Computer Project I – Final Report

1st Asger Høøck Song Poulsen
Undergraduates at Aarhus University,
Dept. of electrical and computer engineering
Study number: 202106630
AU-id: au704738

2nd Andreas Kaag Thomsen
Undergraduates at Aarhus University,
Dept. of electrical and computer engineering
Study number: 202105844
AU-id: au691667

***Abstract*—With the continuos improvement of autonomous robots they could sooner than later provide a competetive advantage to more complex tasks such as driving a car or performing a search and rescue operation by reducing error rate, decreasing long-term costs and provide access to dangerous locations. In this paper, we report on our first hands-on experience with embedded programming, working with ROS on a virtual machine and design of real time control software for Aarhus University's computer engineering students. We briefly look into basic Arduino programming and use this knowledge to further implement a program on the Turtlebot3 Burger Robot to drive and avoid obstacles autonomously. We conclude that the implementation of the see-think-act cycle is advantageous for the development of autonomous robots. Furthermore, the course/project has given us as 1st year computer engineering students foundational tools to see, think and act towards developing more complex robots in the future.**

I. Introduction

Overall, the project can be divided into four parts with certain aim and objectives:

1) Understanding the basic concepts of robotics programming by working with an Arduino - that is, the *see-think-act cycle* (see figure 1). Reading and analyzing data from a range sensor and RGB sensor.
2) Working with the *Robot Operating System* (ROS) on a virtual machine. Learning the structure that ROS provides and going through the beginner's tutorial.
3) ROS programming on Raspberry PI. Programming the robot to avoid obstacles in different scenarios.
4) Optimizing the robot's performance both with respect to linear speed and collision avoidance. Finalizing the code and testing the robot on an obstacle course.

II. Specifications

We have used the following equipment throughout the course.

- Arduino PRO MICRO - 5V/16MHZ
- Ultrasonic Range Finder (LV-MAXSONAR-EZ0)
- RGB Light Sensor ISL29125
- LED's, cables etc.
- Turtlebot3 Burger Robot equipped with i.a. a Raspberry Pi 3 and a 360°LiDAR sensor (lds-01).

For coding we have used the language C++ for programming the Arduino on the Arduino software. For programming the Turtlebot we have been using Python and the command prompt with the built in nano-editor.

III. Design and Implementation

In this section we are going to describe our thoughts on design and implementation of the project. We are only going to desribe the underlying abstract thoughts and the concrete implementation will be elaborated in section IV.

A. Part 1: Arduino Programming

We were to consider the *see-think-act* cycle as can be seen in figure 1:

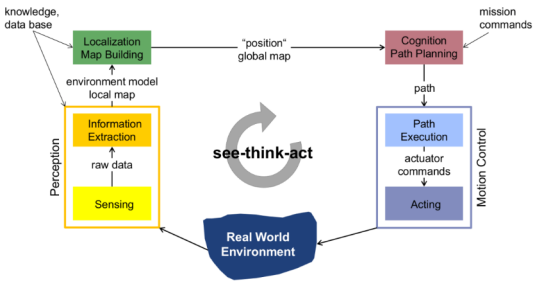


Fig. 1: See-think-act cycle

On our breadboard we connected the Arduino to which we connected the ultrasonic range finders and some LED's. The cycle was then implemented as follows: the range finder sensors **saw** any obstacle we put in front of it, and sent some data back to Arduino. This data was computed - that is the **think** step of the cycle. Then the **act** step was performed, which in this case was the LED blinking.
Afterwards we extended the circuit to include three range finder sensors plus the RGB light sensor and four LED's. This thus acted as a simulation of the real Turtlebot, which we should work with in part 3.

B. Part 2: ROS Programming

In this part of the course we worked on a Virtual Machine (VM) on which we had installed Ubuntu. We went through the ROS beginner's tutorial and learned the structure of a ROS system.

C. Part 3: Programming the robot to avoid obstacles

As described, the Turtlebot is equipped with a LiDAR 360°laser sensor, which measures the distance. It continuosly returns a set of data:

$$\texttt{dist} = [d_0, d_1, \ldots, d_{359}]$$

where d_i is the distance to the nearest object at angle i. We decided to look at a span of 120 degrees, which we divided into three distinct parts:

$$\texttt{left} = [d_{15}, d_{16}, \ldots, d_{60}], \texttt{N=45}$$
$$\texttt{front} = [d_0, d_1, \ldots, d_{14}, d_{345}, d_{346}, \ldots, d_{359}], \texttt{N=30}$$
$$\texttt{right} = [d_{300}, d_{301}, \ldots, d_{344}], \texttt{N=45}$$

We decided to do this so the robot more often would *turn* instead of just driving *backwards*, since this supposedly would achieve an overall higher linear speed. This will be discussed further in section V.

For the different cases of obstacles we have made several cases of `if/elif` statements. The cases and the decisions in each case can be seen in figure 2 below.

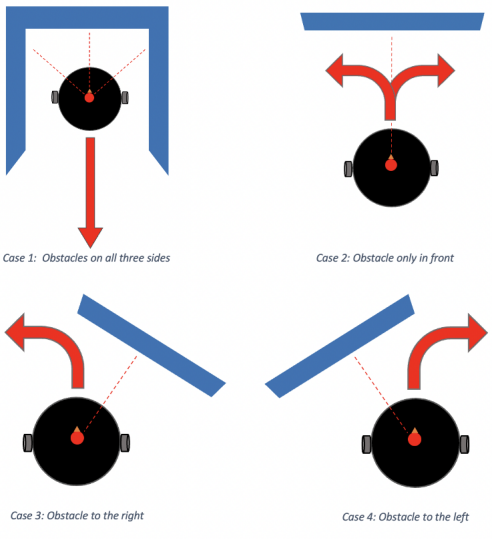


Fig. 2: Obstacle cases

We thus only considered four distinct cases for our intitial setup. However, we later found that this was not sufficient. Our solution to this will be elaborated in the next subsection.

