Report Robotics Group 16

Nayeong Kim, Chantal van Duin,
Lizzy Grootjen and Zuzanna Fendor

May 2018

Introduction

In this report for the practical assignment for the course Introduction to Robotics, we are going to describe the ecological niche of our robot, explain our design choices and our solutions to the problems derived from different challenges. In the end, we are going to draw conclusions from the performance of our robot and our experience in general.

Firstly, what is robotics and why is this assignment relevant for this course? The definition of a robot that we used is : "a machine that senses, thinks, acts and communicates" (Mel Siegel). The field of robotics therefore is about making machines which are able to act within and interact with its environment. In this assignment we had to build a robot using Lego EV3 Mindstorms parts and program it using Java. The robot should be able to accomplish tasks like line following and object detection in a controlled but still real world environment. It can get input from the environment with sensors like the color sensor, process this information and act upon it using the code we provide it with as a form of a thought process. Arguably, the robot can be deemed capable of communication as it can play sounds and can display other observable behaviour. However, nowadays the meaning of communication in usage of robotics is frequently debated in the field. Nevertheless, according to the definition of Siegel mentioned above, the robot we used during this assignment fulfills every aspect of what a machine should be able to do to be classified as a robot.

How was the objective of this assignment, the introduction to the various aspects of working with robots, fulfilled? The assignment puts a major part of its focus on a line following algorithm and PID-controller. The assignment consists of four challenges which each have a different objective. The objective of challenge one is to combine the use of different sensors to orientate the robot in the environment where it is placed in. The second challenge was about making the PID-controller as smooth as possible while still maintaining a reasonable speed. Another objective of this challenge was to build the design of the robot in such a manner that it is able to go up and down a slope. For the third challenge, color recognition, object detection and the display of behaviour played an essential role. For challenge four the objectives were to make the robot choose and follow a path depending on color recognition and object detection as well as localize, grab and drop objects off at a given location with the use of a gripper.

We worked in a team of four people to complete the assignment. We made a timetable during the first practical session and we divided the main tasks. Each member of the group had the responsibility over one of the main tasks: programming, report, poster and mathematics. The supervisor was supposed to divide the tasks and keep an overview on what everyone is doing. However, we discovered that this system did not work very well especially for the programming and report. So in the end, all group members did a little bit of everything and the different programming challenges were divided among the team members. After getting the PID controller to work, each one of us focused on a different assignment or challenge. We communicated with each other when problems occurred and tried to provide each other with new perspectives and insights when one of us was stuck.

Ecological Niche

Challenge 1: Explore your island

The environment for challenge 1 consists of a grid, a maze and two pillars. The grid is a square with a cross in the middle and is made up of a white line on a black background. There are 5 checkpoints in the grid, one at each corner of the grid and one in the middle. The most important aspect for this task is that the robot follows the line well. Our robot uses a color sensor for detecting the light intensity of the surface beneath it. The robot must know which points in the grid it has already passed. In order to do that, it makes use of a gyroscope to detect how many corners it has passed. The task is to visit all checkpoints on the grid, then drives towards the first pillar and react to it. The robot then has to follow the line in the maze, pass the two checkpoints and lastly reach and react to the second pillar.

{Task = line following, Robot = hasColorSensor, Environment = Grid}
{Task = counting corners, Robot = hasGyroscope, Environment = Grid}
{Task = going to the other side of the line , Robot = hasColorSensor, Environment = white line}
{Task = findObject, Robot = hasUltrasonicSensor , Environment = middle of the grid, pillar}
{Task = reactToPillar Robot = hasUltrasonicSensor , Environment = Pillar}
{Task = solveMaze, Robot = hasColorSensor, Environment = Maze}

Challenge 2: Run for your life

For challenge 2, the robot has to follow a line smoothly and cross several obstacles. These obstacles in turn influence the design of the robot.

{Task = line following, Robot = hasColorSensor, Environment = bridge}
{Task = line following, Robot = hasColorSensor, Environment = seesaw}
{Task = cross bridge, Robot = weight, Environment = bridge }
{Task = cross seesaw, Robot = weight, Environment = seesaw }
{Task = cross bridge, Robot = hasWheelAtBack, Environment = bridge }
{Task = cross seesaw, Robot = hasWheelAtBack, Environment = seesaw }

Challenge 3: Friend or Foe

In this challenge, the robot is placed in a field where friends and foes are located. The robot should not walk outside the field. When the robot detects a pillar, it should drive towards the pillar and then detect whether it is a friend or a foe and show appropriate behaviour.

{Task = findObject, Robot = hasUltrasonicSensor , Environment = field with pillars}
{Task = stayInsideField, Robot = hasColorSensor , Environment = field with white outer lines } {Task = reactToPillar, Robot = hasSpeaker, Environment = field with pillars}
{Task = detectColor, Robot = hasColorSensor, Environment = pillar with colored circle }

Challenge 4: Search & Rescue

wFor challenge 4, the robot is placed on an island where it should drive towards and over a narrow passage till it arrives at a splitting on another island. There it should detect a pillar and determine whether healthy food is on the left or on the right side of the end of the splitting with a blue pillar indicating that the healthy food is on the right side and a red pillar indicating it is on the left side. The robot should then pick the food up and drop it off on the island where the robot was first placed. In addition, the robot can go back to grab additional food.

{Task = go to narrow passage, hasGyroscope, Robot = hasColorSensor , Environment = field with white outer lines}
{Task = line following, Robot = hasColorSensor, hasGyroscope, Environment = white line on bridge and ground }
{Task = detect colour healthy food, Robot = hasColorSensor, hasUltraSonicSensor, Environment = pillar with colored circle from close distance of splitting made of a white line }
{Task = walk to healthy food, Robot = hasColorSensor, Environment = white line }
{Task = pick up food, Robot = hasUltraSonicSensor, hasAGripper, Environment = end of white line and an object }
{Task = drop of food, Robot = hasUltraSonicSensor, hasAGripper, hasColorSensor Environment = pillar on field with white outer lines}

Design considerations

Basic robot design

For our basic robot design, we used a template as a guideline [1]. We chose for this design because it seemed very stable and most weight is located at the back of the robot. However, we did adjust the construction to make the attachment of sensors and the gripper possible. We extended the construction in front of the robot and above the brick. The construction above the brick was needed in order to attach the ultrasonic sensor. The extension in front of the robot was build to attach the gripper and the color sensor. We also added a construction at the back to move the centre of the mass even more to the back to ensure more stability when the robot is on the slopes. The gyroscope is build in the construction below the brick for all challenges.

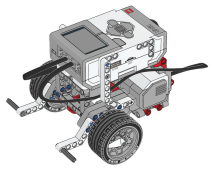

Figure 1: The basic construction for our robot [1]

Challenge 1: Explore your island

For this challenge, our robot has to be able to follow a line, detect objects and detect rotation. Therefore the color sensor, the ultrasonic sensor and the gyroscope but not the gripper is required.

We placed the color sensor in front of the robot right in the middle, where the gripper is placed in challenge 4, as we observed that this location allows for the most precise line following. We also placed the color sensor around 3 cm above the ground because our PID controller works the best at this height. We used the red sample provider of the color sensor in this challenge since color detection is not needed to complete the tasks, only the difference between the white line and the black background.

In order to detect how many corners the robot has passed, we used the gyroscope in our robot. As mentioned, it is placed it in the middle right under the brick. The gyroscope oscillates less when it is placed in the centre of the robot making it more reliable.

For object detection, we use a ultrasonic sensor. We placed it at the front of the robot, approximately at the height of the brick. In this challenge, the sensor does not need to be that low, but we need it at that height for another challenge (challenge 4; search and rescue). We start the object detection in the middle

by the grid. Our robot turns 360 degrees and then searches for the closest object. We chose to place the ultrasonic sensor facing the front of the robot because it was the best choice for the algorithm we used. Another possibility was to place it on the side and then to go around the grid again after visiting the middle of the grid but this seemed less efficient in comparison to our current algorithm.

