**SOW-BKI 115A: Introduction Robotics**

# Report

**PID Controller: Shenghang Wang**
**s1034413**
**shenghang.wang@student.ru.nl**

**Subsumption Architecture: Julian van Bijnen**
**s1037985**
**julian.vanbijnen@student.ru.nl**

**Instructor: Serge Thill**

## 1. Introduction

### 1.1 Simulator features

The left panel on the simulator has a Model browser which I barely use for my project. The top panel has a few icons where you can change the viewing angle and moving the objects. The scene hierarchy next to the Model browser displays all the objects in the scene in a hierarchical way. That is where I can combine two or more objects. I chose to use the API to Python for programming mainly because I never heard about Lua before and the teacher provided us with a communication example between Python and CoppeliaSim.
The main documentation I relied upon:

1. Remote API functions (Python):
   https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm

2. Simple Examples of PID Control:
   https://www.youtube.com/watch?v=XfAt6hNV8XM

3. TCLab PID Control:
   http://apmonitor.com/pdc/index.php/Main/TCLabPIDControl

4. 01: Line-Following Robot | V-Rep Tutorial
   https://www.youtube.com/watch?v=xI-ZEewIzzI

5. CoppeliaSim (V-REP): Creating paths on the floor
   https://www.youtube.com/watch?v=P6-qrxsaChg

6. Exploring the V-REP simulation environment
   https://github.com/AkessonUlrik/ELA406---Exploring-the-V-REP-simulation-environment

7. Python Imaging Library/Editing Pixels
   https://en.wikibooks.org/wiki/Python_Imaging_Library/Editing_Pixels

8. Image Manipulation in Python
   https://www.codementor.io/@isaib.cicourel/image-manipulation-in-python-du1089j1u

9. How to access and edit pixel values in OpenCV with Python?
   http://datahacker.rs/how-to-access-and-edit-pixel-values-in-opencv-with-python/

### 1.2 Remainder of the report

I ended up using ePuck. It has two differential wheels, one camera, and 8 proximity sensors etc. And I found a good tutorial in which it manually adds additional vision sensors to create a line-following robot. So, my design of PID controller also utilizes the added vision sensors.

## 2. Evaluating a PID controller and a subsumption architecture

### 2.1 Description of PID control

The PID control combines three ways of converting error to command in a system to eliminate error. Generally, the error is the target value minus the current value.
A PID controller is used for maintaining a certain feature of a dynamic system, e.g. keeping a constant speed for a vehicle moving on an uneven terrain.

### 2.2 Description of a subsumption architecture

Subsumption architecture is hierarchical reactive architecture, in which the lowers layers have precedence over layers further up the hierarchy.

### 2.3 Evaluating a PID controller

My initial idea was using summit-xl-ros-stack:

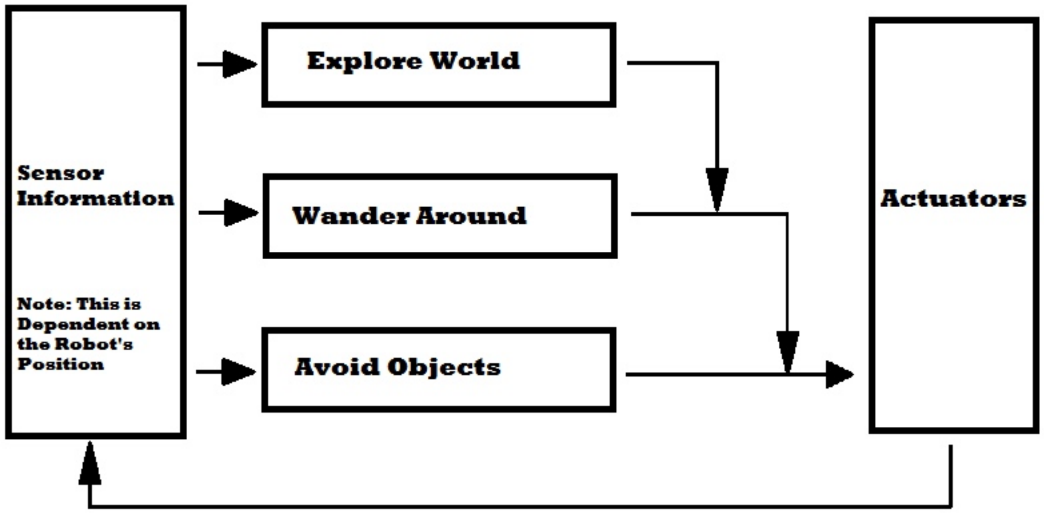
i) Proportional (P)
To increase its speed to 1.5 m/s from a standstill (0.0 m/s) at point A towards the point B which is 150 m away;

ii) Derivative(D)
To stop right at the point B without slipping due to inertia;

iii) Integral (I)
To start from a standstill at the point B moving towards the point C which is 1000 m away, keep increasing the speed (accelerating) till 50 meters away from the point B, then keep the speed moving for 20 seconds.