D. Part 4: Optimizing the robot's performance

In the last part of the course we should test the robot on an arbitrary course of obstacles. We did, however, test the robot continuosly in part 3, so we focused this part on optimizing the existing code. The goal for the optimization part was to maintain the linear speed as high as possible and the number of collisions as low as possible. This will be elaborated in subsection IV-D.

IV. Experiment setup and results

In this section we will describe the concrete implementation of the project and the results we got, which will include several code snippets and figures.

A. Part 1: Arduino Programming

We connected the Range sensor to the Arduino and intialized it as follows:

```
int SENSOR = A0; //range sensor connected to port A0
double range_input = 0; //variable for storing input
    values
void setup()
{
  pinMode(SENSOR,INPUT); //sensor declarared as an
    input
}
```

Listing 1: Initialization of variables

In the control loop we updated `range_input` with the value read from the sensor with using the `analogRead` function. Through a series of `if-else` statements we made the LED blink with different intervals for different distances:

```
if (range_input+margin < 20 && range_input+margin >
    0) {
    TXLED0;
    delay(50);
    TXLED1;
    delay(50);
}

else if (range_input+margin < 30 && range_input+
    margin > 25) {
    TXLED0;
    delay(333);
    TXLED1;
    delay(333);
}

else if(range_input+margin < 25 && range_input+
    margin >20) {
    TXLED0;
    delay(1000);
    TXLED1;
    delay(1000);
}
TXLED1; //LED turned off by default
```

Listing 2: `else-if` statements for Arduino

We were thus able to see the robot **act** in different ways depending on what it **saw**, which substantiated the *see-think-act* cycle.

Afterwards we extended the circuit to include three range sensors in the same way as above. We discovered that our measurements at first were wrong since we did not convert the input values to distances. Looking at the equipment specifications we found out that the sensor uses a scaling factor of $6.4\frac{mV}{inch}$. By dividing the analog input with this scaling factor, a measurement in inches is calculated. Afterwards, this is multiplied by $2.54\frac{cm}{inch}$.

```
front_input_cm = (front_input/6.4)*2.54;
left_input_cm = (left_input/6.4)*2.54;
right_input_cm = (right_input/6.4)*2.54;
```

Listing 3: Calculations of ranges

Now we had the correct measurements which we also manually confirmed with a ruler.

Lastly we also implemented the RGB light sensor on the breadboard, which was intialized in the same way as the

range sensors. The RGB sensor returned one value measured in lux for the luminance. If the luminance was less than 30, one LED was turned on. With everything connected as described above, our final circuit is depicted in figure 3 below.

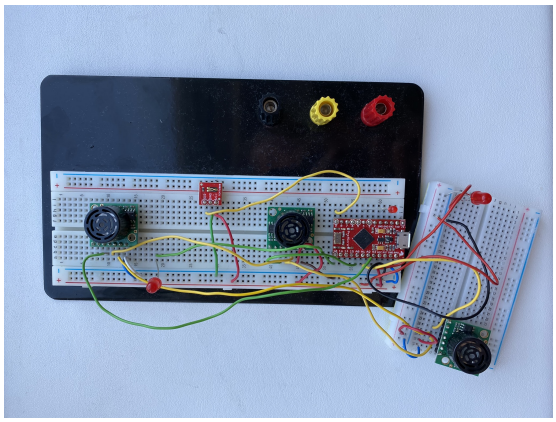


Fig. 3: Final Arduino circuit

We now had a circuit that would serve as the building blocks for our final implemenatation of the Turtlebot with each range finder sensor acting as a representative of each slice desribed in subsection III-C. We will elaborate this further in subsection IV-C1.

B. Part 2: ROS Programming

As mentioned we followed the ROS beginner's tutorial in which we installed ROS using several commands. We got stuck in part 2 of the tutorial, so we decided to go back to the beginning and reinstall everything. After doing this, we were able to complete all the steps from 1 through 17. We got a deeper understanding of how the ROS system is structured and learned several ROS-related commands. We will explain the most essential for our further work with the Turtlebot below[1]:

1) **roscore**: roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate.
2) **rosrun**: rosrun allows you to run an executable in an arbitrary package from anywhere without having to give its full path.
3) **source**: When opening a new terminal your environment is reset and you are obliged to use the `source` command to re-source the .bashrc file.
4) **nano**: nano is a text editor that is used to edit files. We used it to edit our code files directly in the terminal.

In practice we only needed to run three commands when the robot should be tested. Firstly we should run `source /.bashrc` in each new terminal to make sure the environment is set up correctly. Afterwards we needed to run `roslaunch turtlebot_bringup turtlebot_bringup.launch` to start the ROS-master (`roscore`). Then, in a different terminal we should run the code, we had written ourselves by writing the command: `roslaunch turtlebot3_example turtlebot3_obstacle.launch`.