For the placement of the motors and wheels of our robot, we chose to place two motors with two attached wheels at the front of the robot. In addition the robot has one wheel with a continuous track at the back to ensure stability.

Challenge 2: Run for your life

In order to cross the bridge and the seesaw, the color sensor should not be located too close to the floor. But the color sensor should also not be located too high, otherwise it will be more difficult for the sensor to detect the difference between the white line and the black background. We used the red sample provider of the color sensor in this challenge, since color detection is not needed to complete the tasks, only the difference between the white line and the black background.

We also had to take the placement of the weight of the robot into account. If all the weight is located at the front, the robot will not be able to drive down the seesaw and the bridge correctly. The robot will bend over causing the color sensor to be closer to the line which results in the PID controller not working properly anymore.

In the earlier stages of the design of the robot, we had placed a small wheel at the back of the robot. However when the robot had to get on the bridge and seesaw, this small wheel easily got stuck behind the surface of the bridge and seesaw as there was a gap between the surface and the ground. We solved this by using a bigger wheel at the back which made the robot be able to cross this gap without a problem.

Challenge 3: Friend or Foe

For this challenge, the robot has to be able to detect the pillars with the ultrasonic sensor. In order to detect the pillars, the sensor should not be located too high and also not too low. We put the sensor at the height of the brick, so that the robot can detect the pillars easily.

In order to detect the different colors, the color sensor should not be removed to far from the ground but also not be to close to it. Otherwise the robot has trouble distinguishing between the different colors. We also observed that a fully or almost empty battery can have influence on the precision of the detection of colors. We chose to have the color sensor at the same height as the PID controller/challenge 2 as that seemed to work really well. For this challenge, the robot should also be able to detect white in addition to blue and red because the robot should not go outside of the field. We used the rgb sample provide of the color sensor to detect the different colors.

Challenge 4: Search & Rescue

For challenge 4 there were multiple design considerations that needed to be implemented. Firstly, the robot should be able to grab food with a gripper. We looked at the shape of the provided Lego blocks which represent the food and based our design of the gripper on that shape. The enclosing part of the gripper is designed to be very thin so that it can close around the thinner part of the Lego block while the broader upper part of the Lego block rests on top of the gripper. Another aspect that we took in consideration while designing the gripper was that it also should be compact to match the compactness of the rest of the design of our robot. The gripper is able to close and open by attaching the enclosing Lego part of the gripper to an EV3MediumRegulatedMotor.

We then considered the height on which the gripper should be attached to the robot for it to be able to grab food. To grab food with the gripper, the food first needs to be detectable for the Ultrasonic sensor. In our original design of the robot, after we had only completed challenge 1 and 3, the Ultrasonic sensor was placed on top of the robot, see figure 2. The problem we ran into with the old design was that when the gripper was attached to the robot in a position below the ultrasonic sensor where the ultrasonic sensor was able to detect food, no placement with the height extending blocks made it possible for our gripper to correctly close around the food. In addition, this placement of the gripper compromised the compactness of our robot. So we had to change the position of the ultrasonic sensor to be able to complete challenge 4.

Figure 2: Old robot design

After trying various positions with the gripper, we decided that the best placement of the gripper would be in front of the robot a few centimetre above the ground. This is to ensure that when the gripper has enclosed the Lego block and the upper part rests on top of the gripper, the food will not drag on the ground while driving back to the island. To keep our robot compact, we decided to build in the motor part of the gripper in our construction under the brick and in between the wheels of the robot.

We placed the Ultrasonic sensor just above where the gripper would be positioned on a height where it would not come in direct contact with the gripper but where it still would be able to detect the pillars and the Lego blocks when placed on one height extending block.

The problem we ran into with this design was that before this construction, our color sensor was placed in the middle, where now the enclosing part of the gripper is placed during challenge 4. As an easy solution we tried to attach the color sensor to the side of the robot however we discovered that we could not place the color sensor on its original height that way. This difference made it more difficult for the color sensor to detect the contrast between colors and especially the distinction of the line resulting in a lesser smooth line following performance. To solve this height issue we attached the color sensor to the gripper at the same height it originally was placed.

Solutions/Inventions/Construction/Implementation

Challenge 1

Challenge 1 consists of two parts: the grid and the maze :

- Grid → The robot is placed randomly at one of the possible starting positions near the grid. First the robot has to reach the grid where it turns to the right and after which the gyroscope then resets. The robot goes around the grid by following the outer side of the line until the angle registered by the gyroscope has changed by 360 degrees meaning that therefore all corners have been visited. After the last corner, the robot goes to the inner side of the line. It does this by going straight for a 500 ms after passing the corner. Then it turns 90 degrees to the left, drives forward until the color sensor registers an entirely black surface. It then turns 90 degrees to the right and starts following the line again. One of the problems we encountered was that the robot went back to the outside of the line, even though it was placed on the presumed ideal position for the PID controller. We solved this problem by swapping the motors in the PID controller to reverse the preference for one of the sides. After the inner side of the line is reached and the robot positions itself parallel to the line, the gyroscope resets itself. From this point, the robot follows the line as long as the change in the angle that is registered by the gyroscope is smaller than 170 degrees. We chose 170 degrees and not 180 degrees to ensure that the robot will stop in the middle of the grid even if the gyroscope offset would be a little bit off. When this angle is registered, it means that the robot has reached the middle, where it stops and starts the maze part of the challenge.

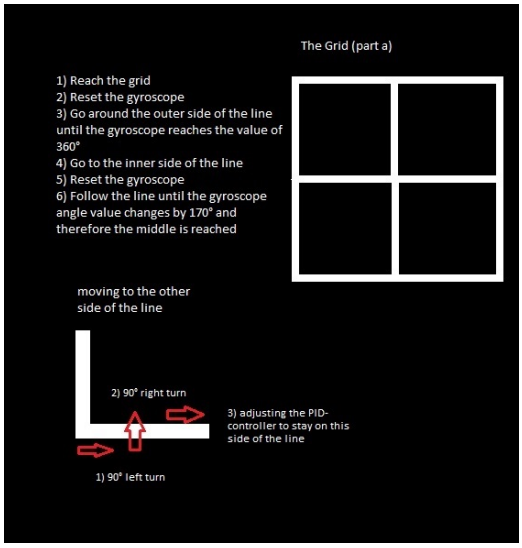

Figure 3: Illustration of the first part of challenge 1

- Maze → The first thing the robot needs to do after reaching the middle is to localize the first pillar. In order to find the nearest object, the robot turns 360 degrees to the left and saves the closest distance to an object. If it sees something which is closer than the initial minimal distance, this initial value gets overridden. After turning 360 degrees, it starts to turn to the right until it detects an object at the same distance to the previously stored nearest distance. The robot then stops turning and drives in the direction of that object/pillar while the distance between the robot and the object is bigger than 20 cm. When the robot reaches the pillar, it makes a sequence of beeps and starts following the line. The important thing for this part of the challenge is that the PID controller is very smooth and that the robot stays at one side of the line. The robot keeps following the line as long as the second pillar

is further than 30 cm. When the second pillar is within that range of 30 cm, the robot stops following the line and drives straight to the pillar until the distance is 15 cm. When this distance is reached, the robot stops and plays the melody from Tchaikovsky's Sleeping Beauty Waltz to show that the second pillar has been found and that the challenge has been finished.

