Computer Technology Project I

TurtleBot3 - Robot Design

Daniel Pihl
Århus University
AU712814

Steffen Petersen
Århus University
AU722120

***Abstract*—This project aimed to implement autonomous navigation for a Turtlebot3 robot using the Robot Operating System (ROS), with a primary objective of obstacle avoidance and achieving a high average linear velocity in a simulated "search and rescue" scenario. The robot navigation was designed based on the "see - think - act" concept, using LIDAR and RGB sensing to detect the environment and simulated victims. The existing ROS framework was utilized and extended on a Raspberry Pi 3 to control the robot and establish communication with the sensors.**

The navigation algorithm underwent several iterations to enhance the robot's movement, taking into account the trade-offs between linear and angular speed. The final implementation incorporated techniques for smooth turns, corner avoidance, and collision handling. The system's performance was evaluated through experiments conducted in a controlled environment featuring a predefined obstacle course. The results demonstrated successful autonomous navigation, effective obstacle avoidance, and good speed while minimizing collisions. This project provided valuable insights into ROS and the principles of robot implementation, sensor integration and navigation algorithms.

I. Introduction

In this project, our main goal was to implement a Turtlebot3 robot for a simulated version of search and rescue purposes.
This would involve creating a program to allow the robot to move autonomously and avoid obstacles, utilizing the Robot Operating System (ROS), all while searching for targets on the floor.
We also aim to optimize this robot for the highest possible speed, through an efficient navigation logic, following the "see - think - act" concept.

II. Specifications

A. TurtleBot3 Burger

In this project we are using the TurtleBot3 Burger robot, to practice our ROS based robot programming.
The TurtleBot3 Burger is equipped with a Raspberry Pi 3 Model B [1] in combination with an OpenCR 1.0 board, to provide us with options for programming and controlling it.

B. LDS-01 LIDAR

Additionally the Burger has an LDS-01 [2] LIDAR scanner mounted on top, which we use for the navigation of the Burger.

C. Raspberry Pi 3 Model B

The Raspberry Pi is a small-scale computer ideal for hobbyist purposes, with IO pins, that we could use to add extra sensors to our robot [3].

D. RGB Sensor

Using the extra IO support of the Raspberry PI, we could add the ISL-29125 RGB sensor [4], that allows us to simulate scanning for victims.

III. Process

In this section we will elaborate on the process of developing our own understanding of the problem, and the solutions we have come up with along the way.

A. Design and Implementation

In this section we are going to talk about our own thoughts on design and implementation but in an abstract way.
Concrete implementation and thoughts will come in the later sections.

In the early stages of our process, we started out with the RGB sensor as it was a component that just had to be mounted on the robot and connected to the Pi. If we could make that work, it was a simple matter of just connecting it later.
Our intention was to make the sensor read some coloured tags from the floor, which would simulate victims that our robot had to recognize as it moves around.
Firstly we installed the necessary libraries on the Pi, *smbus* to communicate with the sensor, and *time* for the initialization.
To start working on the implementation, we were given a template, that already had the code for communicating through *smbus* with the sensor, to configure the registers of the sensor. We then had to add to that, a way to extract the red, green and blue values from the registers within the sensor, so that we could use these in our to-be robot code, and detect victims.
After we got the RGB sensor working we moved on to theorize what implementation we wanted to use for navigation with the robot.

Our Turtlebot3 was equipped with a LIDAR sensor which is capable of scanning 360° with a laser, that could measure distance. It would return data in the form of an array like so:

$$dist = [d_0, d_1, ..., d_{359}]$$

Where d_i is the distance to nearest object at i degrees from the front of the sensor, going counterclockwise. In the first implementation we would simply look at a span of 120° that we would divide into three subcategories:

$$left = [d_{60}, d_{59}, ..., d_{15}]\,, N = 45$$

$$front = [d_{14}, d_{13}, ..., d_0, d_{359}, d_{358}, ..., d_{345}]\,, N = 30$$

$$right = [d_{344}, d_{343}, ..., d_{300}]\,, N = 45$$

The reason we chose to only look forwards, was due to us thinking it would be smarter for the robot to simply turn instead of having to go backwards, since this would allow the robot to achieve an overall higher average linear speed.
Using these cones, we would have to specify what the robot would do in different cases, which we implemented with if/else statements. Each of the abstract cases can be seen in the illustrations below. Note that in Case 1, we would ideally make a sharp swing turn, instead of reversing out of the corner.