[1]These definitions are taken from the ROS beginner's tutorial, which is linked in section VII

C. Part 3: Programming the robot to avoid obstacles

1) Using the Arduino setup as building blocks: In this part we are going to describe our initial implementation. The optimization we did, will be described in part 4.
We were given a robot navigation example from a GitHub repository[2] from where we were to get inspiration and further implement our own robot navigation. The idea of the initial example was to: *Drive forward until we get close to an obstacle, then stop*. The robot kept driving by initiating the following `while`-loop

```
1 while not rospy.is_shutdown():
```

Listing 4: control loop

So `while` we keep publishing the topic, we iterate through the following `else-if` statements that should make the robot react if the cases from figure 2 were to be encountered. For case 1 we initially programmed the robot to back out of the blind alley and then turn around:

```
1  if (right_distance < SAFE_STOP_DISTANCE and
       left_distance < SAFE_STOP_DISTANCE and
       front_distance < SAFE_STOP_DISTANCE):
2      if turtlebot_moving:
3          twist.linear.x = -LINEAR_VEL
4          twist.angular.z = 0.0
5          self._cmd_pub.publish(twist)
6          rospy.loginfo('Turning backwards for 1
       second')
7          t.sleep(1.0)
8          twist.linear.x = 0.0
9          twist.angular.z = 3.14/2
10         self._cmd_pub.publish(twist)
11         rospy.loginfo('Turning around')
12         t.sleep(1.0)
13         turtlebot_moving = False
```

Listing 5: Case 1

Below here we will explain this code line for line. Note that each item corresponds to each line number.

1) Requirements for the specific case.
2) This variable was set to `true` each time the robot was driving forward. The robot would thus only **act** if it was not already turning away from some other obstacle.
3) In order to alter the robot's movement we had to update the `twist`-command. In this line we updated the linear velocity, which should be negative for the robot to move backwards.
4) Here, we update the twist command with respect to the angular velocity which in this case should be zero since the robot should *only* turn backwards.

[2]The Github Repository of the first navigation example is linked in the references

5) In this line we *publish* the newly updated `twist`-command to the turtlebot. When we do this, the robot will act accordingly.
6) We write to log what we are doing.
7) We wanted the robo to turn backwards for some time, in this case 1 second.
8) Now the robot should only turn and we thus set the linear speed to zero.
9) The robot should turn in place, so we set the angular speed to $\pi/2$.
10) Publishing the new twist command.
11) Writing to the log.
12) Wait for 1 second.
13) Now we set the beforementioned variable `turtlebot_moving` to `false`. It will *only* be set to `true` again when the robot should drive forward (shown in listing 9).

These explanations also hold for all other cases which will only be described shortly below.

For our remaining cases we used a function called `small_turns` as shown in listing 6 to control the robot. The function's functionality corresponds exactly to the steps desribed above - it has only been generalized. The function takes three arguments; a direction, an angular velocity, and a linear velocity which are all used to control the robot[3].

```
def obstacle(self):
twist = Twist()
turtlebot_moving = True

def small_turns(dir, angle, vel):
    if(dir == 'left'):
        twist.linear.x = vel
        twist.angular.z = angle
        self._cmd_pub.publish(twist)
        rospy.loginfo('Turning left')
        turtlebot_moving = False

    elif (dir == 'right'):
        twist.linear.x = vel
        twist.angular.z = -angle
        self._cmd_pub.publish(twist)
        rospy.loginfo('Turning right')
        turtlebot_moving = False
```

Listing 6: `small_turns` function.

For case 2: If the robot found an obstacle to the front it was programmed to look to the left and right respectively, look for next potential obstacle and then turn in the opposite direction of the next potential closest obstacle:

```
elif (front_distance < SAFE_STOP_DISTANCE + 0.05):
    if turtlebot_moving:
        if (right_distance < left_distance):
            small_turns('left', turn_angle + 0.2)
        elif(left_distance < right_distance):
            small_turns('right', turn_angle + 0.2)
```

Listing 7: Case 2

For case 3 and 4: If an obstacle to the left is within the `SAFE_STOP_DISTANCE` turn right and vice versa.

[3]For our initial implementation we only used the two first inputs and the linear velocity when turning was consistenly equal to $0.1\frac{m}{s}$.

```
elif(left_distance < SAFE_STOP_DISTANCE or
    right_distance < SAFE_STOP_DISTANCE):
    if turtlebot_moving:
        if(right_distance < left_distance):
            small_turns('left', turn_angle)
        else:
            small_turns('right', turn_angle)
```

Listing 8: Case 3 & 4

Lastly, by default, we drive forward if there are no obstacles to avoid. Note that this is where we set `turtlebot_moving` to `true`.

```
else:
    twist.linear.x = LINEAR_VEL
    twist.angular.z = 0.0
    turtlebot_moving = True
    self._cmd_pub.publish(twist)
```

Listing 9: Else: drive forward

Thus, ending an iteration of the control-loop `while not rospy.is_shutdown():`.

2) Retrieving data from the LiDAR sensor: In the original GitHub repository, the LiDAR sensor only looked at one measurement. We altered the function `get_scan` to fit our needs:

```
def get_scan(self):
    scan = rospy.wait_for_message('scan', LaserScan)
    scan_filter = []

    left_lidar_samples_ranges = 60
    right_lidar_samples_ranges = 345
    front_lidar_samples_ranges = 15

    front_lidar_samples1 = scan.ranges[0:
    front_lidar_samples_ranges]
    front_lidar_samples2 = scan.ranges[
    right_lidar_samples_ranges:360]
    left_lidar_samples = scan.ranges[
    front_lidar_samples_ranges:
    left_lidar_samples_ranges]
    right_lidar_samples = scan.ranges[300:
    right_lidar_samples_ranges]

    scan_filter.append(left_lidar_samples)
    scan_filter.append(front_lidar_samples1)
    scan_filter.append(front_lidar_samples2)
    scan_filter.append(right_lidar_samples)

    return scan_filter
```

Listing 10: `get_scan` function

We recieve the LiDAR data when calling `rospy.wait_for_message` and then create an empty list `scan_filter` from which we would fill in the data that we would need, hence **filter** out the unecessary data, since we are only interested in distances in a span of 120 degrees as mentioned in subsection III-C. From there we append the sections we are interested in to the list `scan_filter` with Python's `append` function. When doing so, `scan_filter` consists of five elements which are each tuples of distances.