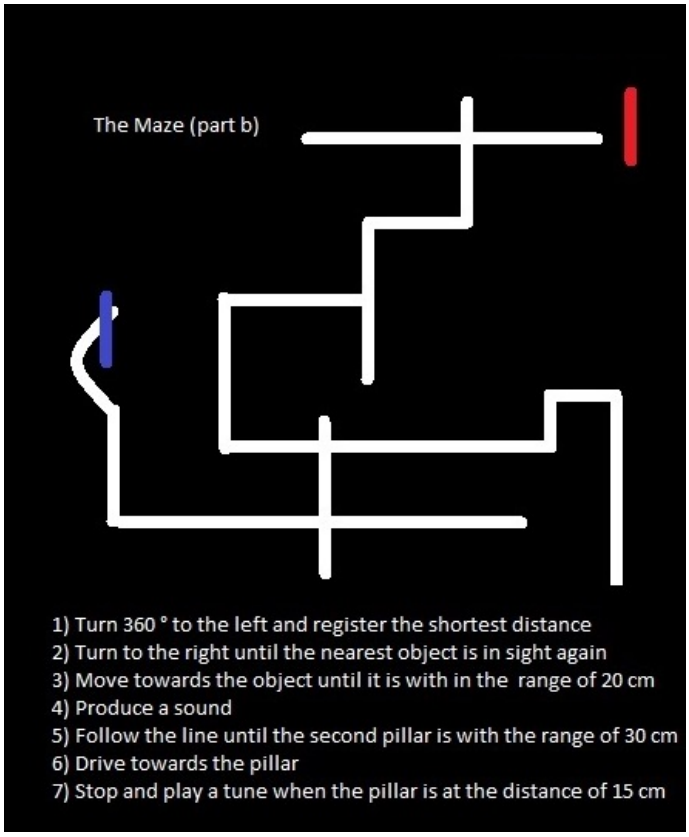

Figure 4: Illustration of the second part of challenge 1

Challenge 2 and 3

For more explanation how we solved challenge 2, see the Line Following Algorithm section of this report. For challenge 3, see the Friend or Foe Algorithm section.

Challenge 4

Our code for completing challenge 4 consists of several parts :

- The island drop off → The robot is placed inside the island, a square consisting of white lines. The robot then drives forward till the color sensor detects the line, which is when the value of the color sensor is equal to the offset value (value where robot detects 50% white and 50% black). When it has detected the line which indicates that the robot has reached the shore of the island, the robot stops and turns 35 degrees to the right and walks a bit forward. This is to ensure that the robot follows the right side of the line. Then the line following part is activated.
- Line following → This part of the code of challenge 4 is consisted of a while loop that loops till 4 pieces of food have been grabbed and have been dropped off on the staring island. The loop contains various

if-statements that activate different functions which will be discussed in more detail below. In the case that none of the conditions of the current input match the conditions of any of the if-statements, the robot will simply follow the white line with the use of the line following algorithm of the PID controller.

- Detection of which side the healthy food is located on → This part of the algorithm is activated in the line following while-loop mentioned above when the ultrasonic sensor detects an object located 30 cm away from the robot and no pillars have been detected yet. When this if-statement is entered, the robot comes to a stop. Since the pillar is located a small distance away from the splitting, the robot walks forward with the use of a counter. After the end of the counter is reached, the robot stops again and detects the color of the patch below the pillar where the robot is now atop of. The color of the patch indicates on which side of the splitting of the white line the healthy food is on.
  The detection of the color is determined by comparing values of the rgb values of the color sensor with beforehand calculated rgb values of the different colors white, black, red and blue. If the color patch is red, boolean healthRed is initialized as true whereas when it is blue the boolean healthyRed is initialized as false. The robot then drives backwards with by counting down the value the previous counter used to drive forward to return to its original position on the line. When it has arrived to its original position, it checks whether healthyRed is true and when this is the case the robot rotates 68 degrees to the left and walks forward so that the robot is located on the left side of the line instead of the right. If healthyRed is false, the robot stays positioned on the right side of the line. The amount of pillars detected is increased by one, the program leaves the if-statement and resumes to the line following algorithm part of this challenge.

- Grab food → This part of the algorithm is activated when the robot currently has no food in its gripper, more than one pillar has been detected and the ultrasonic sensor detects an object located 15 cm away. The robot then follows the line till the object is located 12 cm away where it comes to a halt and activates the gripper. The gripper opens, after which it drives forward after a delay where it stops again to close the gripper. The food is now enclosed in the gripper and then the robot drives backward again to its original position on the line.
  If healthyRed is false meaning that the robot is on the right side of the end of the splitting, the robot will turn 190 degrees to the left so that the robot is once again positioned on the right side of the line, this time facing the island. This is to ensure that the robot will follow the line back to island and not end up at the left end side of the splitting where the poisonous food is located in the case of false healthyRed while following the line. If healthyRed is true meaning that the robot is on the left side of the end of the splitting, the robot can drive back to the island by just following the line since it is already located on the right side of the line of the left side of the splitting with no danger of going to the right side of the splitting to the poisonous food. The amount of food is increased by one, the boolean hasFood is initialized as true, the program leaves the if-statement and resumes to the line following algorithm.

- Drop food off on island → This part of the algorithm is activated when the robot has food in its gripper, more than one pillar has been detected and the ultrasonic sensor detects an object located 30 cm away. When the robot detects the pillar on the island, it walks forward with the use of a counter. It then opens the gripper, dropping the food and then closes the gripper again. The boolean hasFood is initialized as false and the robots turns 180 degrees to the right so that it faces the passage and line again. The island drop off part of the algorithm is activated again, in which the line following part of the algorithm is activated.

Placement Color Sensor

After changing the placement of the color sensor from the middle to the side when attaching the color sensor to the gripper for challenge 4, we noticed that the turns the robot made during the line following with the PID controller were less accurate and less smooth than before. This had to do with the fact that when the color sensor is located to the side, the intensity of turn left of the PID controller should be smaller than the

turn right to compensate for the more left alignment of the color sensor on the robot. However since our color sensor before was located precisely in the middle, we had calculated our intensity of turn left and turn right as equal in our PID controller class.

We did not want to change our values for the intensity in the PID controller class as it could result in a lot of time spent to figure out what the right intensity is to compensate for the left alignment of the color sensor. We thus decided as solution that we would place the light sensor in the middle for every challenge except for challenge 4, since we only needed to use the gripper then. This was possible because challenge 4 does not require a lot of turning during the line following however this is not the case for the other challenges such as challenge 1 where very smooth turning is essential.

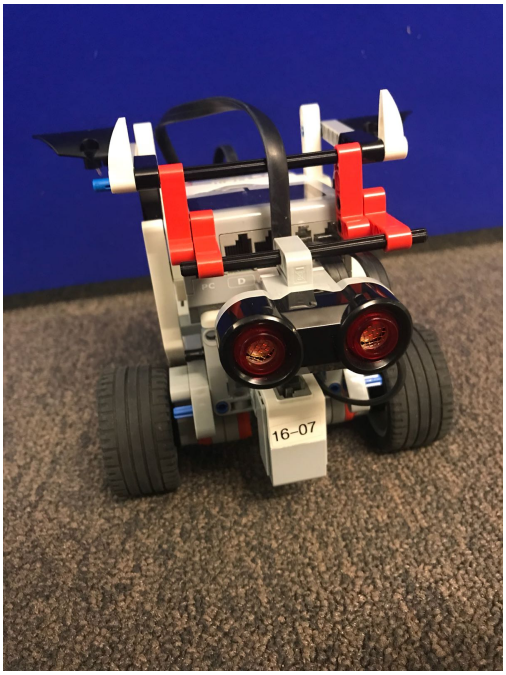

Figure 5: New robot design without gripper

Line Following Algorithm

For our line following algorithm, we created the finite state machine displayed in figure 6. The idea behind the PID controller is relatively simple: you have a baseline value, which is the offset. This is the condition where the robot detects 50% white and 50% black. Based on this condition, the error of the robot can calculated. The error tells how far the robot is away from its base condition. If the error is smaller than zero, then it means that the robot detects more black than white and that it should turn right. When the error is bigger than zero, then the robot detects more white than black and it should turn left. If the error equals zero, then it means that the robot is right on track and does not have to make a turn.