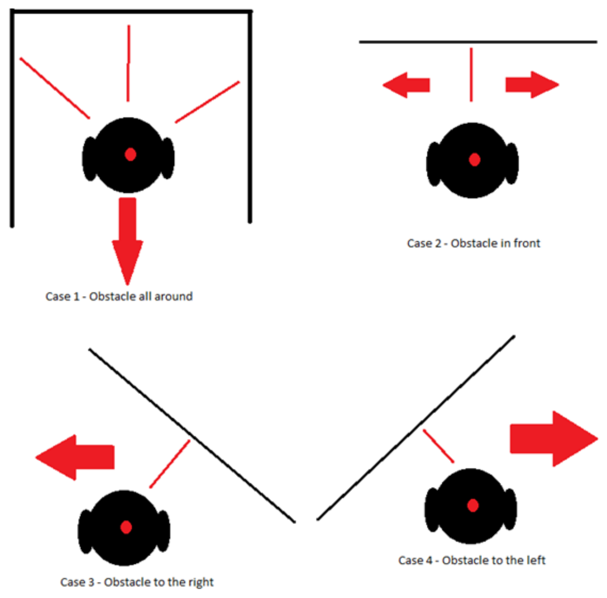
Case 1 - Obstacle all around
Case 2 - Obstacle in front
Case 3 - Obstacle to the right
Case 4 - Obstacle to the left

Fig. 1. Theoretical cases for navigation

Moving on, we could now get to work practically with the real robot, and at first we would need to set up ROS and get familiar with it, before running our own code on it. Up until now it had all been mostly theoretical, which would only get us so far.
ROS is a necessary tool to help us control the robot. In short it is a collection of tools, libraries and conventions that allows us to simplify the creation of complex and robust robot behavior.
Having ROS installed, we needed to get to know it, and to that end we followed a tutorial. We downloaded the Turtlebot3 packages to set up a virtual environment using ROS features, where we would be able to drive a simulation of a turtle around a little screen with our keyboard.

From here on and till the end of our timeframe, we continuously worked on the Robots navigation and optimization. We were presented with a sort of starter build of the code [5] for our robot's obstacle avoidance.
The task at hand first was to update the given code and make the robot read from a different span of angles using the LIDAR.

At first we wanted to implement a way to read from angles -45° to 45° to represent left and right. The current aim was to design according to the following picture [6]

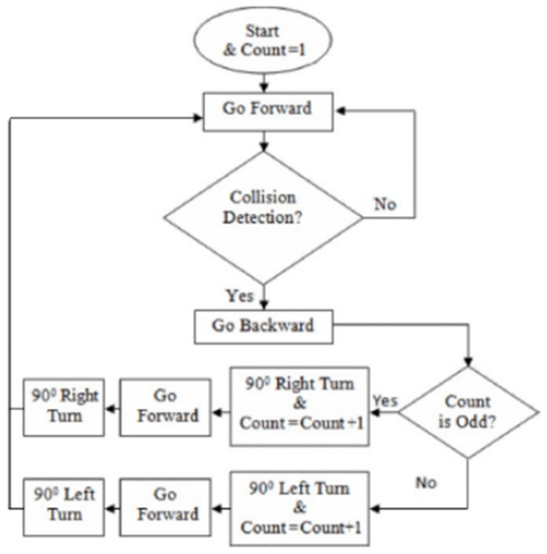

Fig. 2. Simple navigation logic

We got the simple navigation to work and wanted to implement a reading from both Lidar and the RGB sensor in a frequency that made sense. After we got the basics down, had a robot that could do simple maneuvering and send relevant data from both LIDAR and RGB sensor, it was time to optimize the navigation further.
We would start by optimizing how the robot navigates and move on to optimizing the linear speed, where we had to consider the linear speed vs the angular speed. In short, if we were to move faster forwards, we wouldn't be able to make a sharp turn due to the higher speed. Likewise, if we must make a sharp turn, we won't be able to have very high speed. To achieve a higher linear speed whilst turning would want to break the turn into smaller steps. If our robot would be facing an obstacle, we want to calculate how much it should turn. Instead of making the entire turn in one go, we wanted to apply half of the angle and store how much angle there is left to apply it at the next publishing to complete the entire turn. See the following two pictures. [7]