3) Tuples in python: When testing the robot we encountered that the LiDAR would return a lot of zeros which would represent an obstacle too close to the robot such that the robot would and should react to. This is because the LiDAR sensor has a max range of 3.5 m, so if an obstacle was further away

than this it would just set the distance data to 0 which was a problem. We solved this by converting the zeros to 3.5 (max distance) instead, but in Python you can not edit data in tuples, so we had to convert this list of tuples to a list of lists and from there convert the zeros to 3.5. We created the following function `non_zeros` to do so.

```
# function converts list of tuples to list of lists and then converts zeros to 3.5:
def non_zeros(l):
    # convert tuples to list
    for i in range(len(l)):
        l[i] = list(l[i])

    # if element zero, set to 3.5
    for i in range(len(l)):
        for j in range(len(l[i])):
            if l[i][j] == 0:
                l[i][j] = 3.5
```

Listing 11: `non_zeros` function

The first for-loop converted the tuples to lists and the next converted zeros to 3.5.

4) A short note on the see-think-act cycle*:* Considering our code above on a more abstract level, we can see that this actually corresponds 1:1 with the see-think-act cycle as seen in figure 1. Our `get_scan`-function thus acts as the **see**-step in which we 'sense' our environment, so to speak. Afterwards we have the **think**-step in which we determine the best action to take. That is, the different `if/elif`-statements. Finally, we have the **act**-step in which we execute the action we have decided upon. This is when we publish our `twist`-command to the Turtlebot.

5) Defining the macros: Several of the code snippets above shows a bunch of global variables that we had initialized in the very beginning of our code. These initializations can be seen in listing 12.

```
LINEAR_VEL = 0.22
STOP_DISTANCE = 0.2
LIDAR_ERROR = 0.05
SAFE_STOP_DISTANCE = STOP_DISTANCE + LIDAR_ERROR
MAX_ANGLE = 2.84
DEFAULT_TURN_ANGLE = 3.14/2
```

Listing 12: Global variables

Now, we are going to explain these choices. Note that each item corresponds to each line in listing 12.

1) Since we wanted the highest possible linear velocity, we set the default value of `LINEAR_VEL` to $0.22\frac{m}{s}$, which is the maximum linear velocity of the turtlebot according to its documentation. This was the linear speed the robot would move at when driving forward. When turning, this would be adjusted with respect to this basis.
2) We chose this by empirically testing which distance the robot could avoid an obstacle by smoothly turning. We found that with smaller distances, the robot had to stop completely, which we only wanted in critical cases.
3) This error was the one defined in the initial code setup from GitHub[4].
4) This is the (default) distance we would actually stop at.
5) This is the maximal angular velocity measured in $\frac{rad}{s}$ according to the specifications of the turtlebot.
6) This is the angular velocity the robot would turn at when turning in most cases.

D. Part 4: Optimizing the robot's performance

In this part we are going to describe the changes we applied to the code setup from part 3 and the results we got.

1) Adding a `factor` to `SAFE_STOP_DISTANCE`: We experienced that when the robot encountered an obstacle on its front it did not manage to avoid it optimally. We thus added a factor of 0.08cm to the `SAFE_STOP_DISTANCE` constant in this case. By doing this, the robot began avoiding the obstacle earlier, and thus it was able to avoid it more optimally. Same goes for many of our other cases.

2) Making sure to avoid the obstacle before next control loop iteration: We wanted to make sure that the robot actually avoided the obstacle before the next control loop iteration. We did this by adding a `while`-loop to each case as shown in listing 13.

```
# case 1: obstacle in front
if (front_distance < SAFE_STOP_DISTANCE + 0.08):
    print('obstacle in front')
    # look ahead:
    if turtlebot_moving:
        if (right_distance < left_distance):
            while(front_distance < SAFE_STOP_DISTANCE + 0.08):
                small_turns('left', 'angular vel', 'linear vel.')
                front_new = self.get_scan()
                non_zeros(front_new)
                emergency_check(front_new)
                front_distance = np.mean(front_new[1] + front_new[2])
```

Listing 13: `while`-loops for obstacle avoidance

By doing this we ensured that the robot would not stop avoiding the obstacle before it had actually escaped it[5]. For each loop iteration we made a new laser scan and checked if the robot was within the `SAFE_STOP_DISTANCE+factor`. If it is, we called the `small_turns` function, and turned the robot in the opposite direction of the closest obstacle as normally.

3) Checking the blind angles: In addition to the three directions we had already implemented, we decided to focus on two more directions; the so-called blind angles. These were meant to make the robot able to avoid an obstacle if it was only

[4]We later learned that this error was actually not correct according to the specifications of the LiDAR sensor. The accuracy for measurements in the range 120mm − 499mm was ±15mm. However, we experienced good results with the chosen values.

[5]Note that the 2nd and 3rd argument in the `small_turns` function are written in pseudo-code since we did further optimizations, which will be described in subsection IV-D4. The function `emergency_check` will also be described in subsection IV-D5.

slightly in front of it, but not so much that the average would get low enough to cause a turn event. We calculated these by measuring the minimal distance between two obstacles that robot could drive through - that is, the width of the robot, which we measured to be 20cm. We wanted the robot to begin avoiding the obstacles at the blind angles from a distance of 40cm. We got the results using GeoGebra as depicted in figure 4. We also wanted these angles to be very narrow such that the robot would be able to detect even a small corner of some obstacle. For this reason we only looked at a span of 8 degrees.