In appendix A, you can see the full PID controller code. We will explain how we implemented the PID controller line by line. In appendix C, you can see the program we use for challenge 2. In the main method, we create the motors, the color sensor and calibrate the robot to calculate the offset at the beginning of the challenge. The offset is the average from black and white, as you can see in the calibrate function. We calibrate the robot by putting it on the white line, we then press the button which makes the code store

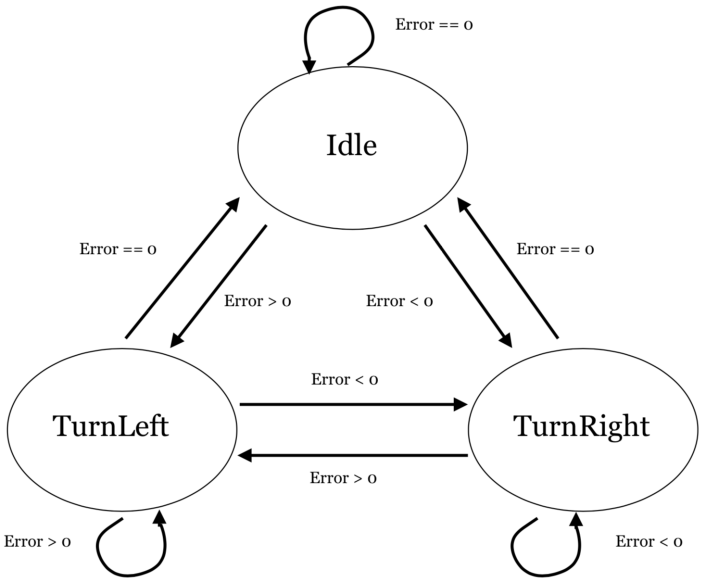

Figure 6: Finite state machine PID-controller

that sensor value. The same is done for black, after which we calculate the offset based on those two sensor values.

```
float offset = (white + black) / 2;
```

In a loop, we call the sensorValue and put it in our PID controller to determine if the robot should turn or not.

```
while (true)
  {
  sensorValue = sensor.redSample()[0];
  control_PID.control(sensorValue);
  }
```

In the constructor of the PID controller class, we save the motors and the offset. We will use these in the `public void control` method. When the function `control` is called, we calculate the error. As mentioned before, the error is the extent to which the robot is on the road. If the error equals zero, then it means the sensorValue has the same value as the offset, meaning that it sees 50% black and 50% white. We then calculate the derivative, which is the actual error minus the last error. From this derivative can be determined how much the robot oscillates between the last error and the actual error, which then can be adjusted.

```
double error = sensorValue - offset;
derivative = error - lastError;
```

After calculating these values, how much the robot should turn can be calculated. We calculate the turn by multiplying the P-value of the robot with the error and adding this to the D-value of the robot times the derivative. This gives the extent to which the robot should turn. We then adjust the power of the motors, for motor A you take the robots speed, T_P, plus the turn and for motor B T_P minus the turn. This way, the the power of one motor will be increased while the power of the other motor is decreased, which causes the robot to turn.

```
double turn = K_P * error + K_D * derivative ;
double powerA = T_P + turn;
double powerB = T_P - turn;
```

We created a function to set the power of the motors, such that one wheel drives backwards if the power is smaller than zero (in the case that is necessary). This function sets the power of a certain motor to the given value. After this we save the actual error as last error so that we can compare the new error to the previous error.

```
setMotor(motorA, powerA);
setMotor(motorB, powerB);
this.lastError = error;
```

We used trial and error in order to find the K_P and the K_D value. We started with only the K_P to try out which value works the best for it and afterwards finding and adding the K_D value. We found out that for our robot the values `K_P = 1500` and `K_D == 100` gives the best results. However, we do not have a clear explanation why our robot works the best with a high K_P and a low K_D.

Friend or Foe Algorithm

In this challenge, you have a reactive agent which has different actions and different states. We can describe our algorithm in an abstract way using the abstract architectures. The states in which our robot can be found are as followed:

$$E = \{onWhite,\ onRed,\ onBlue,\ pillarDetected,\ noPillarDetected\}$$

The robot can perform the following actions:

$$Ac = \{findObject,\ rideToObject,\ turnAround,\ showAggressiveBehaviour,\ showRomanticBehaviour,\ driveForward\}$$

At the beginning of the challenge, the robot is randomly placed inside the field where it has not detected anything yet, so the initial state is `noPillarDetected`.

$$e_0 = noPillarDetected$$

Our state transformer is as followed:

$$\tau : \mathcal{R}^{ac} \rightarrow E$$

Using this information, we can form our environment, our reactive agent and our system $\mathcal{R}$.

$$Env = (E, e_0, \tau)$$

$$Ag : \mathcal{R}^{e} \rightarrow Ac$$

$$\mathcal{R} = (Env, Ag)$$

Our robot is a state-based agent meaning that it first observes the environment then updates the environment state and lastly selects an action. Depending on the internal states, the robot will choose the most convenient action. In the case of this challenge, we want the robot to find 6 pillars. So the internal states D

with D is the number of pillars found is defined as followed :

$$D = \{0, 1, 2, 3, 4, 5, 6\}$$

We programmed our robot that it will continue looking for pillars until it has reached internal state D = 6. Using this information, we can describe our robots' behaviour by a reactive, 'behavioural' approach based on subsumption hierarchy. Figure 7 shows the hierarchy we used for our robot.

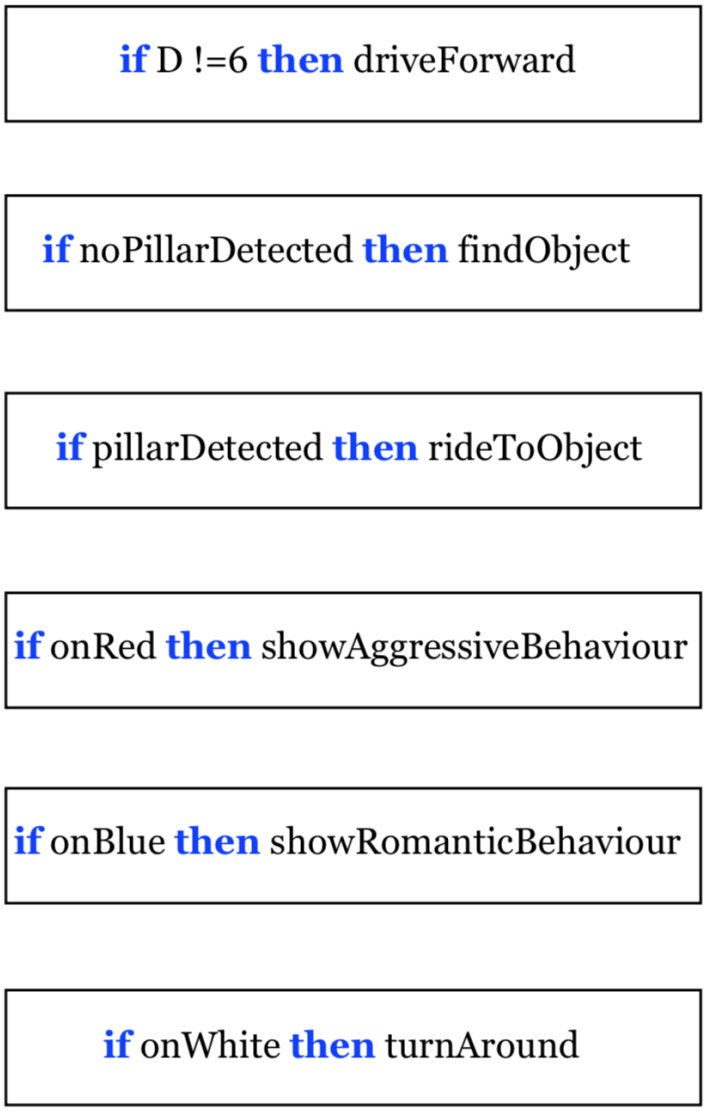

Figure 7: Subsumption hierarchy for the friend or foe challenge

The initial state of our robot is e_0 = `noPillarDetected`. If the internal state D != 6, then the robot will just drive forward. When the robot has not detected any pillar, it should try to find one. When the robot does detect a pillar, it should drive towards it. If the robot detects a red color patch below a pillar then it

should show aggressive behaviour. In the case that the robot detects a blue color patch below a pillar, it should show romantic behaviour. When the robot detects white, it should turn around before doing anything else as it indicates that the robot has reached the border of the field. This explains the low position of the `if onWhite then turnAround` rule.