For Proportional(P) task we can take set error as target velocity minus current velocity, that way it would reduce the error linearly to reach the target velocity.
For Derivative(D) task we can set the error as the distance between the point B and the robot; since we need to reduce the velocity before reaching the B point to avoid slipping, we need to slow down before the point B. The derivative of the error would be negative since the robot is approaching the B point. That way it can constrain the Proportional controller from accelerating the robot too much.
For Integral(I) task we can set the error as 50 minus the distance between the robot and the point B; since after the robot reaches the point that is 50 m away from the point B, it still needs to maintain its speed at that moment moving for another 20 seconds, with the PD control the speed would drop after the error becomes 0, we need the Integral control here to maintain the speed.

### 2.4 Evaluating a subsumption architecture



Brooks, Rodney (1999). Cambrian Intelligence: The Early History of the New AI. Cambridge, Massachusetts: The MIT Press. ISBN 978-0-262-02468-6.

The task is to explore the world by wandering around and whenever there is an obstacle within a certain distance the robot tries to avoid it. The hierarchy of the architecture is showed above.

Firstly, the robot needs to make sure it will not damage itself, thus, the Avoid Objects behavior has to be at the bottom to have precedence over other behaviors. After securing its safety it can Wonder Around so as to Explore world which is our deliberative behavior. The whole reactive hierarchy architecture is demonstrated to be necessary through this task.

## 3. **Implementation**

### 3.1 Chosen tasks

We decided that one person is responsible for the PID controller implementation, another is responsible for subsumption architecture implementation. Julian wanted me to do the PID controller, and I agreed. So, he is responsible for subsumption architecture implementation, I'm responsible for PID controller implementation.
After exploring the supporting resources online for a few days, I changed the task design for demonstrating the PID controller. Since there are more resources online for creating a line-following robot, it would be more realistic for me to work it out at home to make a line-following robot. Thus, I decided to make a line-following robot for demonstrating the PID controller.

### 3.2 PID controller

#### 3.2.1 Implementation

My PID controller updates the velocity of a wheel (left or right) based on the difference between its target velocity and its actual velocity.

```python

class PID:
    def __init__(self, KP, KI, KD):
        self.KP = KP
        self.KI = KI
        self.KD = KD
        self.integratedError = 0
        self.last = None

    def update(self, target, value, dt):
        error = target - value

        self.integratedError = self.integratedError + self.KI * error * dt

        if self.last == None:
            self.last = value

        derivative = (value - self.last) / dt

        self.last = value

        P = self.KP * error * dt
        I = self.integratedError
        D = -self.KD * derivative

        # print("PID", P, I, D)

        return P + I + D

```

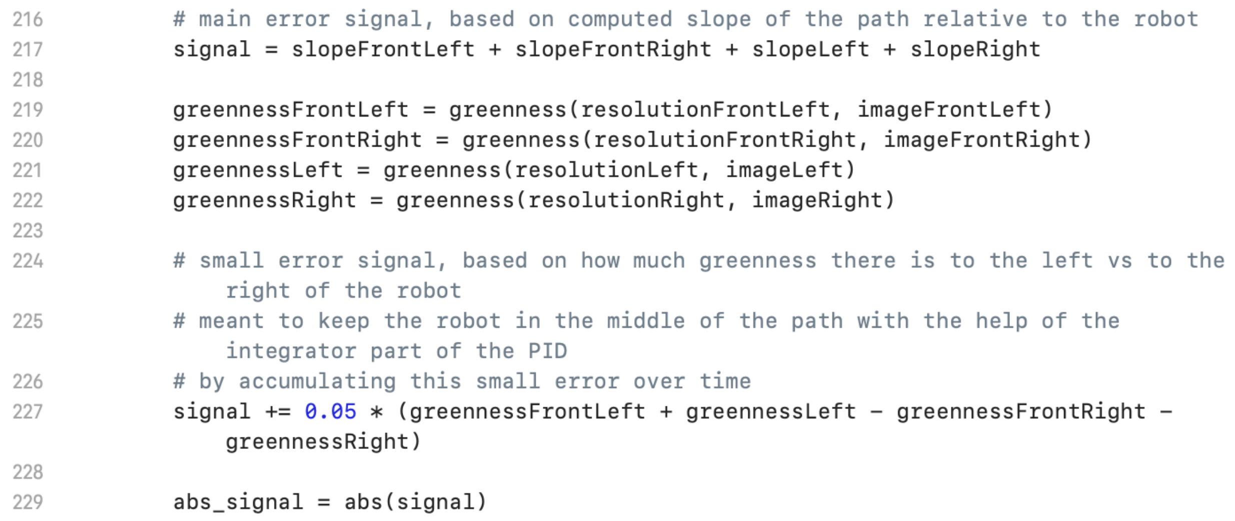
......

```python
        leftVelocity += pidLeft.update(targetLeft, leftVelocity, 0.1)
        rightVelocity += pidRight.update(targetRight, rightVelocity, 0.1)