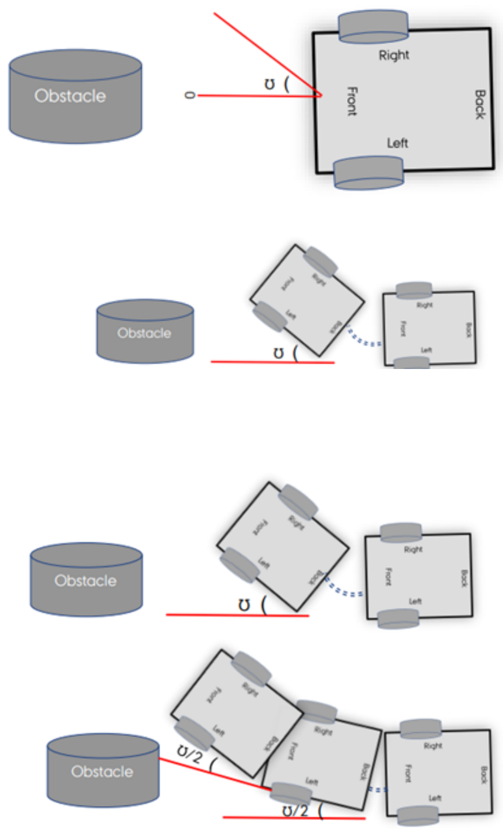

Fig. 3. Speed Optimization for turns

By doing this, we would in theory be able to always achieve a higher linear velocity whilst still avoiding obstacles in our way.
The final two things we wanted to implement was a way to calculate the average linear speed and to optimize that so we would have the highest possible linear speed over the course of the time our robot was driving, and a collision counter, which would increment if the distance to an obstacle was lower or equal to 4cm for the front of the robot or 5cm for other angles.
Our goal was to keep the linear speed as high as possible and the collision counter as low as possible.

B. Experiment setup and Results

1) Method: During our process of making the program for the robot, we have been using a reiterated model called the waterfall model. This model is perfect for the structure of our project. Firstly, we would analyze the requirements.
Then we make decisions based on how we think the robot should operate in different situations (System Design).
Then we would make an implementation of our code (Implementation), and lastly, test the implementation by letting the robot run, and observing it (Testing). Whatever failures we might find with the program, we could then reiterate the process by either going back to system design, if our design choices were wrong, or go to change the implementation if it simply behaved unexpectedly. See the following illustration.

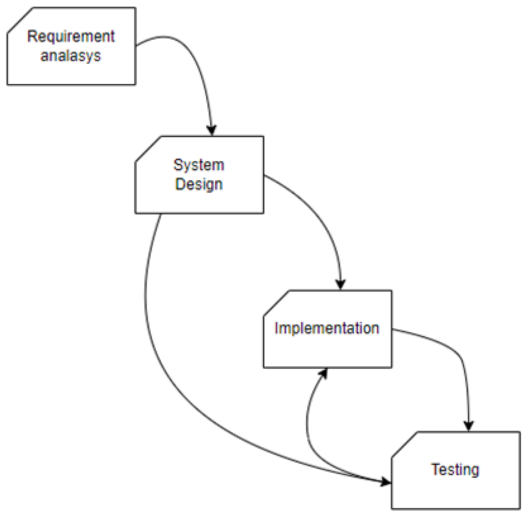

Fig. 4. Waterfall Model (selfmade illu.)

2) ROS Setup: As previously mentioned, before we could begin truly working with the actual robot, we needed some abstraction in place, in this case ROS.
ROS allows us to use packages that have previously been used for working with this robot platform, and then quite simply customize our own node that can run our implementations of code.
To familiarize ourselves with the workings of ROS, we did start by running through the tutorial of virtually running a simulation using the different features of ROS. Learning the uses for ROS topics, packages, nodes and messages.

