EPO-4

Final report

19 June 2015

Delft University of Technology

Group A7

Bas Generowicz	4029542
Enzo Claveau	4156935
Tiamur Khan	4247329
Ziar Khalik	4098250

TUDelft Delft University of Technology

Challenge the future

Contents

Abstract

In this report the second part of the EPO-4 project is elaborated. The report consists of two chapters which discuss the localisation of the car (named KITT) and the implementation of driving KITT autonomously. By using the time difference of arrival of an audio beacon at different microphones it is possible to locate KITT. Two different design concepts were developed in order to let KITT drive autonomously. Because of limited test time not both the deisgn concepts could be implemented effectively, therefore it was decided to focus on one solution. Both designs will be addressed in this report.

Chapter 1

Introduction

1.1 Introduction

The final challenge of the project *Autonomous driving challenge* consists of a number of tasks. To pass the final challenge, KITT must drive from point A to point B with an accuracy of atleast 30 cm. The more advanced tasks include driving to intermediate way-points and the addition of obstacles on the field. In order to complete these tasks a positioning system must be designed and a control system to drive the car to certain points must be implemented.

On the field there are 5 microphones. KITT produces audible signals, which are then received by the microphones. The time difference of arrival (TDOA) for each microphone is then analysed to find the location of the car. With the help of the localisation system, a control unit was designed that drives the car autonomously. The report will discuss the performance of the localisation system as well as the design and implementation of the control system.

Chapter 2

Localisation

2.1 Introduction

To make it possible to locate KITT, 5 microphones are used together with a speaker mounted on KITT which is transmitting an audio beacon. By performing channel estimations it is possible to retrieve the time difference of arrival (TDOA) of the audio beacon at the different microphones. With this data it is then possible to locate KITT on the field. The first section of this chapter will explain how the TDOA's are retrieved and in the second section the algorithm is explained that is used to retrieve the location of KITT when knowing the TDOA's.

2.2 Time Difference of Arrival

2.2.1 Specifications

In a previous project a Matlab algorithm was made for estimating a channel. In short, using knowledge of the transmitted signal $x[n]$ at the loudspeaker and a measured signal $y[n]$ at the microphone, an estimate is made of the audio channel $h[n]$ that has filtered $x[n]$ using convolution $y[n] = x[n] * h[n]$.
From this the TDOA (time difference of arrival) was found and using the speed of sound this was related to the distance between the 2 recordings. A plan was made to test this using the given hardware for this project.

2.2.2 Design Plan

1. Switch the audio beacon on using standard settings
2. Make recordings of the transmitted signal at each microphone separately (referred to as a "clean recording", used as $x[n]$ above)
3. Place the beacon on a known location on the field and record data from all the microphones
4. Use the previously made script to estimate the channel and detect the first dominant peak for each microphone
5. Compare the TDOA's between the microphones and evaluate results

2.2.3 Design Results

This process as stated had to be done for all 5 microphones but to more clearly show the results only the process for microphone 1 is shown.

The clean recording for microphone one can be seen in figure 2.1. Since it is a recording close to the microphone it is to be expected that there is little influence from background noise, this can be observed during the silence interval of the signal when it is almost zero.

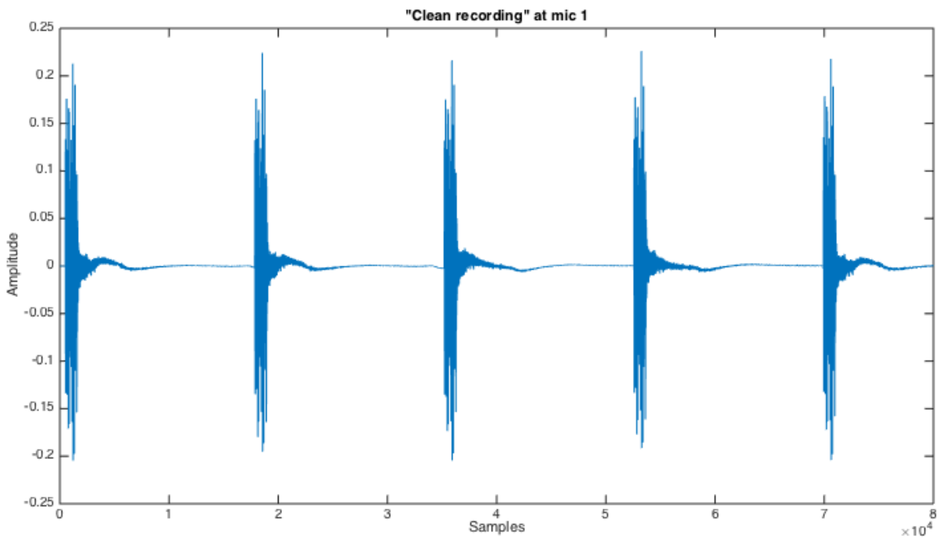

Figure 2.1: "Clean recording" at microphone 1

In Matlab an automatic script was created to truncate the signal for one cycle of the repeated signal and to take away leading and trailing small values so that the signal is as short as possible. This truncated signal can be seen in figure 2.2 and was used as the reference signal for the channel estimation algorithm.

The beacon was then placed on a known location on the field and data was recorded from all the microphones. For a known point (216,88) a data segment is recorded at all 5 microphones ($y_i[n]$). This together with the clean recordings $x_i[n]$ are used to estimate the channel. For microphones one and two the channels can be found in figure 2.3 and 2.4, respectively.

To extract useful information like a TDOA, 2 channels need to be compared for 2 different microphones. This is done in a couple of steps

From figure 2.5:
Speaker 1 is at location (0,0).
Speaker 2 is at location (0,413).
The car is at point A (216,88).

To test the obtained values from figures 2.3 and 2.4 the actual values were first calculated.

The distance from speaker 1 to the car at point A is 123.7 cm
The distance from speaker 2 to the car at point A is 318.8 cm
The difference between these distances is 195.1 cm

Now when looking at the obtained data for h1 the first dominant peak is found to be at the 6756$^{\text{th}}$ sample. For h2 the first dominant peak is found at the 7035$^{\text{th}}$ sample. As is expected the peak occurs earlier for mic 1 since it is closer to point A. Next a calculation is done to find the

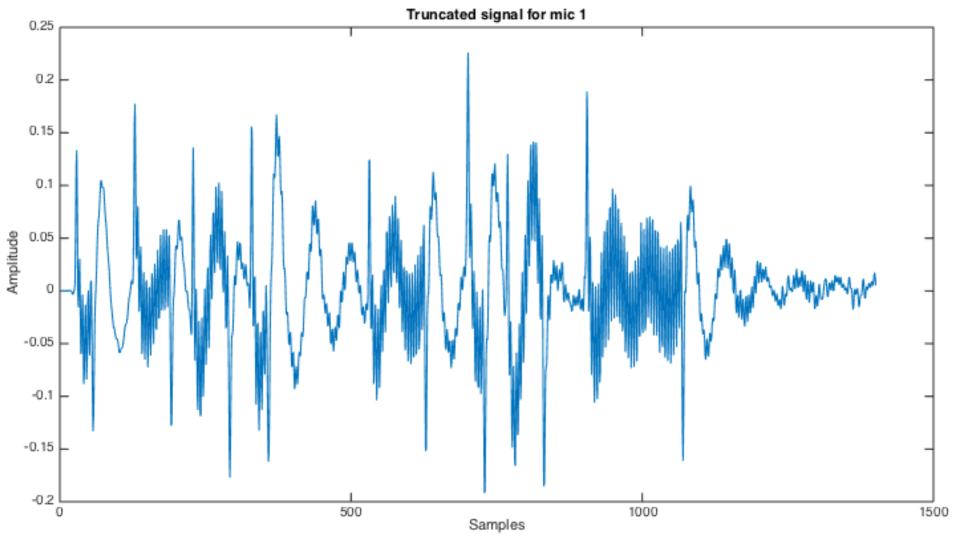

Figure 2.2: Truncated signal for microphone 1

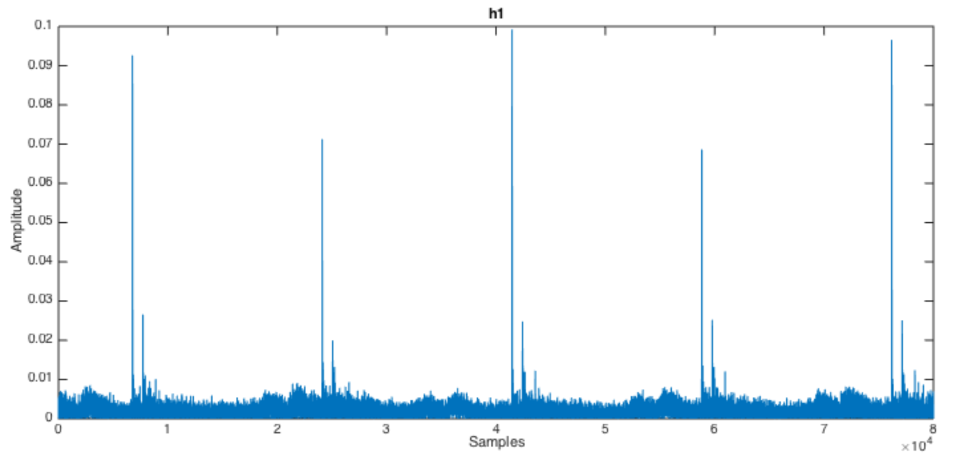

Figure 2.3: h1[n] for mic 1

difference in distance beween the 2 microphones and point A:

$$\frac{(7035-6756)[samples]*340[m/s]}{48000[samples/sec]} = 197.6[cm]$$

The calculated difference in distance is 197.6 cm whereas the actual value is 195.1 cm. When similar calculations were done for different microphone combinations an average fault of 5.6 cm was found. This seems to be pretty reliable.