```

It calculates the target velocity based on the error signal.

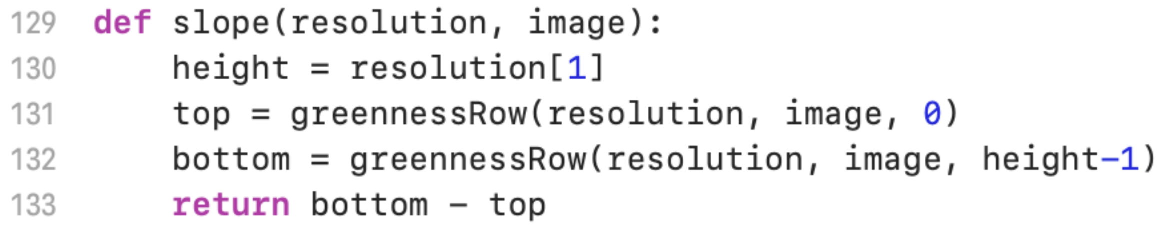
```python
        # damp the speed when the error signal is large, to give the robot time to
            maneuver sharp turns
        damping = 1
        if abs_signal > 0.05:
            damping = max(0.2, 1 / math.exp(abs_signal * 6))
        dampedVelocity = targetVelocity * damping

        correction = 2.5 * abs_signal

        if signal > 0:
            targetLeft = max(0, dampedVelocity - correction)
            targetRight = dampedVelocity
        else:
            targetLeft = dampedVelocity
            targetRight = max(0, dampedVelocity - correction)

```

The error signal is composed of two parts – the main error signal and the small error signal.

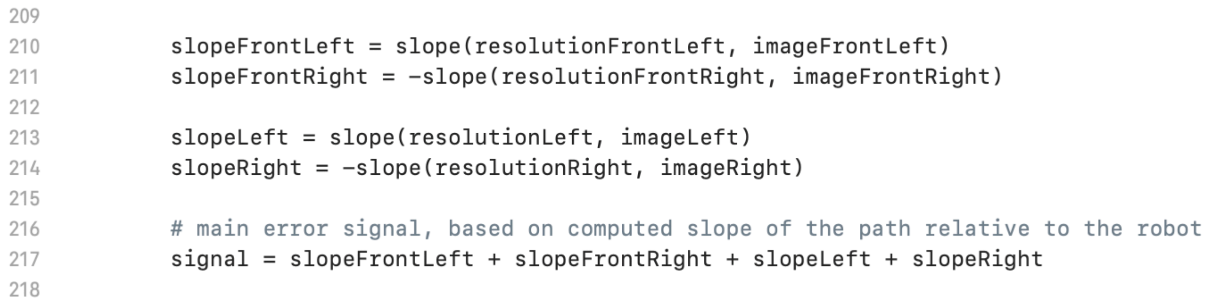


```
        # main error signal, based on computed slope of the path relative to the robot
        signal = slopeFrontLeft + slopeFrontRight + slopeLeft + slopeRight

        greennessFrontLeft = greenness(resolutionFrontLeft, imageFrontLeft)
        greennessFrontRight = greenness(resolutionFrontRight, imageFrontRight)
        greennessLeft = greenness(resolutionLeft, imageLeft)
        greennessRight = greenness(resolutionRight, imageRight)

        # small error signal, based on how much greenness there is to the left vs to the right of the robot
        # meant to keep the robot in the middle of the path with the help of the integrator part of the PID
        # by accumulating this small error over time
        signal += 0.05 * (greennessFrontLeft + greennessLeft - greennessFrontRight - greennessRight)

        abs_signal = abs(signal)
```

The main error signal is based on the calculated slope of the path as seen from the perspective of the robot. If the robot is driving straight along the path and there are no curves, the slope is computed to be zero.