In this hierarchy, you can see that the conditions in the `if`-statements are all states (either normal states or internal states) and the `then`-statements are the actions the robot can take. You can 'draw' arrows from the different states and actions within this hierarchy. Let's start with our robot's initial state Then you will be in the rule `if D != 6 then driveForward`. Now, suppose the robot does not detect a pillar, then you go to the rule `if noPillarDetected then findObject`. The arrow from the first rule to the second one is the state transformer τ. You go from the driveForward action, which is in this case the $\mathcal{R}^{ac}$ (all the runs up until now ending with an action) to a new state E. The Ag will then be the arrow within an if-else block from a state to an action. You will go from $\mathcal{R}^{e}$ (all runs up until now ending with a state) to a new action, in this case from `noPillarDetected` to `findObject`.

Conclusions

As mentioned before in this report, a good working PID controller is the most essential part of completing the assignment. It is necessary in order to complete challenges 1, 2 and 4. An excellent PID controller not only reduces the chance of the robot losing track of the line, but it also makes the robot oscillate less so that the speed of the robot can be increased. Since the PID controller is the foundation of this assignment, we made sure that it was working as well as possible. By adjusting the height of the sensor and calculating of the best values of the constants we tried to make it work optimally. The PID controller we ended up with, enabled the robot to follow the line and make turns quite smoothly. However, we do not have a good explanation of why a high K_P and a low K_D works so well for our PID controller.

We also discovered that our line following algorithm worked noticeably worse when the battery was low. Sadly we only discovered this in the later stages of working on this project resulting that we wasted some time trying to figure out what was wrong with our code while it would work without a problem if the battery was full. We did not succeeded in solving this issue.

The battery also appeared to be a problem for other types of sensors. The color sensor seemed to suffer the most from this issue. When the battery is low, the color detection becomes less reliable and for example detects a blue surface as black. Also the range of the ultrasonic sensor seems to become smaller when the battery is low.

On the other hand, our system is fairly robust when it comes to slopes and other obstacles. Because of the compactness and the placement of the center of mass, our robot is able to handle most slopes present in the challenges. Sometimes when the slope is very steep, the robot can falter while ascending but in the end the robot almost always makes it to the bottom of the slope. Another aspect of our robot which works fairly well is the showcasing of behaviour.

An aspect that could be improved is the speed of our robot during challenge 2. While it is true that the robot is good at following the line smoothly and that it is not incredibly slow, the speed could be increased at long distances where the line is straight. We tried to make an acceleration function for this, however we found that the performance of the robot became too unreliable. We did not have enough time to test whether this was because of the low energy issue of the battery or that it was because our method for acceleration was just not that good.

During this assignment we discovered that the implementation of a theory does not always work out in practice. Working with robots in a real world environment brings, sometimes unexpected, limitations.

While in theory your algorithm should be working, sometimes external factors like a low battery level or a change in light conditions can cause errors. In conclusion, programming a robot to act in a controlled but still real environment with changing external factors gave us an insight in the tricky parts of robotics and how we can try to limit those sensitive factors.

References

1. https://le-www-live-s.legocdn.com/sc/media/lessons/mindstorms-ev3/building-instructions/ev3-rem-driving-base-79bebfc16bd491186ea9c9069842155e.pdf
2. http://www.lejos.org/ev3/docs/lejos/hardware/Sound.html
3. http://stemrobotics.cs.pdx.edu/node/4576
4. http://legolab.daimi.au.dk/Danish.dir/WRO2016Regular/LeftEdgeDrive/PID%20Controller%20For%20Lego%20Mindstorms%20Robots.pdf
5. Happy tune: Sleeping Beauty Waltz, Tchaikovsky
   https://nl.pinterest.com/pin/430656783089202590/
6. Angry tuen: 5th symphony, Beethoven
   https://www.sheetmusicplus.com/title/beethoven-s-5th-symphony-for-easy-piano-digital-sheet-music/20320000

Appendices

A. Code of the PID controller

```
package nl.ru.ai.chip;
import lejos.hardware.motor.UnregulatedMotor;

public class PIDController {
   public static final double K_P = 1500;
   public static final double K_D = 100;
   private float offset;
   public static int T_P = 42;
   double lastError = 0;
   double derivative = 0;
   int counter = 0;
   UnregulatedMotor motorA;
   UnregulatedMotor motorB;

   public PIDController(UnregulatedMotor motorA, UnregulatedMotor motorB,
       float offset)
       {
      this.motorA = motorA;
      this.motorB = motorB;
      this.offset = offset;

       }

   public void control(float sensorValue)
   {
      double error = sensorValue - offset;
      derivative = error - lastError;
```

```
      double turn = K_P * error + K_D * derivative ;
      double powerA = T_P + turn;
      double powerB = T_P - turn;
      setMotor(motorA, powerA);
      setMotor(motorB, powerB);
      this.lastError = error;
   }

   public void setMotor(UnregulatedMotor motor, double power)
   {
      if (power > 0)
      {
         motor.forward();
         motor.setPower((int) power);
      } else
      {
         power = power * (-1);
         motor.backward();
         motor.setPower((int) power);
      }
   }
}
```