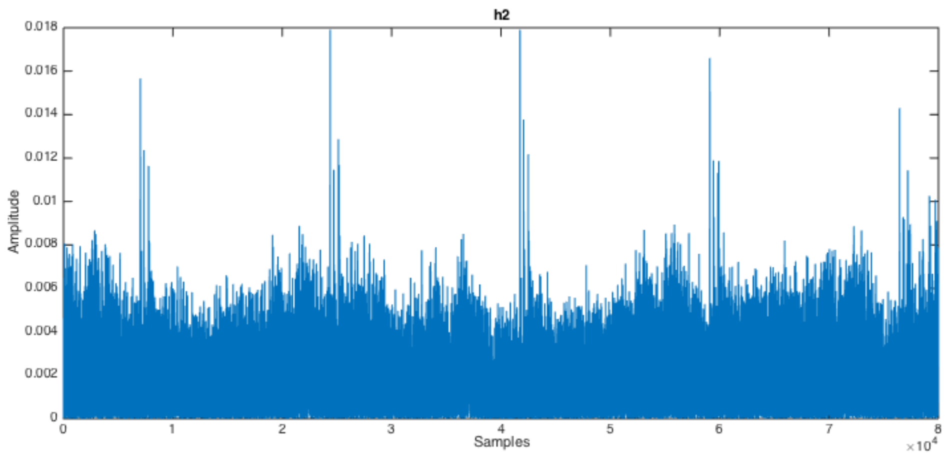

Figure 2.4: h2[n] for microphone 2

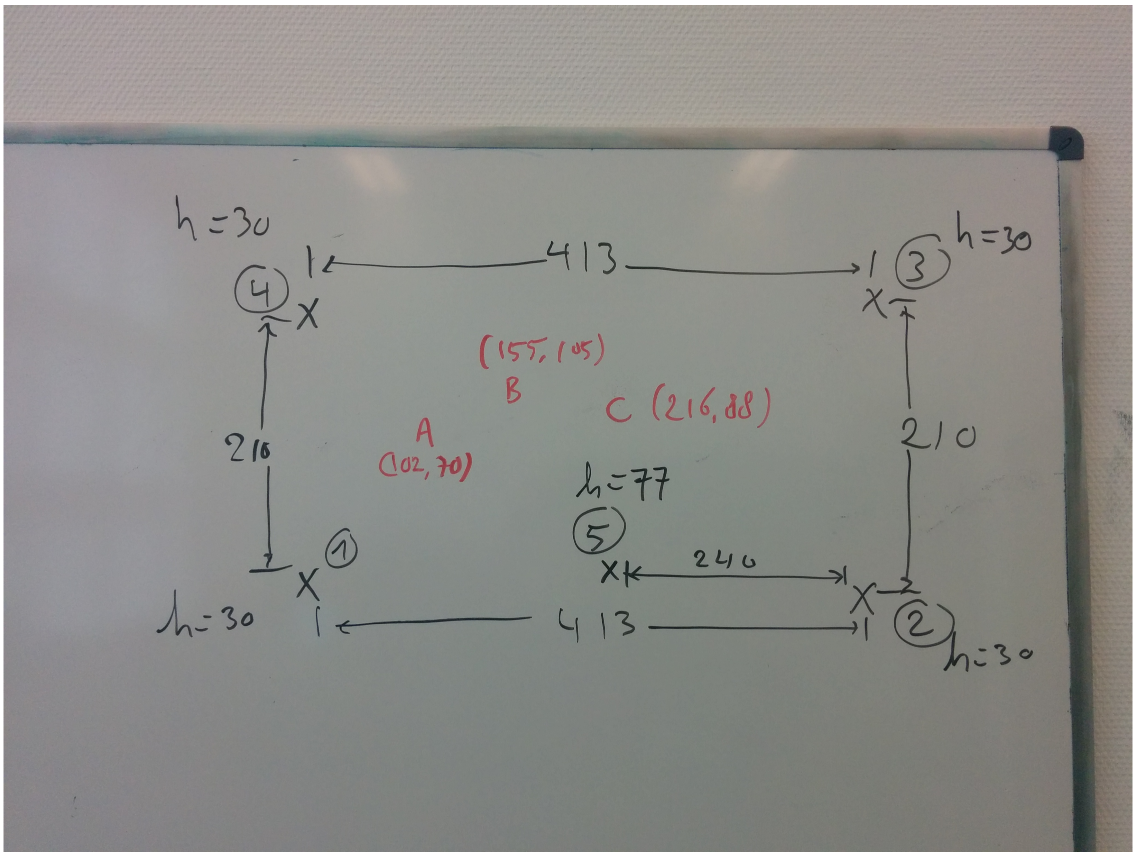

Figure 2.5: Overview for experimentation

2.2.4 Verification

When applying the above knowledge to the final set-up using 5 microphones some problems occurred. The localisation was accurate when the car was close to the center of the set-up seen in figure 2.5, however if the beacon was placed more to one side (say point A) the coordinates given from the localisation became inaccurate and inconsistent. When looking at the channels of the far away microphones the peaks became hard to distinguish from noise. The first idea to fix this was to optimise the peak finding function.
Initially the highest point was found for h1, In h2-h5 the highest point was found within a certain time from the h1 peak. As an example is figure 2.6, if the peak of h1 were to be found at 1.6 or 4.6, for h2 there would be no peak around that point and a useless value would be found.

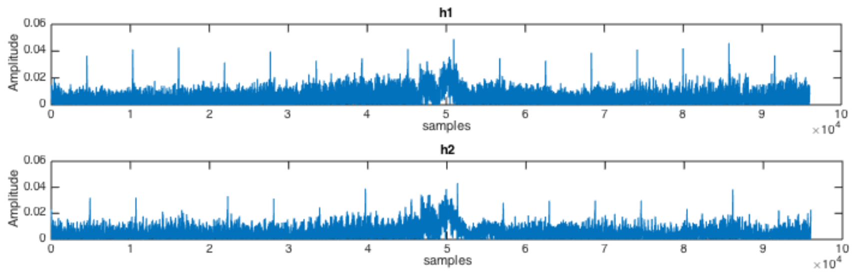

Figure 2.6: Channels of microphone 1 and 2

As a solution to this issue the maximum value of every h is taken and modulo's with the known period of the signal, this way you find the value at which each deviates from this number. These can then be compared and a more reliable TDOA can be made.
This solution was still not as reliable as necessary. When looking at figure 2.6 the amount of noise can make it very hard to distinguish the peaks. After looking for a solution for a long time the fault was found to lie in the channel calculation. In the calculation frequency domain deconvolution takes place $H = \frac{Y}{X}$. If there is a lot of noise on X (small values) this translates to big values in H. In 2.7 every value of X smaller than a threshold number is filtered out of H. This difference can be seen when comparing the same signal in the 2 plots. Using the improved version reliable results were obtained.

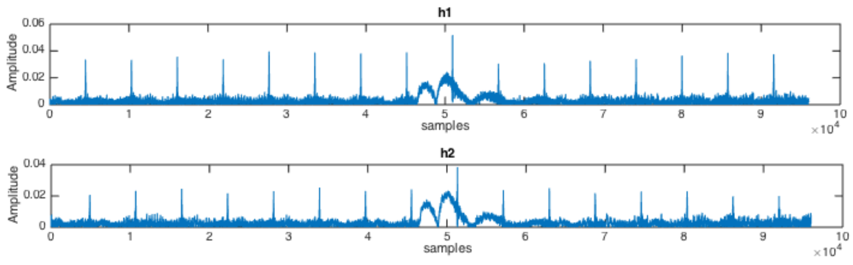

Figure 2.7: Channels of microphone 1 and 2

2.3 Localisation Algorithm

2.3.1 Specifications

Now that the multichannel data is measured and the time delay of arrivals between each pair of microphones are calculated, the car can be located. The localisation is done by using the algorithm given in [1]. This gives an estimation of the location of the car. This estimated location has to be determined as accurately, preferably within a radius of 30 cm, and fast as possible.

