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*—This is our abstract.**

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*. Reading and analyzing data from a range sensor and RGB sensor.
2) ROS Programming on Raspberry PI. Learning the structure that ROS provides and going through the beginner's tutorial.
3) Programming the robot to avoid obstacles. Then optimize the robot's performance both with respect to linear speed and collision avoidance.
4) 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.

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

Design and implementation of the system.

A. Part 1

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

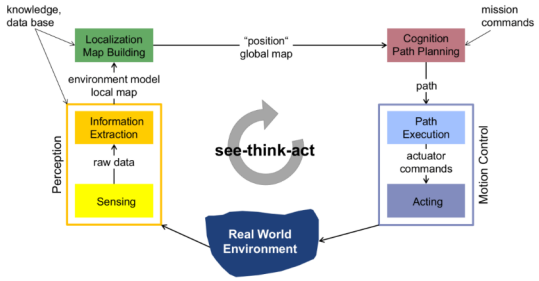

Fig. 1. See-think-act cycle

On our breadboard we connected the Arduino to which we connected the ultrasonic range finders and som 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

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

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

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

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:

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

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.

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

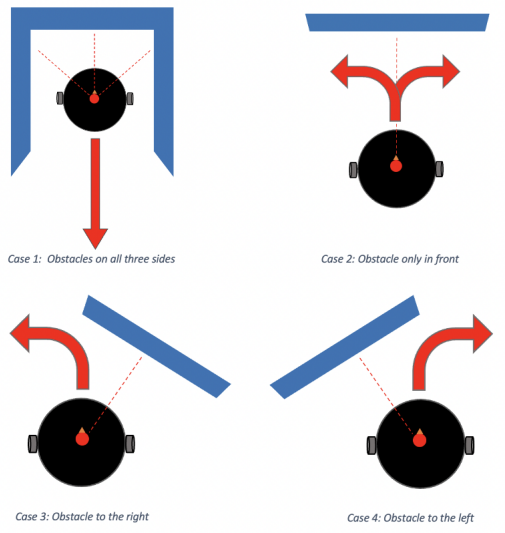

Fig. 2. Obstacle cases

D. Part 4

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 of this course was to maintain the linear speed as high as possible and the number of collisions as low as possible. This will be elaborated in the next section.

IV. Experiment Setup and Results

A. Part 1

We connected the Range sensor 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

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.

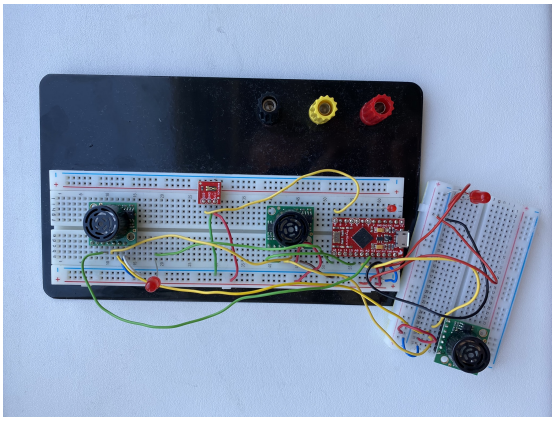

Fig. 3. Final Arduino circuit

B. Part 2

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.

C. Part 3

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 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*.
For case 1 we initially programmed the robot to back out of the blind alley for and then turn around:

```
if (right_distance < SAFE_STOP_DISTANCE and
    left_distance < SAFE_STOP_DISTANCE and
    front_distance < SAFE_STOP_DISTANCE):
    if turtlebot_moving:
        twist.linear.x = -LINEAR_VEL
        twist.angular.z = 0.0
        self._cmd_pub.publish(twist)
        rospy.loginfo('Turning backwards for 1
    second')
        t.sleep(1.0)
        twist.linear.x = 0.0
        twist.angular.z = 3.14/2
        self._cmd_pub.publish(twist)
        rospy.loginfo('Turning around')
        t.sleep(1.0)
        turtlebot_moving = False
```

Listing 4. Case 1

For case 2 the robot was programmed to look to the left and right respectively and then turn in the opposite direction of the 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 5. Case 2

Lastly, for case 3 and 4: If an obstacle on the left is within the `SAFE_STOP_DISTANCE` turn right and vice versa.

```
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 6. Case 3 & 4

D. Part 4

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.

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 7.

```
# 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 + factor):
                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 7. `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[2]. 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.

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

[2]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. Same goes for the variable `factor`, which was described in the subsection before. The function `emergency_check` will also be described in subsection IV-D5.

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 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 result 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.