B. Code for challenge 1

```
/**
 *
 */
package nl.ru.ai.chip;

import lejos.hardware.Button;
import lejos.hardware.Sound;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.robotics.Gyroscope;
import lejos.utility.Delay;

public class ExploreYourIsland
{
   public static float black;
   public static float white;
   public static float offset;
   public static UnregulatedMotor motorA;
   public static UnregulatedMotor motorB;
   public static ColorSensor sensor;
   public static Gyroscope gyro;
   public static PIDController control_PID;
   public static UltrasonicSensor uss;
   public static double inf = Double.POSITIVE_INFINITY;
   public static float dist = (float) inf;
   public static float angle = 0;
   public static Tune song = new Tune();
```

```
public static void main(String[] args) throws InterruptedException
{
   gyro = new GyroSensor(SensorPort.S2);
   motorA = new UnregulatedMotor(MotorPort.A);
   motorB = new UnregulatedMotor(MotorPort.B);
   sensor = new ColorSensor();
   offset = calibrate(sensor);
   control_PID = new PIDController(motorA, motorB, offset);
   uss = new UltrasonicSensor(SensorPort.S4);

   Button.waitForAnyEvent();
   // findObject();
   // rideToObject();
   grid();
   search();
   maze();
}

/**
 * walks through the grid
 */
private static void grid()
{
   reachGrid();
   turnToPosition();
   goAround();
   toInnerSide3();
   toMiddle();
   // song.playTune();
}

/**
 * searches for the nearest pillar
 *
 * @throws InterruptedException
 */
private static void search() throws InterruptedException
{
   findObject();
   Sound.setVolume(50);
   Sound.beepSequenceUp();
}

/**
 * follows the line in the maze and stops when it sees the pillar
 *
 * @throws InterruptedException
 */
private static void maze() throws InterruptedException
{
   boolean mazeSolved = false;
   float sensorValue = sensor.redSample()[0];
   while (!mazeSolved)
   {
      sensorValue = sensor.redSample()[0];
      control_PID.control(sensorValue);
```

```
      if (uss.getRange() < 0.30)
      {
         mazeSolved = true;
      }
   }
   while (uss.getRange() > 0.15)
   {
      driveForward();
   }
   System.out.println("maze finished");
}

/**
 * drives towards the grid
 */
private static void reachGrid()
{
   float sensorValue = 0;
   while (sensorValue <= offset)
   {
      driveForward();
      sensorValue = sensor.redSample()[0];
   }
   stop();
}

/**
 * after reaching the grid, turns to the right position, parallel to the line
 */
private static void turnToPosition()
{
   gyro.recalibrateOffset();
   turn90Right();
   findOffset();
}

/**
 * drives around the grid on the outer side of the line
 */
private static void goAround()
{
   float sensorValue = 0;
   while (gyro.getAngle() < 360 && gyro.getAngle() > -360)
   {
      sensorValue = sensor.redSample()[0];
      control_PID.control(sensorValue);
   }
}

/**
 * goes to the inner side of the line and turns to the right postion, parallel
 * to the line
 */
private static void toInnerSide3()
{
   stop();
```

```
   Delay.msDelay(1000);
   float sensorValue = sensor.redSample()[0];
   System.out.println("To inner side!");
   driveForward();
   Delay.msDelay(500);
   System.out.println("Delay's over");
   findOffset();
}

/**
 * walks to the middle of the grid and stops there
 */
private static void toMiddle()
{
   System.out.println("TO THE MIDDLE");
   gyro.reset();
   float sensorValue = sensor.redSample()[0];
   while (gyro.getAngle() < 170 && gyro.getAngle() > -170)
   {
      sensorValue = sensor.redSample()[0];
      control_PID.control(sensorValue);
   }
   stop();
   System.out.println("Middle reached");
}

/**
 * turns until it finds an object within the range
 *
 * @throws InterruptedException
 */
private static void findObject() throws InterruptedException
{
   System.out.println("Finding object");
   while (gyro.getAngle() < 360 && gyro.getAngle() > -360)
   {
      if (uss.getRange() < dist)
         dist = uss.getRange();
      turnLeft();
   }
   while (dist < uss.getRange())
   {
      turnRight();
   }
   System.out.println("last part");
   while (dist > 0.20)
   {
      dist = uss.getRange();
      driveForward();
   }
   dist = (float) inf;
}


/**
 * finds the offset for the gyroscope when the robot stops turning
```

```
 */
private static void findOffset()
{
   float sensorValue = 0;
   while (gyro.getAngularVelocity() != 0)
   {
      sensorValue = sensor.redSample()[0];
      control_PID.control(sensorValue);
   }
   gyro.reset();
   System.out.println("Offset found");
}

/**
 * callibrates the offset
 *
 * @param sensor
 * @return
 */
private static float calibrate(ColorSensor sensor)
{
   white = calColor(sensor, "white");
   black = calColor(sensor, "black");
   float offset = (white + black) / 2;
   return offset;
}

/**
 * callibrates for the white and black surface
 *
 * @param sensor, colorName
 * @return
 */
private static float calColor(ColorSensor sensor, String colorName)
{
   System.out.println("Detect " + colorName);
   Button.waitForAnyPress();
   float color = sensor.redSample()[0];
   System.out.println(color);
   return color;
}

/**
 * the robot stops
 */
private static void stop()
{
   motorA.setPower(0);
   motorB.setPower(0);
}

/**
 * the robot drives straight forward with the speed of 50
 */
private static void driveForward()
{
```

```
      motorA.setPower(50);
      motorB.setPower(50);
   }

   private static void turnLeft()
   {
      motorA.setPower(-30);
      motorB.setPower(30);
   }

   private static void turnRight()
   {
      motorA.setPower(30);
      motorB.setPower(-30);
   }

   private static void turn90Right()
   {
      gyro.reset();
      while (gyro.getAngle() < 90 && gyro.getAngle() > -90)
      {
         motorA.setPower(30);
         motorB.setPower(-30);
      }
   }

   private static void turn90Left()
   {
      gyro.reset();
      System.out.println("gyro resetted");
      while (gyro.getAngle() < 90 && gyro.getAngle() > -90)
      {
         motorA.setPower(-30);
         motorB.setPower(30);
      }
      System.out.println("Turning Left finished (In the function)");
   }
}
```

C. Code for challenge 2

```
package nl.ru.ai.chip;

import lejos.hardware.Button;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;

public class RunForLife2 {

   public static void main(String[] args)
   {
      UnregulatedMotor motorA = new UnregulatedMotor(MotorPort.A);
      UnregulatedMotor motorB = new UnregulatedMotor(MotorPort.B);
      ColorSensor sensor = new ColorSensor();
```

```
    float sensorValue;
    float offset = calibrate(sensor);
    PIDController control_PID = new PIDController(motorA, motorB, offset);

    while (true)
    {
       sensorValue = sensor.redSample()[0];
       control_PID.control(sensorValue);
    }
  }

  private static float calibrate(ColorSensor sensor)
  {
    System.out.println("Detect black");
    Button.waitForAnyPress();
    float black = sensor.redSample()[0];
    System.out.println(black);

    System.out.println("Detect white");
    Button.waitForAnyPress();
    float white = sensor.redSample()[0];
    System.out.println(white);

    float offset = (white + black) / 2;
    return offset;
  }
```

D. Code for challenge 3

```
package nl.ru.ai.chip;

import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;

public class FriendOrFoe3
{

  private static UnregulatedMotor motorA;
  private static UnregulatedMotor motorB;
  private static ColorSensor sensor;
  private static UltrasonicSensor uss;
  private static float[] sensorValue;
  private static int pillarsFound=0;
  private static double inf=Double.POSITIVE_INFINITY;
  private static float dist=(float)inf;
  private static Tune tune;

  public static void main(String[] args) throws InterruptedException
  {
    motorA=new UnregulatedMotor(MotorPort.A);
    motorB=new UnregulatedMotor(MotorPort.B);
    sensor=new ColorSensor();
    uss=new UltrasonicSensor(SensorPort.S4);
```

```
  tune=new Tune();

   while(pillarsFound!=6)
   {
     searchPillars();
   }
}

private static void searchPillars() throws InterruptedException
{
  if(checkColor() == 1)
  {
    System.out.println("I don't want white");
    turnAround();
  } else
  {
    findObject();
    rideToObject();
  }
}

private static void detectColor() throws InterruptedException
{
  switch(checkColor()) {
  case 1:
    System.out.println("I don't want white");
    turnAround();
    break;

  // Red pillar:
  case 2:
    System.out.println("red found");
    // Aggressive behaviour
    tune.playAggressiveTune();
    driveBackwardSlow();
    Thread.sleep(1000);
    driveForwardFast();
    Thread.sleep(1000);
    stop();
    pillarsFound++;
    break;

  // Blue pillar:
  case 3:
    System.out.println("blue found");
    // Romantic behaviour
    stop();
    tune.playHappyTune();
    pillarsFound++;
    break;
  }
}

private static void driveBackwardSlow()
{
  motorA.setPower(-20);
```

```
  motorB.setPower(-20);
}

private static void driveForwardFast()
{
  motorA.setPower(80);
  motorB.setPower(80);
}

private static void findObject()
{
  System.out.println("Finding object");
  float initDist=dist;
  while(dist>=initDist)
  {
    dist=uss.getRange();
    turnLeft();
  }
}

private static void rideToObject() throws InterruptedException
{
  float initDist=dist;
  while(dist>0.15 && dist < 2)
  {
    dist=uss.getRange();
    if(checkColor() == 1) {
      System.out.println("I don't want white");
      turnAround();
      Thread.sleep(200);
      dist=uss.getRange();
      findObject();
    }
    if(dist<=initDist+0.03)
    {
      initDist=dist;
      driveForward();
      System.out.println("Driving the right way");
    } else
    {
      System.out.println("Out of sight, find object");
      findObject();
    }
  }
  stop();
  while(noColorDetected())
  {
    driveForwardSlow();
  }
  stop();
  System.out.println("Color detection");
  detectColor();
  Thread.sleep(4000);
  initDist=(float)inf;
  dist=uss.getRange();
```

```
}

private static void driveForwardSlow()
{
  motorA.setPower(20);
  motorB.setPower(20);
}

private static boolean noColorDetected()
{
  return checkColor() != 2 && checkColor() != 3;
}

private static void turnAround()
{
  int count=0;
  while(count<500)
  {
    turnLeft();
    count++;
  }
}

private static void turnLeft()
{
  motorA.setPower(-30);
  motorB.setPower(30);
}

private static void stop()
{
  motorA.setPower(0);
  motorB.setPower(0);
}

private static void driveForward()
{
  motorA.setPower(50);
  motorB.setPower(50);
}

public static int checkColor()
{
  sensorValue=new float[3];
  for(int i=0;i<3;i++)
  {
    sensorValue[i]=sensor.rgbSample()[i];
  }

  if(sensorValue[0]>0.01&&sensorValue[1]>0.01&&sensorValue[2]>0.01)
  {
    //white
    System.out.println("white detected");
    return 1;
  }
  if(sensorValue[0]>0.01&&sensorValue[1]<0.01&&sensorValue[2]<0.01)
```

```
    {
      //red
      System.out.println("red detected");
      return 2;
    }
    if(sensorValue[0] < 0.01 && sensorValue[1] > 0.007 && sensorValue[2] > 0.005){
      //blue
      System.out.println("blue detected");
      return 3;
    }
    else {
      //black
      System.out.println("black detected");
      return 4;
    }
  }
}
```