2.3.2 Design Plan

First thing to do is to implement the given algorithm for the localisation. The algorithm is explained in appendix B of [1]. Basically it comes down to the following matrix calculations when using four microphones:

$$\begin{bmatrix} 2(\mathbf{x_2}-\mathbf{x_1})^T & -2r_{12} & & \\ 2(\mathbf{x_3}-\mathbf{x_1})^T & & -2r_{13} & \\ 2(\mathbf{x_4}-\mathbf{x_1})^T & & & -2r_{14} \\ 2(\mathbf{x_3}-\mathbf{x_2})^T & & -2r_{23} & \\ 2(\mathbf{x_4}-\mathbf{x_2})^T & & & -2r_{24} \\ 2(\mathbf{x_4}-\mathbf{x_3})^T & & & -2r_{34} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} r_{12}^2 - \|\mathbf{x_1}^2\| + \|\mathbf{x_2}^2\| \\ r_{13}^2 - \|\mathbf{x_1}^2\| + \|\mathbf{x_3}^2\| \\ r_{14}^2 - \|\mathbf{x_1}^2\| + \|\mathbf{x_4}^2\| \\ r_{23}^2 - \|\mathbf{x_2}^2\| + \|\mathbf{x_3}^2\| \\ r_{24}^2 - \|\mathbf{x_2}^2\| + \|\mathbf{x_4}^2\| \\ r_{34}^2 - \|\mathbf{x_3}^2\| + \|\mathbf{x_4}^2\| \end{bmatrix} \tag{2.1}$$

Here the $\mathbf{x_i}$ are the locations of the microphones, r_{ij} are the range differences between two microphones, d_i are the distances from the car to microphone i and $\mathbf{x}$ is the location of the car. The locations of the microphones are known and the range differences can be calculated from the TDOA's. The location of the car, $\mathbf{x}$, can be calculated by inverting the first matrix and multiplying it with the third matrix, so $\mathbf{x} = \mathbf{A^{-1}b}$. The algorithm can easily be extended to work with any number of microphones with any height.
The localisation algorithm will initially be tested using simulated Matlab data, which means that the range differences are calculated by hand. The following things will be considered:

- Do height differences between the microphones affect the localisation?
- How sensitive is the algorithm to differences in the speed of sound?

2.3.3 Design Results

First the algorithm, which is found in [1], is tested with simulated Matlab data. Four microphones are used in a grid of four by four, where there is a microphone on every corner. Next step is to create the matrix containing the range differences by hand. For example, the range difference matrix for the point (0,1) would be:

$$\begin{bmatrix} 0 & 1-\sqrt{17} & -4 & -2 \\ 0 & 0 & \sqrt{17}-5 & \sqrt{17}-3 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.2}$$

When this matrix is given as input to the algorithm, it will return the location of exactly (0,1). The result is plotted in figure 2.8. When doing the same thing for the set up of the recordings for point A, which was (102,70), the algorithm gives back the exact location. Which means that it works as intended.
Next thing to determine is whether the heights of the microphones affect the localisation. The four by four grid is extended in height, so the grid is four by four by one now. When first two

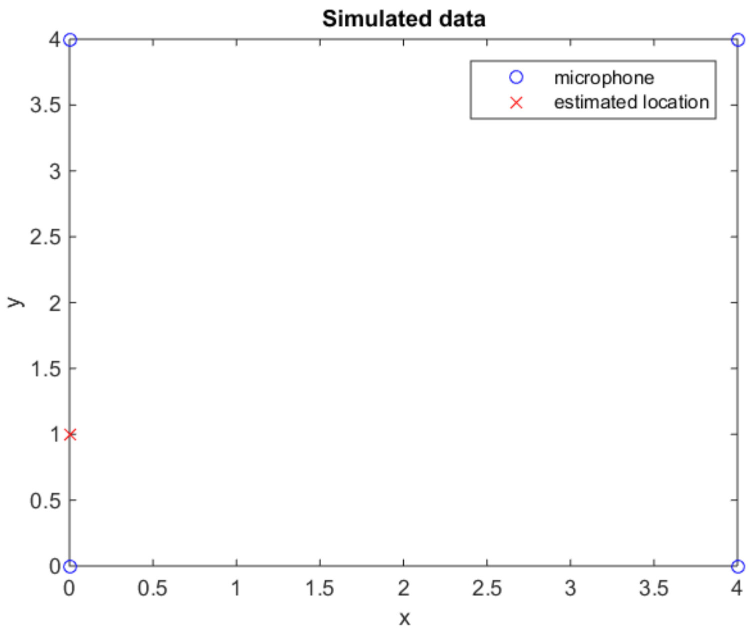

Figure 2.8: Estimated location using the simulated Matlab data

microphones are heightened by 1, the new range difference matrix will be:

$$\begin{bmatrix} 0 & \sqrt{2}-\sqrt{18} & \sqrt{2}-5 & \sqrt{2}-3 \\ 0 & 0 & \sqrt{18}-5 & \sqrt{18}-3 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.3}$$

Without considering the heights of microphones 1 and 2, the algorithm gives a location of (0,1.125) instead of (0,1). So small height differences between the microphones will lead to a small error in the estimated location. Because the sensitivity is small, the algorithm won't be extended to 3D. When the speed of sound is altered by a 1% variation, there is almost no difference in the estimated. So this is also no problem when estimating the location.
Now that the location estimation works with the simulated data, it is time to test it with the data acquired from the recordings. First thing is to test how accurate the location estimation works with four microphones. The TDOA matrices with four microphones for the points A(102,70), B(155,105), C(216,88), D(300,50) and E(413,105) is given as an input for the localisation algorithm. The first couple tries didn't give very good results. However, it had nothing to do with the localisation algorithm. To make sure that the multichannel data recording was correct, new data was recorded. This time the results did meet with the expectations. The result is shown in figure 2.9. It is seen that the estimated locations that are close to the desired location have an accuracy of maximum 10 cm, which complies with the specifications. It can also be seen that the estimated for point E is not accurate at all. This is because the algorithm doesn't work when the car is located at exactly the same distance from two microphones. So this was also expected.

The fifth microphone is at a different height from the other four microphones. Because the estimated locations are already accurate enough, the data of the fifth microphone doesn't have to be used. However from the testing with the simulated data it was concluded that the algorithm wasn't very sensitive to small height differences. Also it is better to use all the data available, so it was decided to experiment how the fifth microphone would affect the estimated location if the height was neglected. The same thing was done again, but this time with the fifth microphone included. Figure 2.10 shows the results. It can be seen that the estimated locations are a bit more accurate. It was decided to use all five microphones to estimate the location of KITT at the final

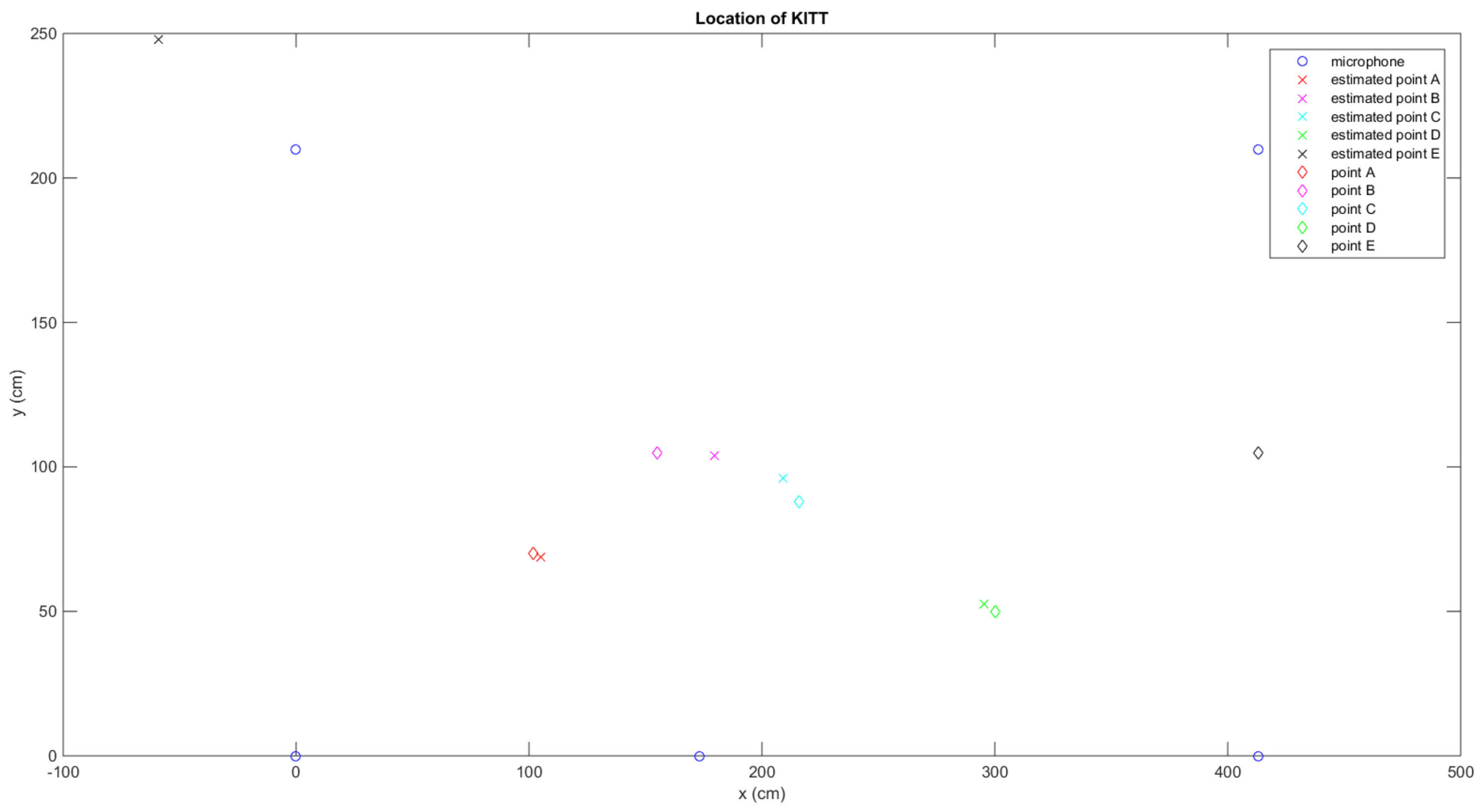

Figure 2.9: Estimated location of KITT at points A,B,C,D and E using four microphones.

challenge.