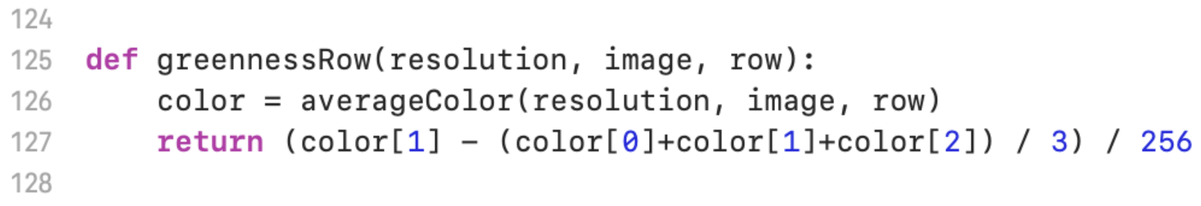


```
def slope(resolution, image):
    height = resolution[1]
    top = greennessRow(resolution, image, 0)
    bottom = greennessRow(resolution, image, height-1)
    return bottom - top
```



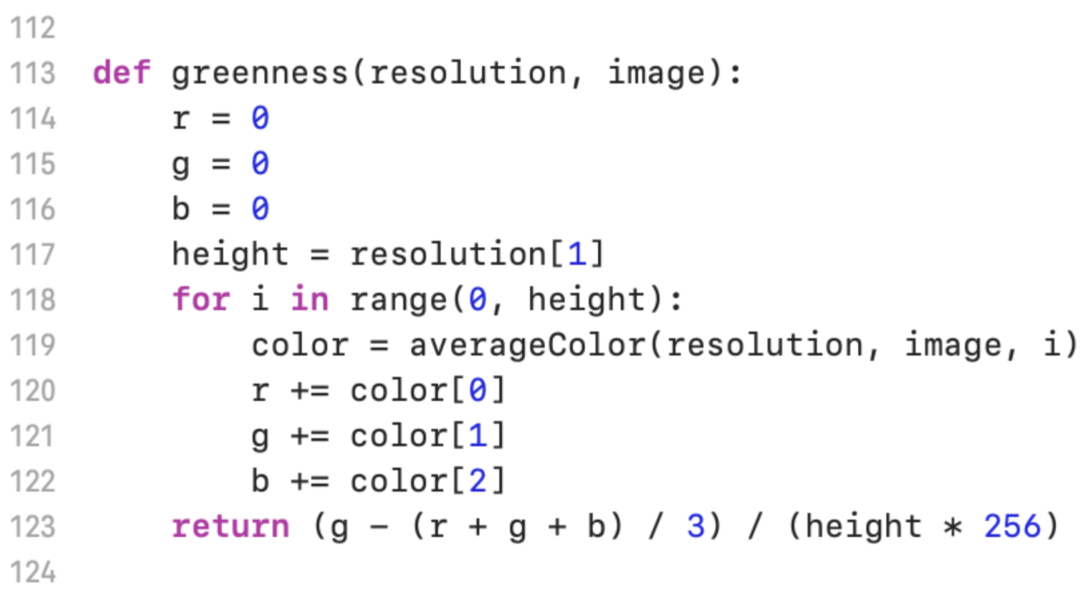
```

        slopeFrontLeft = slope(resolutionFrontLeft, imageFrontLeft)
        slopeFrontRight = -slope(resolutionFrontRight, imageFrontRight)

        slopeLeft = slope(resolutionLeft, imageLeft)
        slopeRight = -slope(resolutionRight, imageRight)

        # main error signal, based on computed slope of the path relative to the robot
        signal = slopeFrontLeft + slopeFrontRight + slopeLeft + slopeRight

```

The greennessRow the slope calculation needs is obtained this way:



```

def greennessRow(resolution, image, row):
    color = averageColor(resolution, image, row)
    return (color[1] - (color[0]+color[1]+color[2]) / 3) / 256