In practice, we used the turtlebot3_example and turtlebot3_bringup packages from the turtlebot3 repository [8].
The bringup package contained a launch file that ensured the necessities for the turtlebot3 would be running, starting the LIDAR and ROScore. On top of this, we could use the turtlebot3_example's turtlebot3_obstacle node to serve as a base for our own implementation of obstacle avoidance and navigation.

This was handy as it already had the code for setting up the Twist message from geometry_msgs and a topic to allow us to control the motors on the robot simply using this premade message.
Additionally it had an example of how to read the data from

our LIDAR, also setting up a listener for the scan message, from the LaserScan package.
That was about the end of what we kept from the example however, as the actual implementation of the *obstacle()* function is completely changed, both in terms of principle and code. Additionally we did modify the *get_scan()* function, though it's principally similar. The main difference, is that we changed how the filter worked, as invalid readings in our implementation should not be set to 0.

3) Test setup: Following the waterfall model, we would need to test each implementation of code, to determine what changes should be made.
We set up a little obstacle course made from cardboard boxes to function as walls, and some red paper dots on the ground to simulate victims to be found. Our idea was the robot should navigate collision free whilst moving around without either getting stuck in a loop where it would go in circles, or just getting stuck overall.
The robot initially had a few problems with the course. The first struggle was it would register itself too close to an obstacle and immediately try to turn, but as it turned a new obstacle would be spotted and it would only move a little bit before turning again, creating a zig-zag effect.
Another problem arose when we tried fixing the first problem, that it would simply get too close to an obstacle and then get stuck on a wall because we turned too late.
Our optimization and final solutions follow in the next section.

4) Optimization: In our first system design, the idea was that we wanted to maintain the linear speed constant throughout, and simply decide to turn more or less depending on how close we were to an obstacle.
We attempted this by partitioning the LIDARs view, as we have illustrated below, into a front cone and 2 cones on both the left and right side of the bot.
In the illustration (Fig. 5) we have also included some red blobs to explain a scenario where these are obstacles.
We would only decide to turn if something was in the front cone, as we would otherwise be able to continue. When an obstacle was found in front, we would evaluate if there was more room on the left or right side of the bot, to decide which way we should turn. In this case we would turn right, as the left has the closer obstacle. To determine how much we would turn, we then had different cases, where we first determine if "Left 1" has an object within a certain distance, and if it didn't we would check "Left 2". The amount to turn was then determined by a constant (different from "Left 1" to "Left 2") plus a variable fraction, based on the distance to the obstacle. This allowed the amount we turned to be relative to how close an obstacle was, which should in theory give us smoother turns.

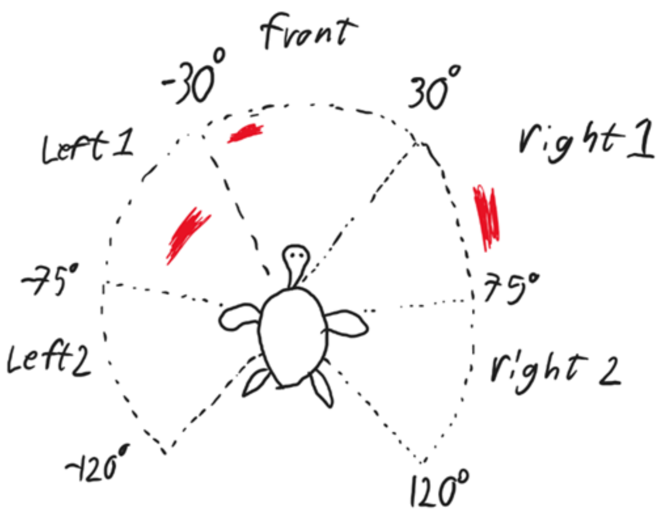

Fig. 5. Design showcase and Example

This implementation however, took many iterations and wasn't quite working, we found the bot would be too stubborn with going forward, and unable to avoid obstacles if it ever got into a corner that was too tight. This would likely be because the decision for turning would be too narrowly minded, as we don't have a full mapping of the terrain, just deciding based on which side had the closer wall. This would eventually get it to turn into corners, but turn too little to avoid a wall on the right, when the left wall was just a little closer. We needed a new edge case to deal with being cornered.