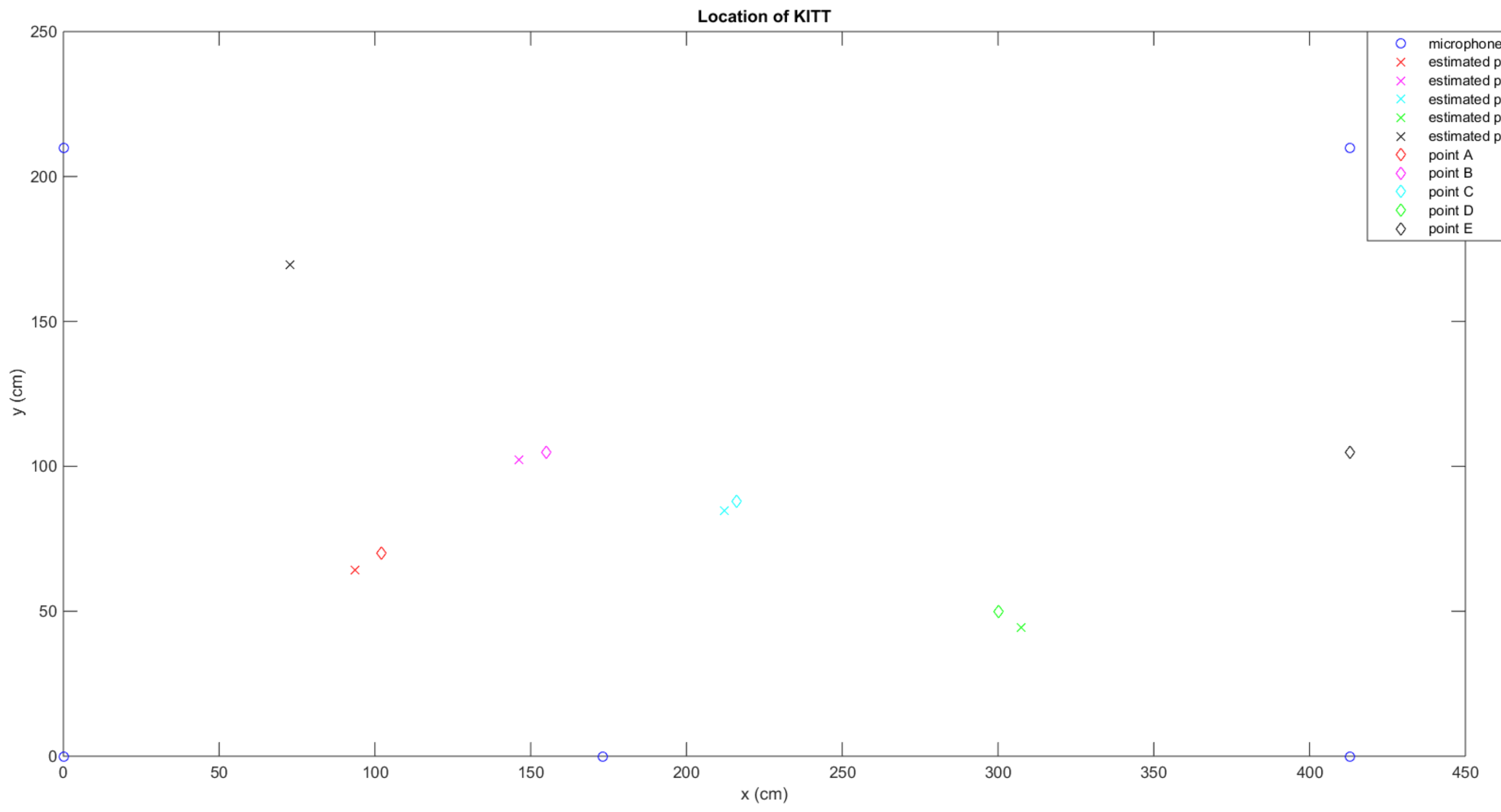

Figure 2.10: Estimated location of KITT at points A,B,C,D and E using five microphones.

Chapter 3

Autonomous Driving

3.1 Introduction

In this chapter will be explained how the implementation for autonomous driving has been achieved. Two different design concepts were developed simultaneously which are both discussed in this chapter. Design 1 needed a shortest path algorithm adapted to the design. This algorithm will also be discussed at the end of this chapter.

3.2 Autonomous driving

3.2.1 Design 1

Design concept

The purpose of this design is to implement a robust system which takes into account the uncertainty of the localisation of KITT. To achieve this, but yet keep the design simple a few restrictions are made which form the basis for the design:

- An imaginary grid is placed on top of the field consisting of n by n squares of a given length dx.
- The square length dx is based on the maximum turning radius of KITT.
- Each square has its own coordinates and the location of KITT is related to these coordinates instead of a precise location.
- KITT can only move around the field in straight lines and not diagonally.
- KITT can only move forward, turn left and turn right.
- When KITT makes a turn the next grid position of KITT is as shown in figure 3.1.

Because the location of KITT is reduced to knowing the grid coordinate in which the car is at a moment, the control of KITT is not sensitive for small errors in the localisation. Another benefit is that this concept is not complex but limited, which makes it easy to test and adjust in the little time that is available to test. By using way points located on the grid (as explained in 3.4 it is possible with this design to perform all three the challenges without making changes to the code (only new way points have to be calculated once an object has been detected). One disadvantage is that because there is assumed that KITT can only drive in a straight line, the car really has to do this. Else the location of the car will not be as expected and the solution for compensating for these errors will make the design more complex. Therefore it is important for KITT to drive straight and to make sure this happens a steering compensation is implemented based on the angle of KITT, see 3.2.1.

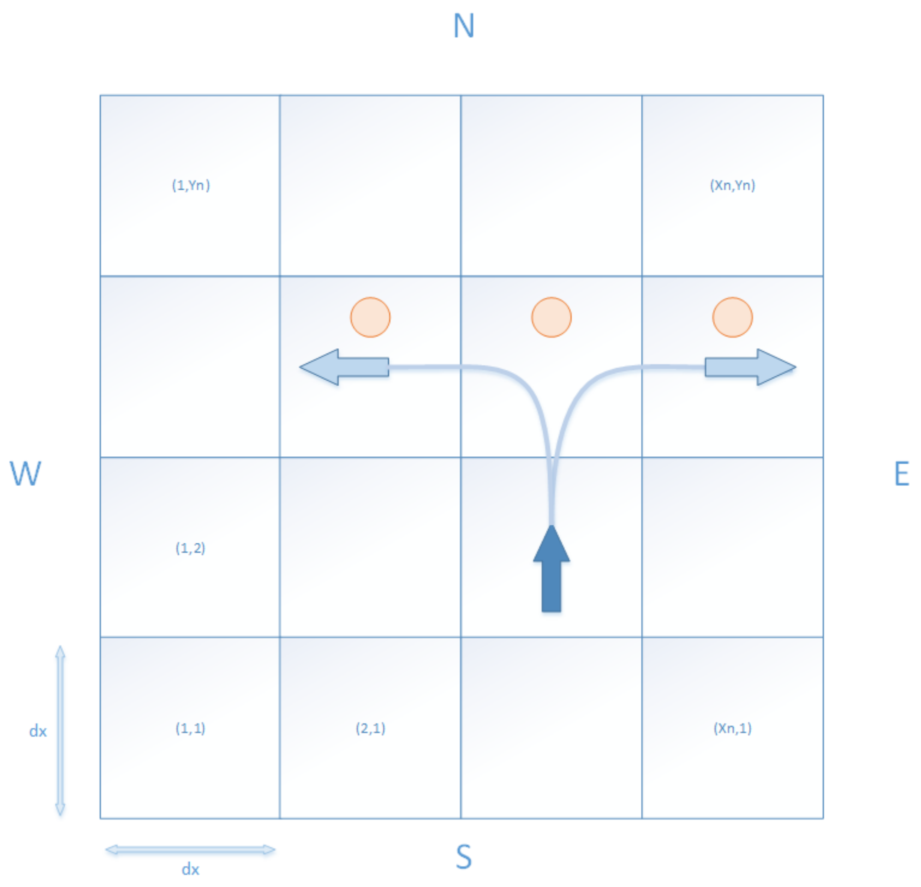

Figure 3.1: The grid with the location of KITT when making a turn. The dark blue arrow stands for KITT and the direction in which it is facing and the light blue arrows are the next locations of KITT in case KITT makes a turn. The orange marks are the possible next way points.

Implementation

The autonomous driving implementation is done by means of a finite state machine (FSM). This FSM is depicted in figure 3.2. The calculated path that KITT has to follow is already known and given by way points located on the grid and they serve as one of the inputs of the FSM to be able to make decisions.

Making the right action

In order for KITT to make the right decision the way point coordinates and the coordinates of KITT are compared with each other. An important part of this is to know the direction in which KITT is facing, denoted with North, East, South and West. For example: when heading North only the y coordinates have to be compared with each other because it is known that the x coordinates are the same since KITT can only drive in a straight line. When a way point has been reached the next way point can either be in one of the squares in front of KITT as shown by the orange marks in figure 3.1. The precise location of the way point determines if KITT has to make a turn or has to move forward.