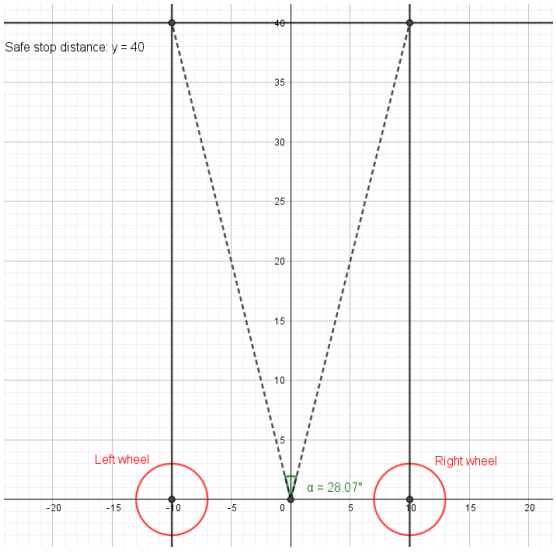


Fig. 4: Blind angles

We thus included the following code in our function `get_scan()`:

```
front_left_lidar_samples = scan.ranges[14:22]
front_right_lidar_samples = scan.ranges[338:346]

scan_filter.append(front_left_lidar_samples)
scan_filter.append(front_right_lidar_samples)
```

Listing 14: Blind angles in `get_scan()`

In addition to this, we added an extra case (named case 3) for checking these blind angles. This was implemented in the exact same way as desribed in subsection IV-D2 using another `elif`-statement. The robot should detect these blind angles quite early, so we added a factor of 0.15cm to the `SAFE_STOP_DISTANCE` constant.

4) Optimizing linear speed: For our initial implementation, our robot would *turn* with a constant linear speed of $0.1\frac{m}{s}$. Our goal was to achieve an overall higher linear speed. We did this by considering the relationship between the linear speed and distance to object as well as the relationship between the angular speed and distance to object. We wanted the following:

- The linear speed should be **smaller**, the closer the obstacle was.
- The angular speed should be **greater**, the closer the obstacle was.

To do this, we considered some distinct cases[6]:

1) Robot is at maximal `SAFE_STOP_DISTANCE` away from the obstacle, that is 0.4m for blind angle obstacles and 0.33m for front obstacles.
   - Linear speed should be `MAX_LINEAR_VEL`, that is $0.22\frac{m}{s}$ for blind angle obstacles and $0.18\frac{m}{s}$ for front obstacles.
   - Angular speed should be minimal, that is $0.79\frac{rad}{s}$ for blind angle obstacles and $2.0\frac{rad}{s}$ for front obstacles.
2) Robot is critically close to the obstacle, that is 0.1m.
   - Linear speed should essentially be zero, that is $0.01\frac{m}{s}$.
   - Angular speed should be `MAX_ANGLE`, that is $2.84\frac{rad}{s}$.

With these values, we made power regression using GeoGebra. The result for linear velocity is depicted in figure 5. Here, $h(x)$ and $g(x)$ are the linear velocities in $\frac{m}{s}$ for front obstacles and blind spots, respectively. x is the distance to the obstacle in m.

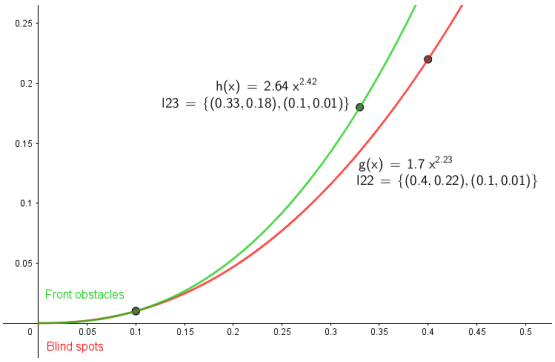


Fig. 5: Linear velocities graphs

The result for angular velocity is depicted in figure 6. Here, $g(x)$ and $h(x)$ are the angular velocities in $\frac{rad}{s}$ for front obstacles and blind spot obstacles, respectively. x is the distance to the obstacle in m.

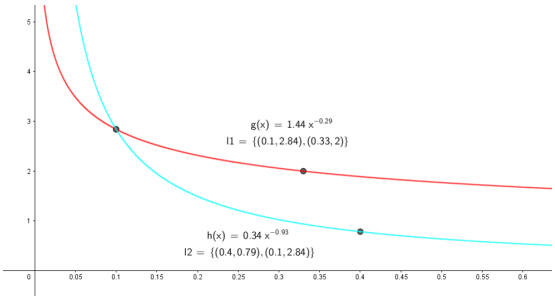


Fig. 6: Angular velocities graphs

Afterwards, we implemented this in our code. We needed to include the power functions we had found

[6]These speeds were tested empirically until we found appropriate values.

each time we called the `small_turns` function. An example for the blind spot case is shown in listing 15.

```
small_turns('right', 0.34*front_left_distance**
(-0.93), 1.7*front_left_distance**2.23)
```

Listing 15: Power functions implemented for blind spot

After implementing these changes in the code, we significantly improved the robot's linear speed.