During the later iterations, we noticed some strange decisions from the bot, and decided to add some debugging print statements, to allow us to see which cones saw what. At this moment we realized we had in fact messed up the cones relative to the LIDAR, as we had reversed it entirely, assuming that the array of data from the LIDAR would be ordered chronologically with the direction it spins.
However this turned out to not be the case, and it was in fact reversed. Still we decided to rework the principles on which it should turn slightly.
Moving to the next iteration, we would instead only evaluate $\pm 90°$ from the front, still keeping the front cone, but only using one left and right cone, and still maintaining the fraction based turn decisions.
Of course here we would also add solutions to combat getting stuck in a corner, even if this meant slowing down or backing up slightly. This was done by adding a new check, to determine if there was an obstacle very close to an even narrower front cone, at just $\pm 15°$, this would usually only be triggered if a previous turn had put our nose close to another wall. In this case we would stop moving forward, and simply continue turning until our entire front-cone was cleared of any immediate obstacles. This of course sacrifices a good bit of the average linear speed, however it felt like it was

the better option compared to getting stuck. Additionally we did add another measure if we ended up being truly stuck, by counting the number of repeated turn decisions, if we repeated the same loop too often, we decided it likely meant we were stuck on a corner, as had been observed during testing, and then we could reverse out of this situation, to make a new decision instead.

In this iteration of the code however, we still observed undesired zigzag behavior, particularly when the bot was in-between two relatively close walls, where it had come away from a turn just recently. This turned out to be because we evaluated which way to turn, based on which side of the bot from -90° to 0° on the left or from 0° to 90° on the right, had the closest obstacle. The bug here was that when coming away from a turn, where we are close to a wall, that same wall will be closer than anything on the other side, and so we would keep avoiding this wall that we were already turning away from, potentially nose-diving into the opposing wall. Our fix for this, was to only judge the turn direction on some more forward facing cones, meaning our evaluation for the left-side became -75° to 0° and on the right from 0° to 75° instead.
Additionally we found the bot was still struggling to find it's path through narrower places, and so we decided to narrow the front cone, down to ±25° instead of ±30°. This did largely fix that problem, however it was now even more difficult to avoid small corners that the wheel could get stuck on, especially after recent turns, as the narrow cone when up close, is quite blind.
All in however, we decided this implementation was acceptable, but it was best suited for smooth corners that can be seen from further away, if it hits a narrow corner, it will be able to navigate its way out, and check the corners, but it may take a little longer.

5) Program design: Our program has its roots in the Turtlebot3_example package [5], using a similar structure for the *Obstacle* class, additionally we added the simple RGB Sensor class from our previous work, to initialize the sensor for use in the main loop.

Below we will describe the details of our obstacle avoidance loop, and what each iteration goes through.
The entire code can be found in our GitHub repository [9], in ”Code samples/FinalNavigation.py”.

- **Get LIDAR data**
  Starting off each loop, we must get a new reading from the LIDAR to be able to navigate with current positional data.
  For this we have the *get_scan()* function, which is quite similar in structure to the one from the template.
  Code snippet of the function is on the next page.

```python
def get_scan(self):
    scan = rospy.wait_for_message('scan',
        LaserScan) # Listener for next
        'scan' message from the lidar
    scan_filter = [] # create empty array
        to store filtered lidar data

    for i in range(len(scan.ranges)): #
        iterate through the lidar samples
        scan_filter.append(scan.ranges[i])
            # append the filtered lidar data
            to the list

    for i in range(len(scan_filter)): #
        iterate through the filtered lidar
        data
        curRead = scan_filter[i]
        if curRead == float('Inf') or
            curRead < 0.01 or
            math.isnan(curRead): # check if
            the lidar data is gibberish
            scan_filter[i] = 10.0 # set the
                lidar data to a large number
                (invalid)
    return scan_filter
```

We listen for a new message from the LIDAR and then proceed to filter this data sorting out values like infinity, 0 and NaN, setting these entries to 10 instead, which we know is not going to affect our navigation.