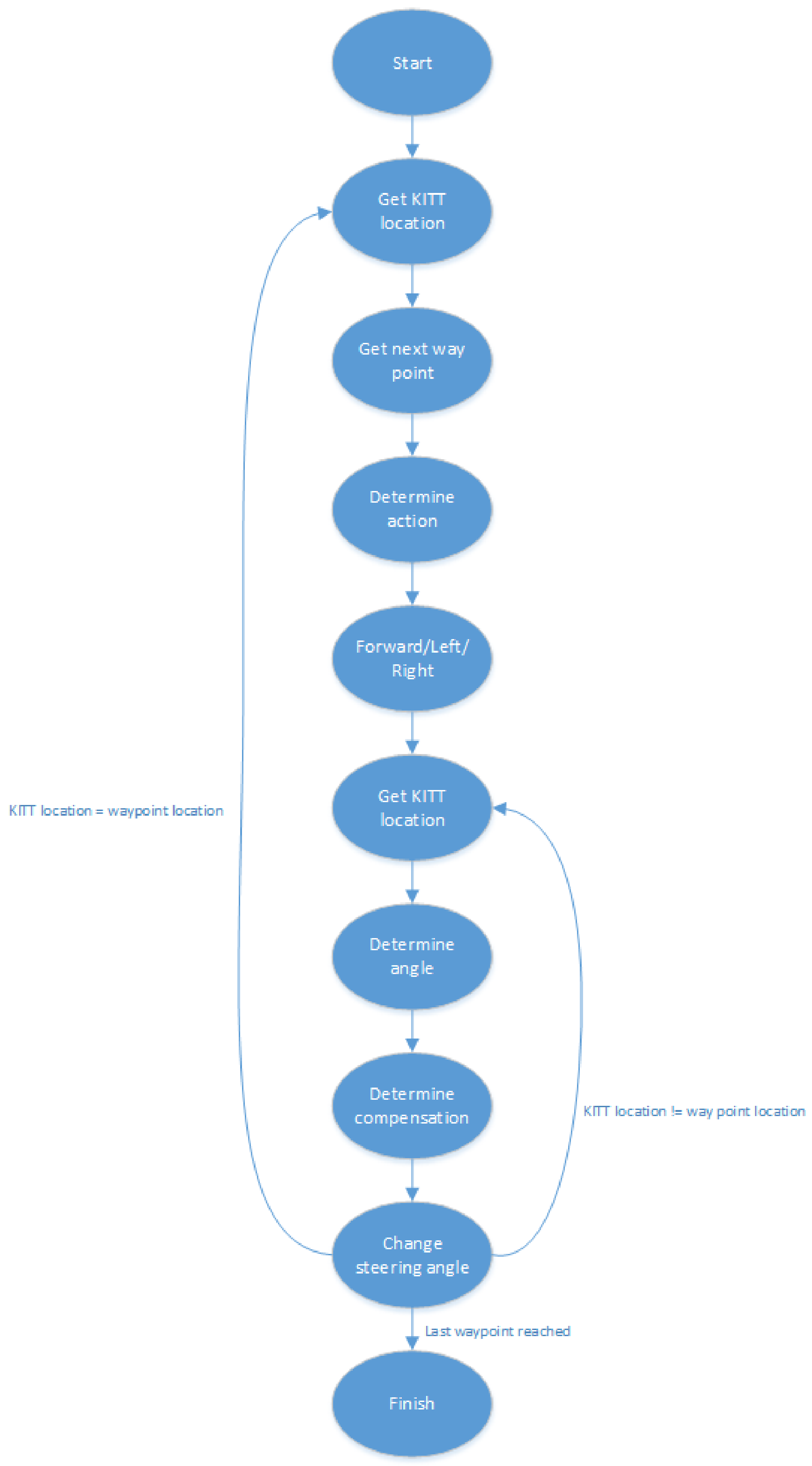

Figure 3.2: Finite state machine for autonomous driving

Making a turn

Making a turn (either left or right) is done by changing the orientation of the wheels and letting KITT drive for a certain amount of time, t_{turn}. t_{turn} has been determined by tests where KITT made turns until the good orientation and location were achieved. After t_{turn} sec the orientation of the wheels is changed back so KITT can drive straight again.

Steering compensation

As explained in chapter 3.2.1 it is important for KITT to keep driving in a straight line. A small deviation in the orientation of the wheels can cause KITT to deflect from the given path and cause problems. In order to let KITT drive in a more or less straight line (straight enough so the car stays in between the squares) the steering angle has to be modified when needed. By using two subsequent location points it is possible to determine the angle KITT is making with some reference line (in this case the reference is the north). Because it is not sure if the angle can be determined precise enough based on the localisation data and it is not desirable for KITT to be continuously changing the steering angle, the angle is mapped to a rough direction which KITT is then facing. See figure 3.3 for a more clear illustration of this. Based on the direction in which KITT is then heading the steering compensation factor is applied. The right compensation factor has to be determined by tests.

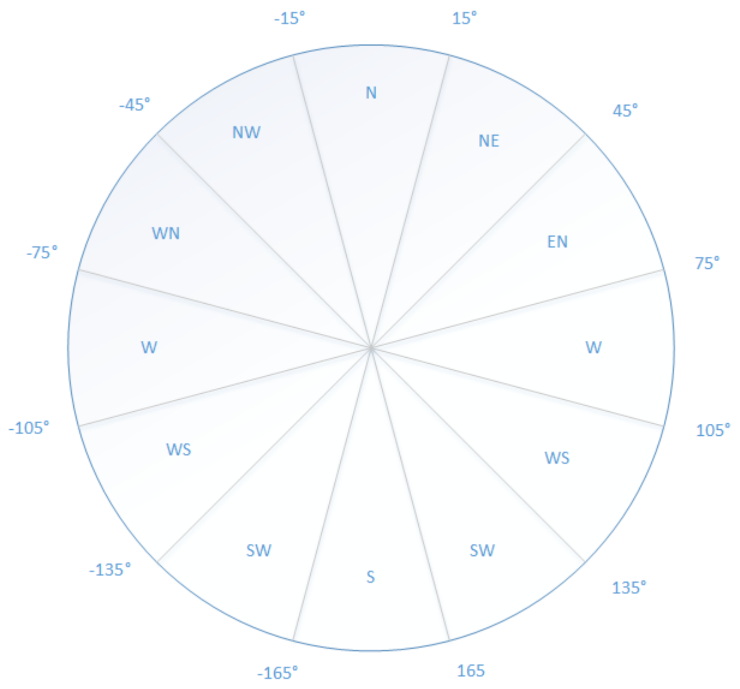

Figure 3.3: Angle mapping to direction of KITT for compensation

Result

At first the localisation of KITT was not consistent enough to be able to test the design. Therefore the design was simulated using example way points and by printing the actions that were called. Doing this helped in debugging the code but off course didn't show how well it performed in practice. The same was done for the steering compensation by simulating two consecutive locations to see if the right angle was calculated and the right compensation factor was applied.

Once the localisation was consistent, it was time to test the design. A lot of code had to be written before doing the first test. This made it very difficult to debug the code even when we knew separate modules (like mentioned above) worked correctly. It would have been easier if 'easy debug methods' would have been added throughout the code, but there was not enough time to implement this in a good way without making the code more complex. From the first tests it became clear that it would take a lot of time to make the code working, time that wasn't available. Therefore the choice was made to test the more primitive design, design 2, which gave much more promising results during the first few tests.

3.3 Design 2

Design concept

To reach a specified destination a function was created in Matlab to find the best route to take, see appendix A.2. The idea was that using the previous and current location the current direction, $\theta 1$, at which the car was driving could be found.Then using the current location and the destination the ideal direction, $\theta 2$, is found. Using the absolute value of $\theta 1 - \theta 2$ (as seen in figure 3.4), the angle which the car needs to turn to exactly face its destination is found. A quick check is preformed to see which of the two $\theta's$ is larger, this determines whether the car has to turn right or left. This function also calculates the distance between the current location and the destination using Pythagoras. The description of this function is as follows:
$[angle, direction, distance] = find\_route(current\_location, previous\_location, destination)$

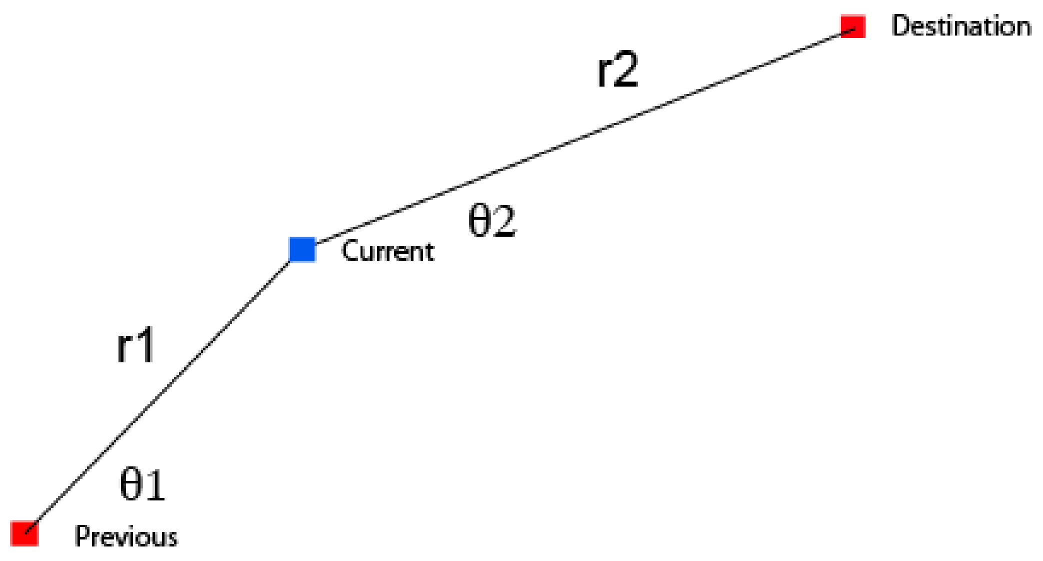

Figure 3.4: Planning a route to a destination

Using this, a new function is made to make use of this data and give commands to the car to drive to the destination. This function can be found in appendix A.2. This was done by splitting the possible angles into states and assigning commands to each state separately, depending on the size of the angle and even the distance from the destination. The exact details on what settings were used can be seen in the Matlab code and will not be discussed in great detail as they were optimized by trial and error. The general idea is that when the angle that the car needed to turn to was less than 15%, the car was allowed to drive straight. And when the angle increases, the car is set to drive in the given direction (right or left) for longer and longer until it corresponds to a change in an angle equal to the one required. The format of the function is shown below.

