                    elif stop_color_mode == "reflective":
                        value = self.value(port)
                    print("VALUE", value)
```

(continues on next page)

(continued from previous page)

```
                if value in stop_values:
                    break  # leave the loop
            except Exception as e:
                print(e)
                break
        self.stop()  # stop the robot
```

6.7 lego

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

run.black_circle, 40
run.calibrate, 44
run.check, 44
run.crane, 40
run.led, 44
run.red_circle, 39
run.swing, 42
run.tan_circle, 40
run.turn_to_black, 44

s

spockbots.check, 46
spockbots.colorsensor, 51
spockbots.gyro, 50
spockbots.motor, 46
spockbots.output, 45

INDEX

A

alignonblackline() (*spockbots.motor.SpockbotsMotor method*), 46
aligntoblack() (*spockbots.motor.SpockbotsMotor method*), 46
aligntowhite() (*spockbots.motor.SpockbotsMotor method*), 46
angle() (*spockbots.gyro.SpockbotsGyro method*), 50
angle_to_distance() (*spockbots.motor.SpockbotsMotor method*), 46

B

beep() (*in module spockbots.output*), 45
beep() (*spockbots.motor.SpockbotsMotor method*), 46

C

calibrate() (*spockbots.motor.SpockbotsMotor method*), 47
check() (*in module spockbots.check*), 46
check_kill_button() (*spockbots.motor.SpockbotsMotor method*), 47
clear() (*in module spockbots.output*), 45
clear() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
color() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
color() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
color() (*spockbots.motor.SpockbotsMotor method*), 47

D

distance_to_angle() (*spockbots.motor.SpockbotsMotor method*), 47
distance_to_rotation() (*spockbots.motor.SpockbotsMotor method*), 47
drift() (*spockbots.gyro.SpockbotsGyro method*), 50

F

flash() (*in module spockbots.output*), 45
flash() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
flash() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
followline() (*spockbots.motor.SpockbotsMotor method*), 47
followline_pid() (*spockbots.motor.SpockbotsMotor method*), 47
forward() (*spockbots.gyro.SpockbotsGyro method*), 50
forward() (*spockbots.motor.SpockbotsMotor method*), 48

G

gotoblack() (*spockbots.motor.SpockbotsMotor method*), 48
gotocolor() (*spockbots.motor.SpockbotsMotor method*), 48
gotowhite() (*spockbots.motor.SpockbotsMotor method*), 48

I

info() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
info() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52

L

led() (*in module spockbots.output*), 45
left() (*spockbots.gyro.SpockbotsGyro method*), 50

O

on() (*spockbots.motor.SpockbotsMotor method*), 48
on_forever() (*spockbots.motor.SpockbotsMotor method*), 48

P

PRINT() (*in module spockbots.output*), 45

R

read() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
read() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52

readfile() (*in module spockbots.output*), 45
reflection() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
reset() (*spockbots.gyro.SpockbotsGyro method*), 50
reset() (*spockbots.motor.SpockbotsMotor method*), 48
right() (*spockbots.gyro.SpockbotsGyro method*), 50
run.black_circle (*module*), 40
run.calibrate (*module*), 44
run.check (*module*), 44
run.crane (*module*), 40
run.led (*module*), 44
run.red_circle (*module*), 39
run.swing (*module*), 42
run.tan_circle (*module*), 40
run.turn_to_black (*module*), 44
run_black_circle() (*in module run.black_circle*), 40
run_calibrate() (*in module run.calibrate*), 44
run_check() (*in module run.check*), 44
run_crane() (*in module run.crane*), 40
run_led() (*in module run.led*), 44
run_red_circle() (*in module run.red_circle*), 39
run_swing() (*in module run.swing*), 42
run_tan_circle() (*in module run.tan_circle*), 40
run_turn_to_black() (*in module run.turn_to_black*), 44

S

set_black() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
set_white() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
setup() (*spockbots.gyro.SpockbotsGyro method*), 51
setup() (*spockbots.motor.SpockbotsMotor method*), 48
signal() (*in module spockbots.output*), 45
sleep() (*spockbots.motor.SpockbotsMotor method*), 49
sound() (*in module spockbots.output*), 45
spockbots.check (*module*), 46
spockbots.colorsensor (*module*), 51
spockbots.gyro (*module*), 50
spockbots.motor (*module*), 46
spockbots.output (*module*), 45
SpockbotsColorSensor (*class in spockbots.colorsensor*), 51
SpockbotsColorSensors (*class in spockbots.colorsensor*), 52
SpockbotsGyro (*class in spockbots.gyro*), 50
SpockbotsMotor (*class in spockbots.motor*), 46
status() (*spockbots.gyro.SpockbotsGyro method*), 51
still() (*spockbots.gyro.SpockbotsGyro method*), 51
still() (*spockbots.motor.SpockbotsMotor method*), 49
stop() (*spockbots.motor.SpockbotsMotor method*), 49

T

test_color() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
test_reflective() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
turn() (*spockbots.gyro.SpockbotsGyro method*), 51
turn() (*spockbots.motor.SpockbotsMotor method*), 49
turntoblack() (*spockbots.motor.SpockbotsMotor method*), 49
turntocolor() (*spockbots.motor.SpockbotsMotor method*), 49
turntowhite() (*spockbots.motor.SpockbotsMotor method*), 49

V

value() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
value() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
value() (*spockbots.motor.SpockbotsMotor method*), 49
voltage() (*in module spockbots.output*), 45

W

write() (*spockbots.colorsensor.SpockbotsColorSensor method*), 51
write() (*spockbots.colorsensor.SpockbotsColorSensors method*), 52
writefile() (*in module spockbots.output*), 45

Z

zero() (*spockbots.gyro.SpockbotsGyro method*), 51