- **Sort and Evaluate Data**
  Having received the filtered data from the *get_scan()* function, we need to organize it and evaluate it, to be able to use it for navigation. (Code snippet from obstacle function)

```python
    scan_read = self.get_scan() # get the
        filtered lidar data

    # Discard readings beyond the scope we
        want to work with here (take from
        -90 to +90 deg)
    lidar_distances = scan_read[:90][::-1]
    lidar_distances.extend(scan_read[270:][::-1])
        # Store our data from left to right
        view

    #Partition readings into cones for
        evaluating navigation
    frontCone = lidar_distances[65:115] #
        -25 deg to +25 deg
    rightCone = lidar_distances[115:] # 25
        deg to 90 deg
    leftCone = lidar_distances[:65] # -90
        deg to -25 deg

    rightEval = min(rightCone)
    leftEval = min(leftCone)
    frontEval = min(frontCone)
    narrowFront = min(frontCone[15:45]) #
        Front cone of +-15 deg
    leftTurnE =
        min(lidar_distances[25:90]) # Eval
        for left side turn
    rightTurnE =
        min(lidar_distances[90:155]) #
        right turn eval
```

We do this by creating a new array, intended to store our readings in order from left to right, starting at -90 degrees up to 90 degrees. Since our LIDAR returns an array of 360 readings, one from each degree of angle, we take the first 90 readings which are on the left side of the bot, and reverse this, because it by default is stored right to left. And we then add to that, the final 270th reading up to the 360th reading, which would be on the right side of the bot, also reversing this.
After this we simply use the new array, to create cones for viewing zones of the bot, and then proceeed to store the readings of shortest distance in each of these relevant areas, to use for evaluation in our navigation.

- **Check for Victims**
In order to check for victims, we read the current red and blue values from the RGB sensor, as the tags we are looking for a red, we know to look for a higher value of red and lower value of blue. These new values are compared to an older baseline value, to avoid reading double. The first baseline is made in initialization of the obstacle avoidance loop.

```
    newRed, dummy, newBlue =
        sensor.get_rgb() # Get values for
        victim detection (red and blue)

    if newBlue < curBlue * 0.75 or newBlue
        > curBlue * 1.25: # Only care about
        the RGB readings if the data is 25%
        different from last baseline
        (prevents double counts)
        curBlue = newBlue # new baseline
        if newRed > 400 and newBlue < 200:
            # Check if we are currently over
            a red tag
            victims += 1
            rospy.loginfo('Victim found,
                total count: %d', victims)
```

We've decided that 25% was a good amount to allow the readings to fluctuate, and whenever it has changed more than this, it's because we've either travelled over a tag we need to read, or we've just come off a tag we've already read. This means we can create a new baseline here, to compare future readings with. When this happens, we need to also check if we did come across a new tag, or if we went off of one, so this is where we use the new readings from the sensor, to determine if we are over a red tag, and if so increment our counter.

- **Check for collision**
Checking for collisions we need to evaluate on the LIDAR readings, we've decided to simplify the check quite a lot, and so we simply check if the robot is within 5cm + the LIDAR error.

```
if min(scan_read) < 0.05 + LIDAR_ERROR: #
    If something is within 5 cm (+ error)
    count up collision (unless on cooldown)
   if collision_cd < 1:
      collision_count += 1
      collision_cd = 10
      rospy.loginfo('Collision detected,
          total collisions: %d',
          collision_count)
collision_cd -= 1 # Cooldown for loop
    cycles on colissions
```

To avoid any double counting, we have implemented a cooldown counter, that decreases with every loop-cycle, so we shouldn't read multiple collisions in the same corner, as we would be gone by then.