$change\_angle(angle, direction, destination)$

Results

To optimise the time it takes for the car to reach a destination a few settings were added to the change_angle function. If the destination was more than 2 meters away from the current position the timers were increased to make more drastic movements. For example if the angle was less than 15% and the car was more that 2 meters away, the car would move forward significantly more than if it was closer to the destination. Again, this value was tweaked until the results were satisfactory. One set of recorded results can be seen in figure 3.5.

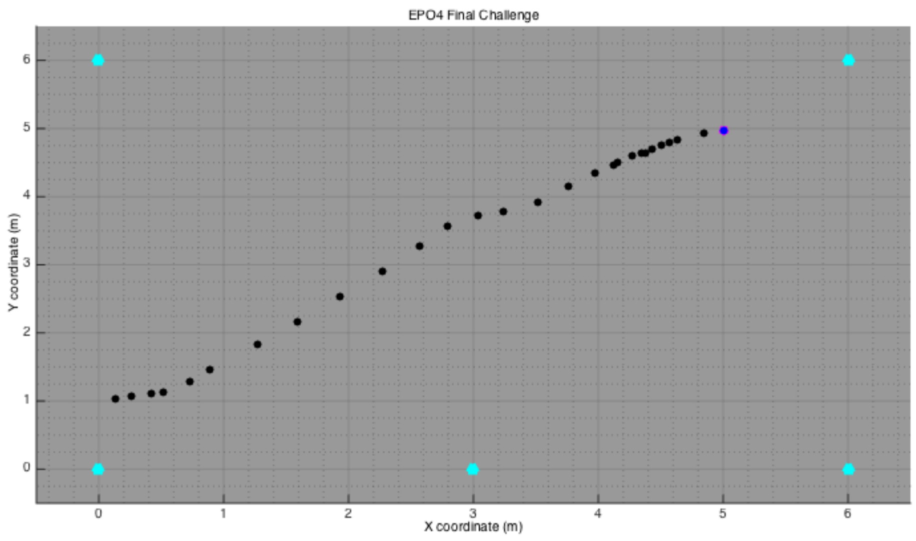

Figure 3.5: Recorded route to a destination

3.4 Grid path-finding

3.4.1 Specifications

The design discussed in section 3.2.1 uses a grid to navigate. To avoid obstacles and to reach destinations as fast as possible, the path-finding in the grid has to be optimized. To find the shortest path from starting node to destination node, a path-finding algorithm was implemented. The algorithm of choice was Dijkstra's shortest path algorithm.

3.4.2 Design plan and implementation

Dijkstra's shortest path algorithm finds the shortest path from the starting node to every other node in the graph. The algorithm of Dijkstra takes the following steps:

1. Assign a distance value of zero for the initial node and infinity for all other nodes. The distance for a node V represents the distance from V to the initial node.
2. keep a list of all unvisited nodes (Q) and the final nodes (S). Initially set all nodes in Q.
3. Find node V with the smallest distance in Q. Remove this node from Q and add it to S. For all connected nodes from V, update their distances. The distance for a connected node is the minimum of its current value for distance and the distance of V plus the weight between V and the connected node.
4. Repeat step 3 till Q is empty. When Q is empty, the algorithm is done and all distances have been determined from the initial node.

This algorithm was implemented in MATLAB, see appendix A.3. Once its initial node was located, the algorithm would generate the shortest path to drive. With the ultrasonic sensors, KITT would drive this path until an obstacle is detected. Once an obstacle is detected, the corresponding node V of the obstacle is determined. Then for all nodes connected to V, all weights are set to infinity to ensure that the obstacle will be avoided. After the weights have been set, a new path is requested by KITT. The path now does not contain the obstacle detected previously and KITT will continue driving the nodes.

3.4.3 Optimizations

A small change in the algorithm of Dijkstra was made to minimize the number of turns KITT has to take to reach its destination. This was done by adding turning penalties if the direction of node V differed from the direction of its connected nodes. Since the accuracy of KITT's steering can show big variances, this was a desirable addition to the path-finding.

3.4.4 Design results and verification

The algorithm was tested on a four by four grid. The positioning of the nodes can be found in figure 3.6.

A path was requested from node 11 to 44 given the direction of KITT was faced north. The result can be found in listing 3.1

Listing 3.1: MATLAB output

```
>> grid.dijkstra(11,44,'n')

ans =

    11     21     31     41     42     43     44
```

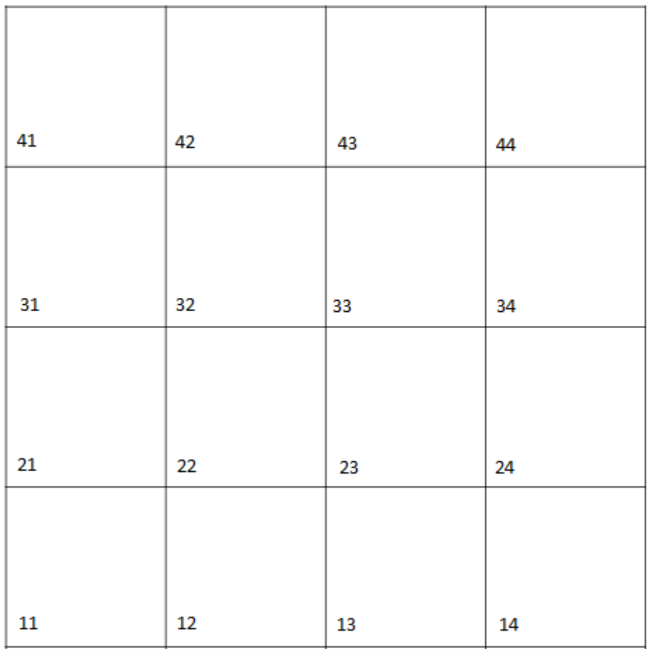

Figure 3.6: Mapping of nodes on a 4x4 grid

As can be seen from listing 3.1, a simple path was created to reach the destination. The result for adding obstacles on the field is shown in listing 3.2

Listing 3.2: MATLAB output

```
>> obstacle_weight = 1e4;
>> grid.set_weight(obstacle_weight, 31, 41);
>> grid.set_weight(obstacle_weight, 23, 24);
>> grid.dijkstra(11,44,'n')

ans =

    11     21     31     32     33     34     44
```

An obstacle was set between nodes 31 and 41 and node 23 and 24. As can be seen from the output listed in listing 3.2, the path returned avoided the obstacle and kept the number of required turns to a minimum.

3.4.5 Discussion

The path returned by the algorithm is the shortest path to reach the node. Although the algorithm provides a solution for our obstacle detection challenge, steering these nodes requires fixed angles to drive and little to no deviation from its path. When testing the steering implementation we quickly realized that it requires a lot more tweaking than we had initially expected. Due to the limited testing time it was then decided to implement the design discussed in section 3.3. This design does not benefit from the addition of a grid so there is no path-finding required.

Chapter 4

Discussion

4.1 Discussion

4.1.1 Group process

The group worked well together. The subsystems were distributed evenly in our team and each team member had a target to work on at any given time. All team members were aware of the progress of other team members. The process of every task of every team member was discussed multiple times per week and formal records were made at the end of every project session to keep track of the current progress. Since the size of the group is relatively small, it was easy to communicate within the group. The communication was good at every moment during project hours, via mail and via our mobile phones. All this resulted in good teamwork. Mistakes made by other team members were found quickly and were always discussed to learn from.

4.1.2 Work division

The subsystems that were part of the final system for the challenges were the positioning system and the autonomous driving controller. These two systems were split into two groups. Each team member had its contribution to both subsystems but the final responsibilities for the localisation were given to Ziar and Bas and the driving controller to Enzo and Tiamur.

4.1.3 Outcome

The team was overall satisfied with the process of the project. Most tasks were completed within time frame. The main obstacle that prevented better performance on the challenges was the limited amount of testing time. It took us two weeks to realize our initial design for the controller was unlikely to succeed. This was because most of our time in the testing field was spent trying to locate the car. A good amount of that time was also lost by fixing problems we could not prepare for, example of these are: blue-tooth connections not setting up, (re)programming the motherboard and assembling/disassembling KITT. Fortunately, most of these problems were solved towards the end of the project.

Overall there was a lot to learn from this project. Dealing with practical limitations of the system was a big part of that. All team members were pleased with the experience they've gotten from the project. The results at the final challenge were pleasing and all team members were satisfied with their contribution to the project.