```

The small error is based on the greenness. If there is an equal amount of green to the left and to the right of the robot the small error is zero.



```python
def greenness(resolution, image):
    r = 0
    g = 0
    b = 0
    height = resolution[1]
    for i in range(0, height):
        color = averageColor(resolution, image, i)
        r += color[0]
        g += color[1]
        b += color[2]
    return (g - (r + g + b) / 3) / (height * 256)
```

Throughout the project I calibrated the controller through trial and error. For example, when I tried to determine the calculation for damping velocity, I knew the robot needed to slow down when the error signal was large, or it would run out in the curve. I figured some exponential damping based on error signal might be reasonable. So that if the error signal is small there is not much damping, but as soon as the error signal starts growing the damping will grow a lot. I guess a linear damping response would also work though I did not try. And importantly, it should not completely stop the robot, so I set the lower limit at 0.2.

The biggest challenge I encountered was to read and analyze the pixels, so as to utilize them to update the velocity of a wheel.

### 3.2.2 Performance

```
pidLeft = PID(4.0, 0.1, 1)
pidRight = PID(4.0, 0.1, 1)
```
It goes crazy jumping around;

```
pidLeft = PID(4.0, 1, 0.01)
pidRight = PID(4.0, 1, 0.01)
```
It works but going fast and slow, not smoothly;

```
pidLeft = PID(20, 0.1, 0.01)
pidRight = PID(20, 0.1, 0.01)
```
It goes crazy the same way as the big D value above;

```
pidLeft = PID(0, 0.1, 0)
pidRight = PID(0, 0.1, 0)
```
It doesn’t work, it goes too far during one correct, thus runs off the track;

```
pidLeft = PID(4, 0, 0)
pidRight = PID(4, 0, 0)
```
With I and D set to 0 it still works, though it’s not very smooth, and not good at staying in the middle of path.

```
pidLeft = PID(2, 0, 0)
pidRight = PID(2, 0, 0)
```
It works and quite smoothly, but not good at staying in the middle of the path;

```
pidLeft = PID(5, 0.2, 0.2)
pidRight = PID(5, 0.2, 0.2)
```
It goes crazy badly;

```
pidLeft = PID(5, 0.2, 0.02)
pidRight = PID(5, 0.2, 0.02)
```
This set seems the best tune according to my limited experiments;

```
pidLeft = PID(2.5, 0.1, 0.01)
pidRight = PID(2.5, 0.1, 0.01)
```
This set works at the beginning, but later it runs off the track;

```
pidLeft = PID(4.0, 0.1, 0.01)
pidRight = PID(4.0, 0.1, 0.01)
```
This set makes it run slightly slower, so it looks more secured, but less smooth.

The integrator is good at accumulating small errors over time and correcting them; the derivative controller prevents too hasty corrections; the proportional controller is essential. My PID controller works quite well with the gains I found after many times of experiments. As recorded above there are a certain range of values for the gains for the PID controller to work properly. Out of that range, it would not work properly.

## 3.3 Subsumption Architecture

### 3.3.1 Behaviors

The main behaviour of the robot is 'mow the lawn'. This exists of two separate behaviours: 'mow' and 'navigate'. 'Mow' consists of three parallel behaviours: measuring the length of the grass, comparing this with the desired length of the grass and turning on and off the mow function. In the simulator, long grass is represented by green rectangles and instead of actually mowing, the activity (mowing or not mowing) gets printed in the console. 'Navigate' consist of three behaviours: drive, measure distance and turn. The main behaviour is drive, but at the same time it measures the distance to any possible objects. Once it gets too close, it turns, drives a little bit and turns again, to continue mowing the next stroke of grass. In the simulator, the borders of the grass are represented by walls. The only existing problem is that regularly, the robot turns in the wrong direction.