- **Navigation control**
The navigation controls consists of a few steps, firstly we need to check for a flag to see if we have been marked as cornered from a previous loop, and if so, we must continue to turn until our front-cone is clear of immediate obstacles.
If we haven't been marked as stuck, we can move on to see, if there are obstacles in front of us, within our set FRONT_SAFE_DIST, and if we are clear, we jump to the else case, which simply lets us continue forward at full speed.
Should we instead have obstacles, we need to turn to avoid them, so we immediately decide which way we need to turn, comparing the evaluation variable for the left side, to the right side, and storing the decision in another flag called rightTurn.
Moving on, we check our narrow front cone, for any very close obstacles, this is to make sure we have room to turn at all, and if we don't, we set the cornered flag and proceed to back up slightly, and the next n-loops would then keep us turning until we are clear to move forward.
If all instead goes well, and our narrow front is clear, we either go to the left or right turn case depending on the flag, and here we firstly have a default case that updates the Twist message variables through
driveUpdate(angular, linear), to a small turn.
We then move on to check if the obstacle is within our set SAFE_STOP_DISTANCE, to determine if we should turn more than the default. If we should, we set a new driveUpdate, with a variable amount of turning, depending on the distance to the nearest obstacle.
The final detail of the control loop, is that we have added counters to check for repeated entries into the turning loops. This is to combat getting stuck on corners, so if we enter the same loop to many times in a row, we are most likely stuck, and so we reverse instead of continuing to turn into the wall.
Once all of the above has been executed, we have

the correct Twist message values as is updated by the *driveUpdate(a, l)* function, and so we can publish these to ROS and have the robot follow our control logic.

```
if(cornered): # If we've been cornered, this lets us turn until we find a way out
    leftLoop = 0
    rightLoop = 0
    if(frontEval > EMERGENCY_STOP_DIST * 1.6):
        cornered = False
        driveUpdate(0.0, LINEAR_VEL)
    else:
        if(rightTurn):
            driveUpdate(-1.0, 0.0)
        else:
            driveUpdate(1.0, 0.0)

elif frontEval < FRONT_SAFE_DIST: # If obstacle in front, turn
    # Turn direction decision
    if(leftTurnE <= rightTurnE):
        rightTurn = True
    else:
        rightTurn = False

    if(narrowFront < EMERGENCY_STOP_DIST): # Determine if cornered
        cornered = True
        driveUpdate(0.0, -0.15)

    elif(rightTurn): # Need to turn, not cornered
        #Turn right
        driveUpdate(-0.7 - 0.16/leftEval, LINEAR_VEL) # Default turn a little if something is in the way
        rightLoop += 1
        leftLoop = 0
        if(rightLoop > 10): # if we seem stuck on a corner we can't see, reverse
            driveUpdate(1.4, -0.4 * LINEAR_VEL)

        elif(leftEval < SAFE_STOP_DISTANCE):
            driveUpdate(-1.0 - 0.22/leftEval, LINEAR_VEL) # Turn depending on how close an obstacle is
    else:
        #Turn left
        driveUpdate(0.7 + 0.16/rightEval, LINEAR_VEL) # prev 0.5 + 0.15
        leftLoop += 1
        rightLoop = 0
        if(leftLoop > 10):
            driveUpdate(-1.4, -0.4 * LINEAR_VEL)

        elif(rightEval < SAFE_STOP_DISTANCE):
            driveUpdate(1.0 + 0.22/rightEval, LINEAR_VEL)

else: # Continue straight, if no obstacles ahead
    leftLoop = 0
    rightLoop = 0
    driveUpdate(0.0, LINEAR_VEL)
self._cmd_pub.publish(twist) # Publish our decision on what to do
```

- **Update variables**
  To finalize the loop, we just need to keep track of the average speed of the robot, this is done using the linear speed we've saved to the twist.message and a simply counter for how many loops we've been through.

```
    #Average linear linear updates here
    self.accumulated_speed += abs(twist.linear.x)
    self.speed_updates += 1
    self.average_speed = self.accumulated_speed / self.speed_updates
```

IV. Discussion

The thing that took a big toll on the progress of our development was the LIDAR sensor. In this section our main topic will be the problems regarding the sensor and how we tried to fix it and work around it. As mentioned earlier we encountered a lot of problems with the LIDAR sensor. One of them came out to be our understanding of how it worked wasn't correct. When addressed the problem resolved itself rather quickly.
However, the sensor wasn't without fault either. A lot of the readings we receive from it would be faulty. In those situations, our *get_scan()* method would make that reading a high number so it would be evaluated to something far away. We tried setting the value to zero, but this resulted in our cones readings would get skewed and make the robot turn and stop a lot more than what was acceptable. The main problem was that these faulty readings seemed almost random, and could cause navigation issues that weren't fixable.