Chapter 5

Conclusion

5.1 Conclusion

The objective of module 3 was to locate the car at any position in the field as fast and accurate as possible. It began with implementing the code to determine the TDOA's from the microphones. The final version of the peak detection was pretty reliable although when constantly plotting the current location in a grid once in a while a point would end completely off the field. This could be because of not-ideal equipment or because of the algorithm used for localisation. After this was done, the algorithm for the localisation had to be implemented. The tests with the simulated MATLAB data went well and it seemed to work. But when testing it with the real data it didn't always give the right location. After spending a lot of time finding the cause, it was found that the fault was in the channel estimation: not all the noise was filtered from the recordings. With the localisation working, the car now has to be programmed to drive autonomously. There were two ideas to do this. The first idea was to make a grid out of the field and calculate the shortest path to the final destination via a shortest-path algorithm. It was thought that this would be the better solution because it didn't rely on a very accurate and fast localisation. Also it would be easier to avoid obstacles using this idea. However because of time constraints this idea couldn't be realised correctly. The other idea was to work with vectors and calculating directions for the car to reach the final destination. This idea was easier to realise within the time available. The final challenge went as expected: the first two assignment could be completed within the given accuracy and pretty fast too. Although the third and fourth challenge were not completed, this was expected because there was too little time to test these assignments.

Appendices

Appendix A

Matlab code

A.1 Localisation

Listing A.1: Matlab function to find channel response

```
function [h] = ch3( x, y)
%UNTITLED6 Summary of this function goes here
%   Detailed explanation goes here

Ny = length(y);
Nx = length(x);
L = Ny - Nx + 1;
Y = fft(y,Ny);
X = fft(x,Ny); % zero padding to length Ny
H = Y ./ X; % frequency domain deconvolution
eps = 0.01*max(abs(X));
% Remove rounding errors
ii = find(abs(X) < eps);
for i = 1:length(ii),
    H(ii(i)) = 0;
end;

h = ifft(H); % convert to time domain
%h = h(1:L); % truncate
h = abs(h); % get absolute value

end
```

Listing A.2: Matlab function to find position

```
% function to locate car given N, TDOA, mic_loc

% param N            :   number of microphones
% param TDOA         :   TDOA matrix for all microphones
% param mic_loc      :   matrix of microphone coordinates
%                        row 1 contains x-coordinates
%                        row 2 contains y-coordinates

function x = locate(N, TDOA, mic_loc)
    speed_sound = 340;

    range_dif = 100*speed_sound*TDOA;

    % temp matrix used to form LHS matrix
    differences = [];
    % temp matrix used to form LHS matrix
    mapping = [];
    % RHS vector
    b = [];

    p = 0;
    for i = 1:N
        for j = 1:N
            if (i < j)
                % mapping variables
                p = p + 1;
                q = j - 1;
                differences(end+1,:) = transpose(mic_loc(:,j) - mic_loc(:,i));
                mapping(p,q) = range_dif(i,j);
                b(end+1) = range_dif(i,j)^2 - (norm(mic_loc(:,i)))^2 + (norm(mic_loc(:,j)))^2;
            end
        end
    end

    % flip RHS to column vector
    b = transpose(b);
    % LHS matrix
    A = [2*differences -2*mapping];
    values = A \ b;
    x = values(1:2);
end
```

Listing A.3: Matlab function to determine current location

```
function [location] = current_location()
load(['data/' 'x_trunc'])       %Load ref signal data

%Options
P = 5800;   % periode
N = 5;      %# of mics
mic_loc = [0 600 600 0 300;0 0 600 600 0]; %mic locations
Fs = 48000;

%Plots
plot_ref_signal = 0;
plot_truncated = 0;
plot_channel = 0;
plot_pulse_train =0;
plot_cross_correlation=0;
plot_peaks =0;

%%%%%%%%%%%%%%%%%%%%%%%%%%% Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Record data from 1-N speakers for current location
    [recorded_signal] = audio_measure(Fs,N);


%Estimate Channel
for i=1:5
    h(i,:) = ch3(x_trunc,recorded_signal(i,:));
end

%Find the peaks
[~,index] = max(h,[],2);                %find peak in xcorr
dif = mod((length(h)-1)/2 - index,P);   %mod period
for i = 1:N
    tmp = dif(i):P:length(h);
    peaks(i,1:length(tmp)) = tmp;
end

%Time differences
%compare peaks
```

```
for i=1:N
    for j=1:N
        if(j>i)
            tdoa(i,j)=(mod(index(i),P)-mod(index(j),P))/Fs;
        end
    end
end

%find locations
location = locate(N, tdoa, mic_loc);

% EPO4figure.setMicLoc(mic_loc'/100)
% EPO4figure.setKITT(location/100)

%%%%%%%%%%%%%%%%%%%%%%%%%%% Plots %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%plot channels
if (plot_channel == 1)
%     figure
    for i =1:5
        subplot(5,1,i)
        plot(h(i,:))
    end
end

%ref signal
if (plot_ref_signal == 1)
    figure
    for i =1:5
        subplot(5,1,i)
        plot(recorded_signal(i,:))
    end
end

%truncated data
if (plot_truncated == 1)
    figure
    for i =1:5
        subplot(5,1,i)
        plot(x_trunc(i,:))
    end
end

%plot pulse train
if (plot_pulse_train == 1)
    figure
    plot(pulse)
end

%plot cross correlation
if (plot_cross_correlation==1)
        figure
    for i =1:5
        subplot(5,1,i)
        plot(t(i,:))
    end
end


%plot h + peaks
if plot_peaks ==1
    figure
    for i=1:5
        x_as = peaks(i,:);
        x_as(x_as==0)=[];
        subplot(5,1,i)
        plot(h(i,:))
        hold on
        plot(x_as,max(h(i,:))/2,'r*')
    end
end

end
```

A.2 Autonomous driving

Listing A.4: Matlab function to find heading direction

```
function [angle,direction,distance] = find_route(current_location,previous_location,destination)
% locations are in the form [x;y]
%
%                   Destination
%
%
%
%       Current
%
%       Previous
angle_cp = atan2(current_location(2)-previous_location(2),current_location(1)-previous_location(1))*180/pi

angle_cd = atan2(destination(2)-current_location(2),destination(1)-current_location(1))*180/pi;

angle = abs(angle_cp - angle_cd);

distance = sqrt((destination(1)-current_location(1))^2 + (destination(2)-current_location(2))^2);

if angle_cp > angle_cd
    direction = 'right';
elseif angle_cp < angle_cd
        direction = 'left';
else
    direction = 'straight';
end

end
```

Listing A.5: Matlab function to steer

```
function state_log = change_angle(angle,direction,distance)
%UNTITLED5 Summary of this function goes here
%   Detailed explanation goes here
speed = 158;
straight = 150;
stop = 150;
right = 100;
left = 200;

if distance > 100
    time1 = 0.5;
    time2 = 0.1;
    time3 = 0.1;
    time4 = 0.3;
else
    time1 = 0;
    time2 = 0;
    time3 = 0;
    time4 = 0;
end

if angle <=15
    state =1;
% elseif angle <= 15
%     state = 2;
elseif angle <= 30
    state = 3;
else
    state = 4;
% elseif angle >150
%     state = 5;
%     disp('angle too large')
end
state_log = state;
%case 1: 0<angle30
switch(state)
    case 1
        drive(straight,speed)
        disp('state 1')
        pause(time1)
    case 2
         if strcmp(direction,'right')
            disp('state 2,right')
            drive(right,speed)
            pause(time2)
        elseif strcmp(direction,'left')
            disp('state 2,left')
            drive(left,speed)
            pause(time2)
         end
     case 3
         if strcmp(direction,'right')
            disp('state 3,right')
            drive(right,speed)
            pause(time3)
        elseif strcmp(direction,'left')
            disp('state 3,left')
            drive(left,speed)
            pause(time3)
         end
    case 4
         if strcmp(direction,'right')
            disp('state 4,right')
            drive(right,speed)
            pause(time4)
        elseif strcmp(direction,'left')
            disp('state 4,left')
            drive(left,speed)
            pause(time4)
        end
    case 5



end
```

Listing A.6: Matlab code for challenge 2

```
% Challenge 2
% Go from A to B with a in between stop via 'waypoint'

state_log       = [];
destination     = [500;497];
waypoint        = [193;400];
state           = 0;
distance        = 10000;
stop_distance   = 100;
pause_time      = 3;
max_voltage     = 17000;
regular_pause   = 0.5;
back_time       = 2;

while 1
    switch(state) %initialize
        case 0
            status_string = EPOCommunications('transmit', 'S');
            status_obj = status(status_string);
            while(status_obj.voltage < max_voltage)
                status_string = EPOCommunications('transmit', 'S');
                status_obj = status(status_string);
            end
            state = 1;
        case 1
            previous = current_location();
            drive(150,158)                                              %drive(direction,speed)
            pause(regular_pause)
            drive(150,150)
            current = current_location();
            state = 2;                                                    %change state)
            EPO4figure.setMicLoc(mic_locs/100)
            EPO4figure.setDestination(destination/100)
            EPO4figure.setKITT(current/100)
        case 2
            if distance > stop_distance
                pause(regular_pause)
                [angle,direction,distance] = find_route(current,previous,waypoint);
%if closer than 1m to object end
                state_temp = change_angle(angle,direction, distance);
                state_log = [state_log state_temp];
                previous = current;
                current = current_location();
                drive(150,150)                  %return wheels to straigth
                EPO4figure.setKITT(current/100)
            else
                pause(pause_time);
                drive(150, 145)
                pause(back_time);
                state = 3;
                distance = 10000;
            end
        case 3
            previous = current_location();
            drive(150,158)                                              %drive(direction,speed)
            pause(regular_pause)
            drive(150,150)
            current = current_location();
            state = 4;                                                    %change state)
            EPO4figure.setMicLoc(mic_locs/100)
            EPO4figure.setDestination(destination/100)
            EPO4figure.setKITT(current/100)
        case 4
            if distance > stop_distance
                pause(regular_pause)
                [angle,direction,distance] = find_route(current,previous,destination);
%if closer than 1m to object end
                state_temp = change_angle(angle,direction, distance);
                state_log = [state_log state_temp];
                previous = current;
                current = current_location();
                drive(150,150);                  %return wheels to straigth
                EPO4figure.setKITT(current/100)
            else
                break

            end
    end
end
```