5) Better collision avoidance: Another goal of our optimization was to avoid collisions with obstacles completely. We did this by implementing a new function called `emergency_check` that would check if the robot was critically close to an obstacle. We defined the robot to be critically close to an obstacle if it is within 0.1m of an obstacle. With a `LIDAR_ERROR` of 0.05m, we introduced a new global variable `EMERGENCY_STOP_DISTANCE = 0.15`. We chose this value since the minimum distance the LiDAR sensor can measure is 0.12m. We did not want to get below this value since it could result in a faulty reading.[7]. If it was within this distance, the robot's linear speed would be set to zero and the robot would turn with max angular speed. The function is depicted in listing 16.

```
1  def emergency_check(l):
2      # calculate mean of our slices:
3      left_distance = np.mean((l[0]))
4      front_distance = np.mean(l[1] + l[2])
5      right_distance = np.mean(l[3])
6      front_left_distance = np.mean(l[4])
7      front_right_distance = np.mean(l[5])
8
9      # calculate mean of special case slices
10     special_case_right = np.mean(l[3][27:35])
11     special_case_left = np.mean(l[0][10:18])
12
13     if (front_distance < EMERGENCY_STOP_DISTANCE and
        left_distance < EMERGENCY_STOP_DISTANCE):
14         small_turns('right', MAX_ANGLE, 0.0)
15
16     elif (front_distance < EMERGENCY_STOP_DISTANCE
       and right_distance < EMERGENCY_STOP_DISTANCE):
17         small_turns('left', MAX_ANGLE, 0.0)
18
19     elif (front_distance < EMERGENCY_STOP_DISTANCE):
20         if (left_distance < right_distance):
21             small_turns('right', MAX_ANGLE, 0.0)
22         else:
23             small_turns('left', MAX_ANGLE, 0.0)
24
25     elif(special_case_right < SAFE_STOP_DISTANCE and
        special_case_left < SAFE_STOP_DISTANCE):
26         make_180()
27
28     elif(front_left_distance < SAFE_STOP_DISTANCE):
29         small_turns('right', MAX_ANGLE, 0.0)
30
31     elif(front_right_distance < SAFE_STOP_DISTANCE):
32         small_turns('left', MAX_ANGLE, 0.0)
```

Listing 16: Emergency check

The function takes an input `l` which is a list of tuples containing the latest readings. We called this function for each loop iteration, both in the overall control loop as well as in the loops described in subsection IV-D2 and depicted in listing 13. By doing this we constantly checked if the robot was so close to an obstacle that it could not avoid it by turning conventionally. Instead, the robot should turn *in place* with maximal angular speed. Note that we initially wanted the turtlebot to drive backwards and then turn, but while testing the implementation from listing 5 we noticed that backing out was not an option unless we furthered our vision span to also look behind, since we could risk backing into an obstacle as illustrated below.

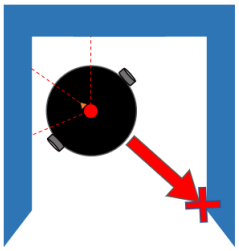


Fig. 7: Initial implementation flaw

So by turning in place until the coast is clear turned out to be an optimal solution for this.

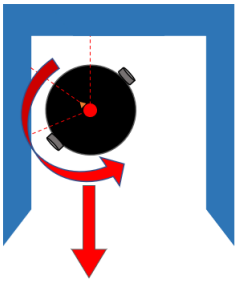


Fig. 8: Illustration of line 25-26 in listing 16

V. Discussion

In this section of the report we will discuss some of the difficulties we encountered during the project. We are also going to discuss whether or not we reached our goal. Again, we have divided it into the four main parts of the project described earlier.

A. Part 1: Arduino Programming

1) **Computing the readings correctly**
We experienced some difficulties in computing the readings correctly since we were not able to find the right scaling factor. We found that many sensors were to be treated differently since they did not give consistent readings. However, since this was only a preliminary exercise to the main project, we decided not to spend much time on the problem, since we had to focus on our main goal.

[7]We would not actually get below this value with the true error of the sensor. See footnote 3.

2) **Preparation to main project**
Working with the Arduino and the appurtenant sensors helped us understand the importance of the see-think-act cycle as shown in figure 1. Our code structure turned out to be predominantly alike in the way we worked with the `if/elif`-statements. This was of course only with respect to some very simple cases, but the underlying though processes were not very different. The body of the `if/elif`-statements was also way more complex in our robot's code. Moreover, we found that working with the sensors was not that easy and that it demanded a lot of effort and reading to actually understand the input from the sensors. This was a valuable lesson that prompted us to work more structured in the main project.

B. Part 2: ROS Programming

1) **Understanding the ROS structure**
Working with the ROS beginner's tutorial gave us a deeper understanding of how the ROS works. We found that in order to get the robot to work we had to follow this structure that ROS provides. Deviation from this would quickly cause the robot to stop working and a bit of troubleshooting would be needed. On a more abstract level we learned that robot programming is extremely complex and requires a lot of prior knowledge, which we do not have the foundation for yet. Given the amount of time of time allocated to this project, we decided to focus on the more important aspects, which would be the programming and optimizations of the robot.

C. Part 3: Programming the robot to avoid obstacles

1) **Getting wrong sensor measurements**
We experienced quite a difficulty in getting the right measurements from the sensor. We knew that it would return an array of size 360 with one distance for each angle. However, we couldn't figure out how they were indexed and our robot thus behaved incorrectly and sometimes turned left when it should turn right. We solved the problem by manually printing out different test angles and then physically measure that angle also. For instance printing `scan_ranges[45]` gave some change in output distance for some specific angle, which we measured. In turned out that we had used it incorretly and the indexing was actually opposite of what we thought. The indexing was counterclockwise, such that angle $0 - 90$ degrees was the first 90 degrees to the left.