Our initial goal of achieving a robot that would move autonomously, avoid obstacles, and detect victims was an overall success.
We managed to successfully implement our initial and later improved theories. With that in mind it is likely possible to achieve better navigation than what we've accomplished. One way to make some improvements would be to make the robot more "aware" of how big it is. We could make the robot evaluate in a radius around itself 360 degrees, instead of our front-focused approach. The radius would have to be a bit bigger than the actual radius of the robot to mitigate potential problems when turning around. This would then be its own function that would run before anything else in the control loop. If a reading from this is returned and is within a specified safety range, it would then turn and make the front as far away as possible from the found obstacle.

A way to improve the navigation further could also have been to map the surroundings using something like SLAM (Simultaneous Localization and Mapping), which is a method used to map and localize our robot in the surroundings. This way we could let the robot map the unknown environments and thus let the robot make informed decisions based on the map. This might however be a bit out of the scope of the project presented to us, but none the less be a way to improve navigation.

V. Conclusion

Even though we've been working with very simple programming and robots, the principles of working on a project that is a bit bigger in size, requires a lot of time investment. We've learned the foundation of understanding more complex robot behavior, and the importance of being structured and working systematically instead of being all over the place and hoping for the best.
ROS provides a structure and playground for working on these complex projects and has improved our capabilities to work on further projects which might be even more complex than this.

We've presented a way to implement basic autonomous navigation which resulted in reasonably good obstacle avoidance and average speed. It works best in more smooth and spread-out environment in terms of obstacles, whereas in tight spaces it could face problems where it couldn't determine whether there was enough space.

The LIDAR sensor presented us with some issues and would get bad readings that had to be mitigated. We've managed to work around this as best as possible, and the navigation functions consistently despite this.
We've also discussed ideas to improve the navigation, using different principles than our implementation. One way to do it, would be to utilize the full 360 degrees of scanning, and accept closer calls with obstacles, and less smooth turns. Another more advanced way, is to continuously map the environment, this should also eliminate the significance of bad readings.

Overall, our navigation system implementation was rather effective. We maintained a high linear velocity whilst still avoiding different obstacles in a smooth way.

VI. Personal Contributions

During the entire process we discussed internally how we were supposed to divide the many tasks between us.
We didn't want to divide us, so we did a lot together in intense collaboration. One might have written some code from home before we met and then we would discuss it thoroughly before we just implemented it on the robot. Or one might have had an idea of how to calculate a certain thing, and then we would discuss it. When working with the robot and the VM we mainly used Steffens computer due to Daniels having some problems setting it up and it cost a lot of time we didn't have.

In the late stages, Daniel focused on writing the report and keeping a birds eye view of the structure, while Steffen kept on improving the robot and fixing the problems that we've encountered.

Regarding the paper, Steffens focal point has been the following:

- Section II Specifications
- Section III-B4 Optimization
- Section III-B5 Program design

While Daniel was focused on the following:

- Section III-A Design an Implementation
- Section III-B1 Method
- Section III-B2 ROS setup
- Section III-B3 Test setup
- Section IV Discussion
- Section V Conclusion

References

[1] Robotis e-Manual - TurtleBot3 Specifications, visited 03-05-2023, https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#specifications.
[2] Robotis e-Manual - LDS-01 Specifications, visisted 03-05-2023, https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/
[3] Raspberry Pi 3 Model B Documentation, visisted 03-05-2023, https://www.raspberrypi.com/products/raspberry-pi-3-model-b/
[4] ISL-29125 RGB Sensor Datasheet, visisted 28-05-2023, https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/isl29125.pdf
[5] Turtlebot3 Package, template code, visisted 28-05-2023, https://github.com/ROBOTIS-GIT/turtlebot3/blob/master/turtlebot3_example/nodes/turtlebot3_obstacle
[6] BrightSpace Week 7, Navigation Slides
[7] Brightspace Week 9, Speed Optimization Slides
[8] Turtlebot3 master repository, visisted 28-05-2023, https://github.com/ROBOTIS-GIT/turtlebot3/tree/master
[9] Our GitHub repository, visisted 29-05-2023, https://github.com/Dan-jpg2/CTP-1