E. Code for challenge 4

```
package Scanner;

import lejos.hardware.Button;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.SensorPort;
import lejos.robotics.Gyroscope;
import lejos.utility.Delay;

public class Challenge4 {
   public static float black;
   public static float white;
   public static float offset;
   public static UnregulatedMotor motorA;
   public static UnregulatedMotor motorB;
   public static ColorSensor sensor;
   public static Gripper gripper;
   public static Gyroscope gyro;
   public static PIDController control_PID;
   public static UltrasonicSensor uss;
   public static int pillarFound = 0;
   public static boolean stop ;
   public static int counter = 0;
   private static float[] sensorValue;
   private static float colorValue;
   private static boolean healthyRed;
   private static boolean hasFood;

   public static void main(String[] args) {
      initialise () ;
      System.out.println("Press to start challenge 4");
      Button.waitForAnyPress();
      islandDropOff();
   }
```

```
private static void initialise() {
   motorA = new UnregulatedMotor(MotorPort.A);
   motorB = new UnregulatedMotor(MotorPort.B);
   gripper = new Gripper(MotorPort.D);
   gyro = new GyroSensor(SensorPort.S2);
   sensor = new ColorSensor();
   offset = calibrate(sensor);
   control_PID = new PIDController(motorA, motorB, offset);
   uss = new UltrasonicSensor(SensorPort.S4);
   hasFood = false;
   stop = false ;
   nrOfFood = 0 ;
}

private static void islandDropOff() {
   reachLine();
   turnRight(35);
   WalkForward();
   followLine();
}


private static void reachLine() {
   float sensorValue = 0;
   while (sensorValue <= offset) {
      motorA.setPower(50);
      motorB.setPower(50);
      sensorValue = sensor.redSample()[0];
   }
   stop();
}

private static void turnToPosition() {
   gyro.recalibrateOffset();
   turnRight(90);
}

private static void turnRight(int turnAngle) {
   gyro.reset();
   while (gyro.getAngle() <= turnAngle && gyro.getAngle() >= -turnAngle) {
      motorA.setPower(30);
      motorB.setPower(-30);
   }
}

private static void turnLeft(int turnAngle) {
   gyro.reset();
   while (gyro.getAngle() < turnAngle && gyro.getAngle() > -turnAngle) {
      motorA.setPower(-30);
      motorB.setPower(30);
   }
}

private static void followLine() {
   while (stop == false) {
```

```
      colorValue = sensor.redSample()[0];
      control_PID.control(colorValue);
      if (uss.getRange() < 0.30 && pillarFound == 0) {
         pillarFound++;
         findColour();
         System.out.println("1st pillar detected");
         if (healthyRed == true) {
            turnLeft(68);
            int count = 0;
            while (count <= 550) {
               walkForward();
               count++;
            }
         }
      }
      if (hasFood == false) {
         if (uss.getRange() < 0.15 && pillarFound >= 1) {
            System.out.println(" food detected");
            hasFood = true;
            foodGrapActivate();
            if (healthyRed == false) {
               turnLeft(190);
            }
         }
      }
      if (uss.getRange() < 0.30 && pillarFound == 1 && hasFood == true) {
         dropFood();
         hasFood = false;
      }

      if (nrOfFood >= 4 && hasFood == false) {
         stop = true ;
      }
   }
}

private static void dropFood() {
   int count = 0;
   while (count <= 550) {
      walkForward();
      count++;
   }
   stop();
   gripper.grab();
   Delay.msDelay(2000);
   gripper.close();
    turnRight(180);
   islandDropOff ();
   }

}

private static void findColour() {
   int count = 0;
   while (count < 12000) {
      walkForward();
```

```
      count++;
   }
   if (count == 12000) {
      stop();
      detectColour();
      count--;
      System.out.println("detect colour achieved");
   }
   while (count < 12000 && count > 1000) {
      walkBackward();
      count--;
   }
   count = 0;
}

private static void detectColour() {
    switch (checkColor()) {
   case 2:
      System.out.println("Red: the healthy food is on the left");
      healthyRed = true;
      break;

   case 3:
      System.out.println("Blue: the healthy food is on the right");
      healthyRed = false;

   default:
      System.out.println("You failed to find pillar");
      break;
   }
}

public static int checkColor() {
   sensorValue = new float[3];
   for (int i = 0; i < 3; i++) {
      sensorValue[i] = sensor.rgbSample()[i];
   }

   if (sensorValue[0] > 0.01 && sensorValue[1] > 0.01 && sensorValue[2] > 0.01) {
      // white
      System.out.println("white detected");
      return 1;
   }
   if (sensorValue[0] > 0.01 && sensorValue[1] < 0.01 && sensorValue[2] < 0.01) {
      // red
      System.out.println("red detected");
      return 2;
   }
   if (sensorValue[0] < 0.01 && sensorValue[1] > 0.005 && sensorValue[2] < 0.01) {
      // blue
      System.out.println("blue detected");
      return 3;
   } else {
      // black
      System.out.println("black detected");
      return 4;
```

```
    }
}

private static void foodGrapActivate() {
    while (uss.getRange() >= 0.12) {
        colorValue = sensor.redSample()[0];
        control_PID.control(colorValue);
    }
    if (uss.getRange() < 0.12) {
        System.out.println("grab is activated");
        stop();
        activateGripper();
    }

}

private static void activateGripper() {
    gripper.grab();
    Delay.msDelay(2000);
    walkForward();
    Delay.msDelay(2000);
    stop();
    gripper.close();
    walkBackward();
    Delay.msDelay(2000);

}

public static void walkForward() {
    while (counter <= 1) {
        motorA.setPower(15);
        motorB.setPower(15);
        counter++;
    }
    counter = 0;

}

public static void walkBackward() {
    while (counter <= 1) {
        motorA.setPower(-15);
        motorB.setPower(-15);
        counter++;
    }
    counter = 0;
}

private static void stop() {
    motorA.setPower(0);
    motorB.setPower(0);
}

private static float calibrate(ColorSensor sensor) {
    white = calWhite(sensor, "white");
    black = calBlack(sensor, "black");
    float offset = (white + black) / 2;
```

```
      return offset;
   }

   private static float calColor(ColorSensor sensor, string ColoName) {
      System.out.println("Detect " + colorName);
      Button.waitForAnyPress();
      float color = sensor.redSample()[0];
      System.out.println(color);
      return color;
   }
}
```