2) **Using `mean` instead of `min` and resolving problems with faulty readings**
Our initial code setup from GitHub used Python's `min` function to find the representative for each direction's minimum distance. However, we found that our readings from the LiDAR contained a lot of zeros, which caused this representative to be zero very often. For this reason we decided to calculate the average of the readings instead using Numpy's `mean` function. After doing this we experienced that our readings became much more representative of the actual distance.
Moreover, since our readings still contained a lot of zeros, we modified our code to use the `non_zeros`-function as shown in listing 11. This converted potential zeros to the sensor's max range of 3.5m. This could however actually become a problem since both very close obstacles and very far obstacles could be represented by the same distance of 0. Thus our average value could become quite ambiguous. A better way to deal with this problem could be to *remove* these faulty reading, making the average more precise and more reliable.
When using `min` we also only concentrate on a single point in our 'slices', which when we tested the LiDAR sensor was not always 100% accurate, so sometimes when there actually was a closer object to the left than to the right the robot would still too often turn left. Since `mean` considers all the points in the 'slices' we get a more representative view of the current situation the robot should react to.

3) **Finding the right span**
In the beginning of the robot programming, we only looked at *point readings* and not an entire span of readings. Of course, this resulted in a lot of problems, since we could not see most obstacles. Thus, we had to find a proper span of angles to look at. As mentioned, we ultimately decided to look at a span of 120 degrees, which is not coincidental. Figure 9 shows a graphic representation of the chosen span, which we will argue for.

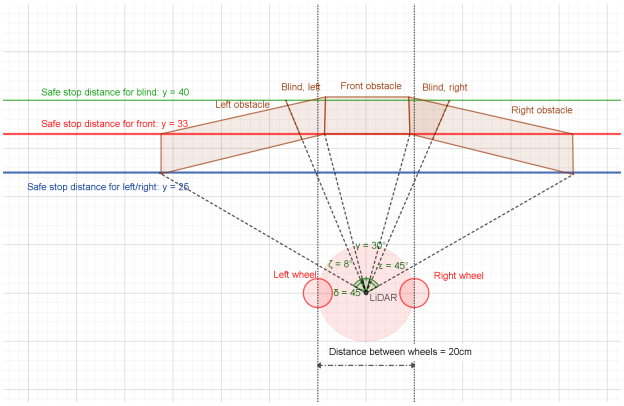


Fig. 9: The chosen span of the robot's sensor-readings.

First of all, the span should be less than 180 degrees, as the robot should *not* try to avoid obstacles that it would not drive into by continuing driving forward. Secondly, the span should be large enough to cover the entire field of view of the robot. If we only looked at, say,

$30 - 50$ degrees ahead, the span was not wide enough to differentiate between different directions, and thus choose in which direction to turn. We the choice of 120 degrees, we had a complete view of the field of view of the robot and achieved the following:

- The robot could see obstacles that was *directly* in front of it. In these cases, it would also be able to select whether to turn left or right. We chose the front span to be only 30 degrees wide such that it could also avoid very narrow obstacles in front of it.
- The robot could see obstacles that was *indirectly* in front of it. That is, to the left *or* right of it. In these cases, it would only turn slightly to the left or right with high linear speed and thus almost maintain its path.
- In the special case that the robot was facing just a corner of an obstacle, it would also see this due the 'blind angle' sections. Again, the robot would only turn very slightly to the left or right with high linear speed and thus practically maintain its path.

The `SAFE_STOP_DISTANCE`s in figure 9 are due to the factors introduced in subsection IV-D1.

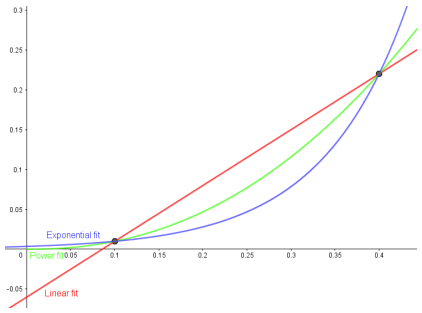
D. Part 4: Optimizing the robot's performance

1) Structuring the `if/elif` statements

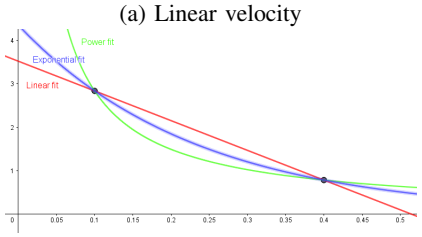
During our implementation we found that the way we ordered our `if/elif`-statements was very important to pay attention to. At first, we did not really pay attention to this, but we experienced that our robot was not working optimally and kept driving into obstacles. We discussed that it was most important that the robot avoided obstacles that were directly in front of it. If it did anything else at first, it would often not turn enough and thus drive into the front obstacle. Next we wanted in to avoid obstacles that were to the left or right, since this case was defined with a higher angular velocity than the blind angle cases. Finally we wanted the robot to slightly turn away from obstacles that were in the blind spots of the robot. In this way the robot would always make the *most radical choice if it was necessary*. Next it would go to the second-most radical choice and so on. If nothing was needed to be done, it would simply drive forward.

2) Finding the proper regression type

As mentioned, we used power regression to determine the relationship between the distance and the speed (angular and linear). But why did we choose this instead of a linear regression? Or exponential regression? Linear regression would indisputable result in a higher average linear speed (see figure 10). We found that if the robot was driving too fast while avoiding an obstacle it would eventually drive into the obstacle. We thus wanted a regression type, which would slow down the robot quite early, avoid the obstacle and then quickly accelerate again. It should, however, not slow down the robot too much since we still wanted a high average linear speed. Thus we chose power regression for linear velocity (the green curve in figure 10a).



(a) Linear velocity



(b) Angular velocity

Fig. 10: Comparison of three different regression types.