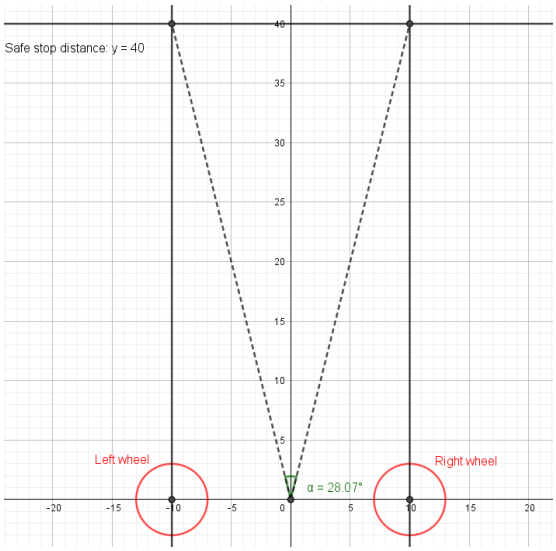

Fig. 4. Blind angles

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 8. 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[3]:

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 $0.22\frac{m}{s}$.
   - 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 crititacally 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, $p(x)$ is the linear velocity in $\frac{m}{s}$, and x is the distance to the obstacle in m.

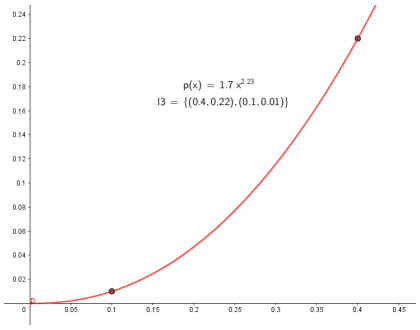

Fig. 5. Linear velocity graph

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.

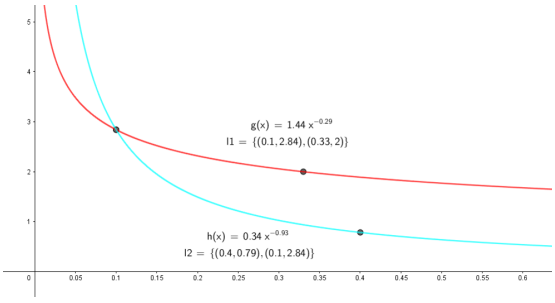

Fig. 6. Angular velocity graphs

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

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

called the `small_turns` function. An example for the blind spot case is shown in listing 9.

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

Listing 9. 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 crititacally 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`. If it was, 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 10.

```
def emergency_check(l):
    # calculate mean of our slices:
    left_distance = np.mean((l[0]))
    front_distance = np.mean(l[1] + l[2])
    right_distance = np.mean(l[3])
    front_left_distance = np.mean(l[4])
    front_right_distance = np.mean(l[5])

    # calculate mean of special case slices
    special_case_right = np.mean(l[3][27:35])
    special_case_left = np.mean(l[0][10:18])

    if (front_distance < EMERGENCY_STOP_DISTANCE and
     left_distance < EMERGENCY_STOP_DISTANCE):
        small_turns('right', MAX_ANGLE, 0.0)

    elif (front_distance < EMERGENCY_STOP_DISTANCE
    and right_distance < EMERGENCY_STOP_DISTANCE):
        small_turns('left', MAX_ANGLE, 0.0)

    elif (front_distance < EMERGENCY_STOP_DISTANCE):
        if (left_distance < right_distance):
            small_turns('right', MAX_ANGLE, 0.0)
        else:
            small_turns('left', MAX_ANGLE, 0.0)

    elif(special_case_right < SAFE_STOP_DISTANCE and
     special_case_left < SAFE_STOP_DISTANCE):
        make_180()

    elif(front_left_distance < SAFE_STOP_DISTANCE):
        small_turns('right', MAX_ANGLE, 0.0)

    elif(front_right_distance < SAFE_STOP_DISTANCE):
        small_turns('left', MAX_ANGLE, 0.0)
```

Listing 10. 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 7. 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.

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

1) **Preparation to main project**

B. Part 2

1) **Understanding the ROS structure**

C. Part 3

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]` which gave some change in output distance for some specific angle, which we measured. In turned out that it our code should be reversed.
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 **??**. 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.

D. Part 4

1) **Structuring the `else-if` statements**

VI. References

- Course material from Brightspace.
- ROS beginner's tutorial: *http://wiki.ros.org/ROS/Tutorials*
- Data sheet and specifications for Ultrasonic Range Finder: *shorturl.at/cBHN1*
- Data sheet and specifications for RGB light sensor: *shorturl.at/htEP8*
- First robot navigation example: *shorturl.at/lpuM8*