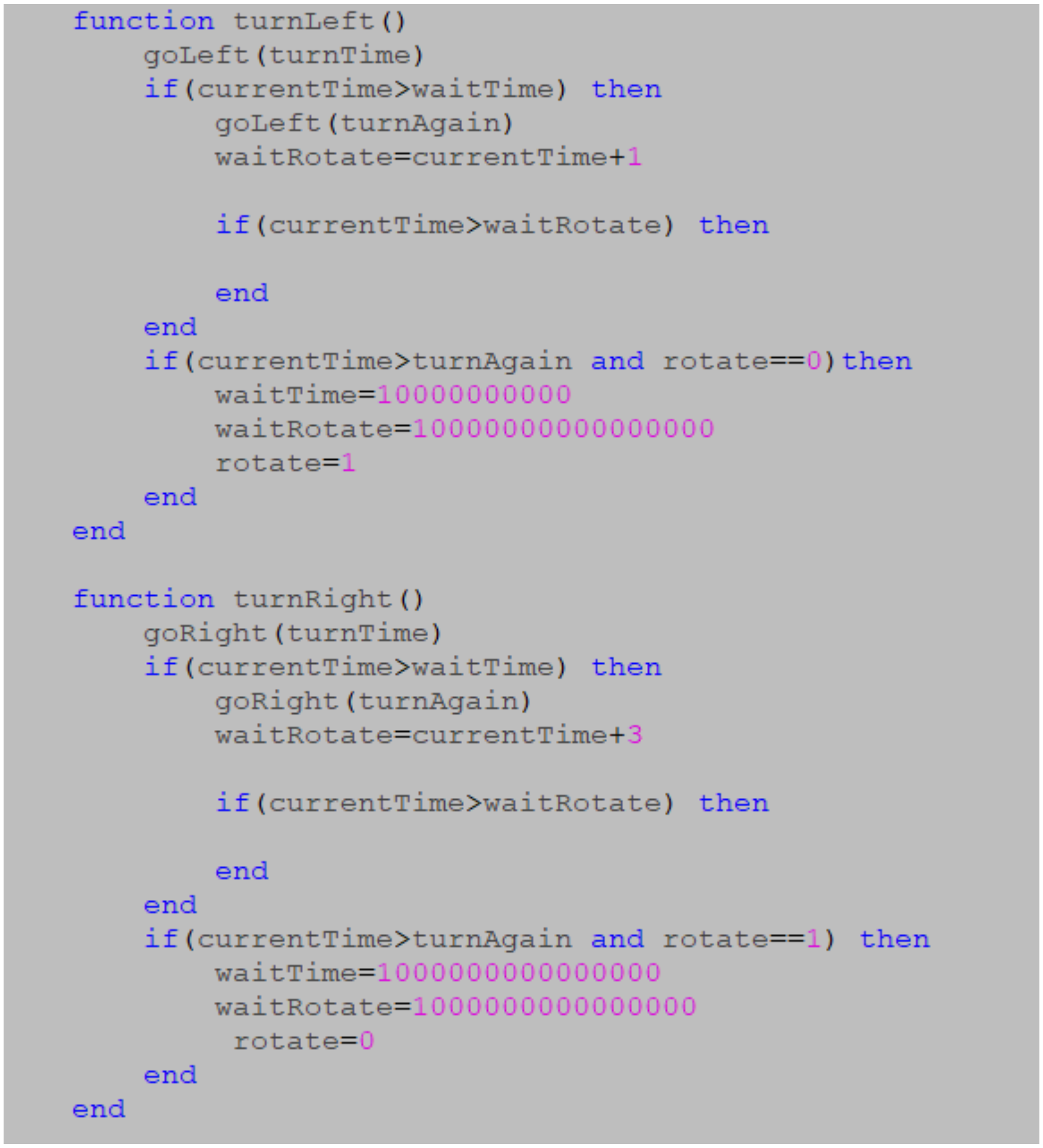
### 3.3.2 Implementation

```
function getDistance(sensor)
    local detected, distance
    detected, distance = sim.readProximitySensor(sensor)
    if (detected<1) then
        distance=1
    end
    return distance
    end
```

The sensors used are two proximity sensors. One to measure the distance to the ground (the 'grass') and one to measure the distance to the nearest obstacle in front of the robot. They both use a getDistance function, that returns 1 if no distance is measured.

```
function sysCall_actuation()
    currentTime=sim.getSimulationTime()

    distancefloor=getDistance(prox2)

    if(distancefloor<0.101) then
        print("mowing")
    else
        print("not mowing")
    end


    distance=getDistance(prox1)

    if(distance<0.2) then
        turnTime=currentTime+3
        waitTime=turnTime+4
        turnAgain=turnTime+7.2
        decideTime=currentTime+0.1
        end

        if(rotate==1) then
            turnRight()
        else
            turnLeft()
        end
```