When it comes to the angular velocity we wanted the robot to only turn very slightly if it was far away from the obstacle. This was because it should essentially maintain its current path if possible. We only wanted it to turn with max angular speed if it was critically close to the obstacle (around 15cm). Looking closely at our chosen curve (the green one in figure 10b), we see a very steep slope around the critical distance. Further mathematical analysis revealed that the slope of the green curve was actually the greatest at $x = 0.15\text{cm} \pm 0.02\text{cm}$.[8] The actual calculations can be seen in table I.

	Distance (measured in meters)		
Regression type	$x = 0.13$	$x = 0.15$	$x = 0.17$
Power regression	-16.22	-12.3	-9.66
Exponential regression	-10.71	-9.83	-9.02
Linear regression	-6.85	-6.85	-6.85

Table I: Slope of curves from figure 10 around critical distance.

In this way the robot would only make a sharp turn when the obstacle was critically close to an obstacle. The acceleration would be the greatest at the critical distance, which we wanted. Thus we chose power regression for the angular velocity as well.

3) Obstacle avoidance for some special cases

We have already argued that the robot would avoid an obstacle even if it was just facing a corner of it, though

[8]The slope of these curves is naturally the acceleration since differentiation of the velocity gives the acceleration.

we emphasized the fact that it would be on *either* of the sides. However, we did not completely succeed in solving the problem of an obstacle in both blind spots. We did calculate these angles such that it would be able to drive through if there was nothing in both blind spots. But we experienced that this only worked when the robot was driving completely straight towards these blind angles. We discussed that the problem might could have been solved by widening the gap between the blind spots and thus having larger safety margin. However, this would compromise the original thought behind these angles. Perhaps a better solution would be to introduce a new case having its own parameters considering this situation. The best approach would be to make the 'slices' of the blind spots dependent on the distance to to the obstacles, since we measured the angles as depicted in figure 4.

4) **Is it possible to avoid the emergency stop?**
As mentioned in subsection IV-D2 we implemented a function called `emergency_check` which would check if the robot was in an emergency situation. If so, it would set the linear velocity to zero and the angular velocity to the maximum possible. However, one could argue that an optimal implementation (with respect to linear speed) would not need this. It is important to consider the trade-off between linear speed, obstacle avoidance, and areal covered[9]. Assume that we set the `SAFE_STOP_DISTANCE` very high, perhaps 1.0m. Then the robot would act very early and easily avoid the obstacle. But this would also have the effect that the robot would not be able to drive close to the obstacle. Ultimately we wanted a robot that could drive close to an obstacle and still be able to avoid it maintaining a high average linear speed. This caused the robot to sometimes be very close to an obstacle, which prompted that the linear speed had to be set to zero for a moment. An implementation which should *only* be optimized with respect to obstacle avoidance and linear speed could avoid this and just drive away from potential obstacles in good time with maximal linear speed. This was not the goal of our implementation and so we chose to have the emergency-function.

5) **Final tests of the robot**
We tested the robot with a varíous of different scenarios. Illustrated below are some of the many test-courses on which we tested the final implementation on.

[9]Our motivation for the project was namely to simulate a search and rescue robot, which could both avoid obstacles *and* find victims. For this reason it is very important to cover as much area as possible. The 'rescue' part was discarded during the course due to different reasons.

Fig. 11: Navigating through tight spaces

We encountered that the turtlebot would mostly navigate easily through tight spaces. There were some few occasions where the wheels would graze an obstacle. Since there are obstacles in almost every direction these were the type of test-courses we found to be the most difficult with the most obstacle collisions. We also made the most break throughs in optimizing the navigation when testing through tight spaces. It was here we implemented the `emergency_check` function which reduced the amounts of collisions dramatically.

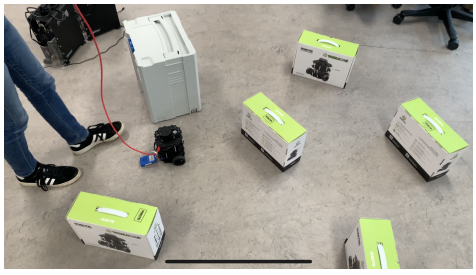


Fig. 12: Placing obstacles in random places

The final implementation navigated through this type of test-course smoothly and without any collisions. It was through these types of test-courses where the optimization of keeping the linear speed as high as possible with regressional input to the `small_turns` function was implemented.

VI. Conclusion

The see-think-act cycle is a powerful tool in understanding basic robot behaviour. Though working with quite simple programming and robots, it is also a powerful tool in understanding complex robot behaviour, which we have now learned the foundation for. We have also learned that in order to program a robot, it is crucial to be structured and work systematically with the different aspects this ecompasses. The *Robot operating system* provides this structure, which again gives the foundation for later working with more complex robots. Finally we have also found that mathematical analysis is a powerful tool in optimizing the robot behaviour. This is especially important in the case of obstacle avoidance, which is the main reason for the existence of this project.

VII. Personal contributions

For programming and SSH'ing into the Turtlebot we have exclusively used Andreas' PC. We chose this since we encountered some difficulties in working on Asger's MacBook. We

did however collaborate closely throughout the entire project. We did also lose a classmate in the middle of the project, which meant we were only 2 left. For this reason it did not make much sense to divide the different parts of the project between us. As a result we have both been focusing on getting a deep understanding of all aspects of the project and associated learning outcomes of the course.

References

[1] Course material from Brightspace.
[2] ROS beginner's tutorial.
[3] Arduino PRO micro Specifications.
[4] Data sheet and specifications for Ultrasonic Range Finder.
[5] Data sheet and specifications for RGB light sensor.
[6] First robot navigation example from GitHub.
[7] Turtlebot3 specifications.
[8] LiDAR sensor specifications.