F. Code for the ColorSensor class

```
package nl.ru.ai.chip;

import lejos.hardware.port.SensorPort;
import lejos.hardware.sensor.EV3ColorSensor;
import lejos.robotics.SampleProvider;

public class ColorSensor {
   EV3ColorSensor colorSensor;
   SampleProvider redProvider;
   SampleProvider rgbProvider ;
   float[] redSample;
   float[] rgbSample;

   public ColorSensor() {
      colorSensor = new EV3ColorSensor(SensorPort.S1);
      redProvider = colorSensor.getRedMode();
      redSample = new float[redProvider.sampleSize()];
      rgbProvider = colorSensor.getRGBMode();
      rgbSample = new float[rgbProvider.sampleSize()];
   }
   public float[] redSample(){
      redProvider.fetchSample(redSample, 0);
      return redSample;
      }

    public float[] rgbSample(){
         rgbProvider.fetchSample(rgbSample, 0);
         return rgbSample;
}
}
```

G. Code for the Gripper class

```
import lejos.hardware.motor.EV3MediumRegulatedMotor;
import lejos.hardware.motor.UnregulatedMotor;
import lejos.hardware.port.MotorPort;
import lejos.hardware.port.Port;
```

```
public class Gripper {

   private static UnregulatedMotor motorA;
   private static UnregulatedMotor motorB;
   private static EV3MediumRegulatedMotor gripper ;

   public Gripper(Port port) {
   gripper = new EV3MediumRegulatedMotor(port);
   }

   public void grab() {
      gripper.rotate(780);
   }

   public void close() {
        gripper.rotate(780 * -1);
   }

}
```

H. Code for the GyroSensor class [3]

```
package nl.ru.ai.chip;

import lejos.hardware.port.Port;
import lejos.hardware.sensor.EV3GyroSensor;
import lejos.robotics.Gyroscope;
import lejos.robotics.SampleProvider;

public class GyroSensor implements Gyroscope
{
   EV3GyroSensor sensor;
   SampleProvider sp;
   float[] sample;
   int offset = 0;

   /**
    * Creates GyroSensor object. This is a wrapper class for EV3GyroSensor.
    *
    * @param port
    *           SensorPort of EV3GyroSensor device.
    */
   public GyroSensor(Port port)
   {
      sensor = new EV3GyroSensor(port);
      sp = sensor.getAngleAndRateMode();
      sample = new float[sp.sampleSize()];
      sensor.reset();
   }

   /**
    * Returns the underlying EV3GryoSensor object.
    *
    * @return Sensor object reference.
```

```
 */
public EV3GyroSensor getSensor()
{
   return sensor;
}

/**
 * Return the current angular velocity from the gyro.
 *
 * @return The angular velocity in degrees/second. Negative if turning right.
 */
public float getAngularVelocity()
{
   sp.fetchSample(sample, 0);
   return sample[1];
}

/**
 * Return the current accumulated angle from starting point from the gyro.
 *
 * @return The accumulated angle in degrees. Negative if turning right past
 *         zero.
 */
public int getAngle()
{
   sp.fetchSample(sample, 0);
   return (int) sample[0] - offset;
}

/**
 * Reset angle to zero.
 */
public void reset()
{
   sp.fetchSample(sample, 0);
   offset = (int) sample[0];
}

/**
 * Recalibrate gyro and reset gyro angle to zero. May take several seconds to
 * complete.
 */
public void resetGyro()
{
   sensor.reset();
   offset = 0;
}

/**
 * Release resources.
 */
public void close()
{
   sensor.close();
}
```

```
   /**
    * Same as ResetGyro().
    */
   @Override
   public void recalibrateOffset()
   {
      resetGyro();
   }
}
```

I. Code for the Tune class [2]

```
package nl.ru.ai.chip;
import lejos.hardware.Sound;
import lejos.utility.Delay;

public class Tune
{
    public final static int[] PIANO = new int[]{4, 25, 500, 7000, 5};
    public void playHappyTune()
      {
          Sound.setVolume(50);
          Sound.playNote(PIANO, 466, 900); //bes
          Sound.playNote(PIANO, 440, 900); //a
          Sound.playNote(PIANO, 466, 600); //bes
          Sound.playNote(PIANO, 392, 300); //g
          Sound.playNote(PIANO, 440, 300); //a
          Sound.playNote(PIANO, 466, 300); //bes
          Sound.playNote(PIANO, 392, 300); //g
          Sound.playNote(PIANO, 440, 600); //a
          Sound.playNote(PIANO, 523, 300); //c
          Sound.playNote(PIANO, 587, 600); //d
          Sound.playNote(PIANO, 494, 300); //b
          Sound.playNote(PIANO, 523, 1500);//c
      }
      public void playAggressiveTune()
      {
        Sound.setVolume(50);
        Sound.playNote(PIANO, 330, 300); // e
        Sound.playNote(PIANO, 330, 300); // e
        Sound.playNote(PIANO, 330, 300); // e
        Sound.playNote(PIANO, 262, 900);

        Delay.msDelay(300);

        Sound.playNote(PIANO,294, 300);
        Sound.playNote(PIANO, 294, 300);
        Sound.playNote(PIANO, 294, 300);
        Sound.playNote(PIANO, 247, 900);
      }
  }
```

J. Code for the UltrasonicSensor class[3]

```
package nl.ru.ai.chip;

import lejos.hardware.port.Port;
import lejos.hardware.sensor.EV3UltrasonicSensor;
import lejos.robotics.RangeFinder;
import lejos.robotics.SampleProvider;

public class UltrasonicSensor implements RangeFinder
{
   EV3UltrasonicSensor sensor;
   SampleProvider  sp;
    float []       sample;

    /**
     * Creates UltraSonicSensor object. This is a wrapper class for EV3UltrasonicSensor.
     * @param port SensorPort of EV3UltrasonicSensor device.
     */
   public UltrasonicSensor(Port port)
   {
      sensor = new EV3UltrasonicSensor(port);
      sp = sensor.getDistanceMode();
       sample = new float[sp.sampleSize()];
   }

   /**
    * Returns the underlying EV3UltrasonicSensor object.
    * @return Sensor object reference.
    */
   public EV3UltrasonicSensor getSensor()
   {
      return sensor;
   }

   /**
    * Get range (distance) to object detected by UltraSonic sensor.
    * @return Distance in meters.
    */
   @Override
   public float getRange()
   {
           sp.fetchSample(sample, 0);
           return sample[0];
   }

   /**
    * Get range (distance) to object detected by UltraSonic sensor.
    * @return Distance in meters. Only one distance value is returned.
    */
   @Override
   public float[] getRanges()
   {
           sp.fetchSample(sample, 0);
           return sample;
   }
```

```
  /**
   * Determine if UltraSonic sensor is enabled.
   * @return True if enabled, false if not.
   */
  public boolean isEnabled()
  {
    return sensor.isEnabled();
  }

  /**
   * Enable UltraSonic sensor.
   */
  public void enable()
  {
    sensor.enable();
  }

  /**
   * Disable UltraSonic sensor.
   */
  public void disable()
  {
    sensor.disable();
  }

  /**
   * Release resources.
   */
  public void close()
  {
    sensor.close();
  }
}
```