The distance to the ground is constantly compared with a set value that represents the desired grass length. If the measured distance is smaller than that value, which means the grass is too long, the robot starts mowing, which in the simulator is represented by printing 'mowing' in the console. Once the measured distance is conform the desired length again, 'not mowing' is printed.
The second proximity sensor is used for the navigate behaviour. The basic result of this behaviour is making the robot drive forward by setting the velocity of the two joints to the same value.
In parallel to this drive behaviour, a proximity sensor constantly measures the distance to the nearest object in front of the robot. If this distance gets so small that the robot would no longer be able to turn properly (0.2m), it starts its turn behaviour. This on its own consists of four behaviours.



```
function turnLeft()
    goLeft(turnTime)
    if(currentTime>waitTime) then
        goLeft(turnAgain)
        waitRotate=currentTime+1

        if(currentTime>waitRotate) then

        end
    end
    if(currentTime>turnAgain and rotate==0)then
        waitTime=10000000000
        waitRotate=10000000000000000
        rotate=1
    end
end

function turnRight()
    goRight(turnTime)
    if(currentTime>waitTime) then
        goRight(turnAgain)
        waitRotate=currentTime+3

        if(currentTime>waitRotate) then

        end
    end
    if(currentTime>turnAgain and rotate==1) then
        waitTime=1000000000000000
        waitRotate=1000000000000000
         rotate=0
    end
end
```

First it turns 90 degrees (this doesn't work exactly as desired, so the robot unfortunately doesn't always turn exactly 90 degrees), then it drives more or less its own length and turns again in the same direction as before. As a result the robot now is in the next 'lane' of the lawn. As last step, a binary value named 'rotate' is changed, to indicate the direction of the last turn. As a result of this, the robot knows which direction to turn next time. Often the robot still turns the wrong way though, so it is not precisely as desired. After this turning behaviour, the robot continues to drive and mow if needed again.
The actual motor commands look as follows

```
function goLeft(time)
    if(currentTime<time) then
        sim.setJointTargetVelocity(leftJointHandle,-54*math.pi/180)
        sim.setJointTargetVelocity(rightJointHandle,54*math.pi/180)
    else
        drive()
    end
end

 function goRight(time)
    if(currentTime<time) then
        sim.setJointTargetVelocity(leftJointHandle,54*math.pi/180)
        sim.setJointTargetVelocity(rightJointHandle,-54*math.pi/180)
    else
        drive()
    end
end

function drive()
        sim.setJointTargetVelocity(leftJointHandle,100*math.pi/180)
        sim.setJointTargetVelocity(rightJointHandle,100*math.pi/180)
    end
```

The biggest challenges were to make the turning go in the right direction, which we did not manage to implement exactly as desired and making the robot turn, drive and turn again, as for a long time either the robot kept turning or refused to drive in between. These issues got fixed properly though.

### 3.3.3 Performance

The individual behaviours do largely work as expected, except for the turning direction as mentioned before. For this reason, the lawn does not necessarily get mown in lanes, but rather randomly. For the rest, all needed behaviours for a lawn mowing robot are in place. The robot still struggles when it gets to a corner though. It doesn't hit the wall, but it does disrupt its direction, which results in a rather random direction of movement. The separate behaviours do actually communicate as desired and do not disturb each other as the overall behaviour is good. Noise has barely any influence on the robot, as it measurements don't need to be 100 percent exact. There is some noise in the motors though, which results in turns that deviate from the desired 90 degrees.

## 4. Reflection and Conclusions

4.1 Implementation

I’m very glad that I have mastered some basic image manipulation technique. Next time I would like to try a advanced robotic arm thinking if I can create a controller which can make the arm play ping-pong with humans.

## 5. PID Controller Demo Explained

In the video, with the PID controller the e-Puck moves along the green track on the ground. It finishes the whole track without running off it. When it approaches to a turn it slows down and makes the turn. Especially at the 90 degrees sharp turn it can manage to pass it without straying.

## 6. Subsumption Architecture Demo Exlained

In the video, the robot starts in a corner of the 'lawn'. The walls represent its borders. When the simulation starts, the robot executes its main behaviours: driving and measuring with its sensors. It will constantly print its mowing behaviour: mowing or not mowing. When it measures 'long grass', represented by green rectangles in the simulator, it changes its mowing status to 'mowing'. When the robot comes too close to a wall, it turns, drives a bit and turns again. This however, does not always happen in angles of exactly 90 degrees. The robot comes to another wall and turns to the other side. When it comes to the next wall however, it turns the same side as the last turn. This will continue in a loop, but as the wheels hit the pieces of grass on the second round, the direction will be disrupted.