Listing A.7: Matlab function to determine current location

```
function [location] = current_location()
load(['data/' 'x_trunc'])       %Load ref signal data
```

```
%Options
P = 5800;    % periode
N = 5;       %# of mics
mic_loc = [0 600 600 0 300;0 0 600 600 0]; %mic locations
Fs = 48000;

%Plots
plot_ref_signal = 0;
plot_truncated = 0;
plot_channel = 0;
plot_pulse_train =0;
plot_cross_correlation=0;
plot_peaks =0;

%%%%%%%%%%%%%%%%%%%%%%%%%%% Main %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Record data from 1—N speakers for current location
    [recorded_signal] = audio_measure(Fs,N);


%Estimate Channel
for i=1:5
    h(i,:) = ch3(x_trunc,recorded_signal(i,:));
end

%Find the peaks
[~,index] = max(h,[],2);                %find peak in xcorr
dif = mod((length(h)—1)/2 — index,P);   %mod period
for i = 1:N
    tmp = dif(i):P:length(h);
    peaks(i,1:length(tmp)) = tmp;
end

%Time differences
%compare peaks
for i=1:N
    for j=1:N
        if(j>i)
            tdoa(i,j)=(mod(index(i),P)—mod(index(j),P))/Fs;
        end
    end
end

%find locations
location = locate(N, tdoa, mic_loc);

% EPO4figure.setMicLoc(mic_loc'/100)
% EPO4figure.setKITT(location/100)

%%%%%%%%%%%%%%%%%%%%%%%%%%% Plots %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%plot channels
if (plot_channel == 1)
%     figure
    for i =1:5
        subplot(5,1,i)
        plot(h(i,:))
    end
end

%ref signal
if (plot_ref_signal == 1)
    figure
    for i =1:5
        subplot(5,1,i)
        plot(recorded_signal(i,:))
    end
```

```
end

%truncated data
if (plot_truncated == 1)
    figure
    for i =1:5
        subplot(5,1,i)
        plot(x_trunc(i,:))
    end
end

%plot pulse train
if (plot_pulse_train == 1)
    figure
    plot(pulse)
end

%plot cross correlation
if (plot_cross_correlation==1)
        figure
    for i =1:5
        subplot(5,1,i)
        plot(t(i,:))
    end
end


%plot h + peaks
if plot_peaks ==1
    figure
    for i=1:5
        x_as = peaks(i,:);
        x_as(x_as==0)=[];
        subplot(5,1,i)
        plot(h(i,:))
        hold on
        plot(x_as,max(h(i,:))/2,'r*')
    end
end

end
```

A.3 Dijkstra's algorithm

Listing A.8: Matlab class to define a node

```
classdef node < handle
    properties
        id;
        x;
        y;
        connected_nodes;
        connected_nodes_weights;
        distance;
        shortest_path;
        direction;
        is_still_in_set;
    end

    methods
        function node_obj = node(node_id)
            node_obj.id = node_id;
        end

        function bool = equals(node1, node2)
            bool = (node1.id == node2.id);
        end
    end
end
```

Listing A.9: Matlab class to define a grid

```
classdef grid < handle
    properties
        nodes;
        w_x;
        w_y;
        w_d;
        id_matrix;
        GRID_SIZE_X;
        GRID_SIZE_Y;
    end

    methods
        function grid_obj = grid(GRID_SIZE_X, GRID_SIZE_Y, ROOM_SIZE_X, ROOM_SIZE_Y)
            grid_obj.GRID_SIZE_X = GRID_SIZE_X;
            grid_obj.GRID_SIZE_Y = GRID_SIZE_Y;

            % distances between nodes
            w_x = ROOM_SIZE_X / GRID_SIZE_X;
            w_y = ROOM_SIZE_Y / GRID_SIZE_Y;

            grid_obj.w_x = w_x;
            grid_obj.w_y = w_y;
            grid_obj.w_d = sqrt(w_x^2 + w_y^2);

            grid_nodes = [];

            % generate nodes
            for i = GRID_SIZE_Y:-1:1
                for j = 1:GRID_SIZE_X
                    s1 = num2str(i);
                    s2 = num2str(j);
                    id = strcat(s1, s2);
                    id = str2double(id);
                    node_obj = node(id);
                    node_obj.x = ROOM_SIZE_X / GRID_SIZE_X * (1/2 + j-1);
                    node_obj.y = ROOM_SIZE_Y / GRID_SIZE_Y * (1/2 + i-1);
                    grid_nodes = [grid_nodes node_obj];
                end
            end

            % reshape to a matrix
            grid_nodes = reshape(grid_nodes, GRID_SIZE_X, GRID_SIZE_Y);
            grid_nodes = transpose(grid_nodes);

            % connect the grid
            for i = GRID_SIZE_Y:-1:1
                for j = 1:GRID_SIZE_X
                    switch(i)
                        case GRID_SIZE_Y
                            switch(j)
                                case 1
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i-1,j) grid_nodes(i,j+1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x];
                                case GRID_SIZE_X
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i,j-1) grid_nodes(i-1,j)
                                    grid_nodes(i,j).connected_nodes_weights = [w_x w_y];
                                otherwise
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i,j-1) grid_nodes(i-1,j)
                                    grid_nodes(i,j).connected_nodes_weights = [w_x w_y w_x];
                            end
                        case 1
                            switch(j)
                                case 1
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i,j+1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x];
                                case GRID_SIZE_X
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i,j-1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x];
                                otherwise
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i,j-1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x w_x];
                            end
                        otherwise
                            switch(j)
                                case 1
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i-1,j)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_y w_x];
                                case GRID_SIZE_X
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i,j-1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x w_y];
                                otherwise
                                    grid_nodes(i,j).connected_nodes = [grid_nodes(i+1,j) grid_nodes(i,j-1)
                                    grid_nodes(i,j).connected_nodes_weights = [w_y w_x w_y w_x];
                            end
                    end
                end
            end

            grid_obj.id_matrix = zeros(GRID_SIZE_Y, GRID_SIZE_X);
            for i = 1:GRID_SIZE_Y
                for j = 1:GRID_SIZE_X
                    grid_obj.id_matrix(i,j) = grid_nodes(i,j).id;
                end
            end
            grid_obj.nodes = grid_nodes;
        end

        % returns the node with id node_id
        function node_obj = find_node(grid_obj, node_id)
            for i = 1:grid_obj.GRID_SIZE_Y
                for j = 1:grid_obj.GRID_SIZE_X
```

```
                if (grid_obj.id_matrix(i,j) == node_id)
                    node_obj = grid_obj.nodes(i,j);
                    return
                end
            end
        end
        node_obj = [];
    end

    % returns the matrix indices of node_id
    function [i, j] = find_indices(grid_obj, node_id)
        for i = 1:grid_obj.GRID_SIZE_Y
            for j = 1:grid_obj.GRID_SIZE_X
                if (grid_obj.id_matrix(i,j) == node_id)
                    return
                end
            end
        end
    end

    % sets the weight of the edge between node id1 and node id2
    function set_weight(grid_obj, weight, id1, id2)
        node1 = grid_obj.find_node(id1);
        node2 = grid_obj.find_node(id2);

        empty = false;
        if (isempty(node1))
            fprintf('node %d is not found.\n', id1);
            empty = true;
        end

        if (isempty(node2))
            fprintf('node %d is not found.\n', id2);
            empty = true;
        end

        if (~empty)
            [x1, y1] = grid_obj.find_indices(id1);
            [x2, y2] = grid_obj.find_indices(id2);
            connected = false;
            for i = 1:length(node1.connected_nodes)
                if (node1.connected_nodes(i).id == id2)
                    grid_obj.nodes(x1,y1).connected_nodes_weights(i) = weight;
                    connected = true;
                end
            end

            for i = 1:length(node2.connected_nodes)
                if (node2.connected_nodes(i).id == id1)
                    grid_obj.nodes(x2,y2).connected_nodes_weights(i) = weight;
                    connected = true;
                end
            end

            if (~connected)
                fprintf('node %d and %d are not connected.\n', id1, id2);
            end
        end
    end

    function node_obj = find_minimum_distance_node(grid_obj)
        min_distance = inf;
        for i = 1:grid_obj.GRID_SIZE_Y
            for j = 1:grid_obj.GRID_SIZE_X
                if (grid_obj.nodes(i,j).distance < min_distance && grid_obj.nodes(i,j).is_still_in_set
                    node_obj = grid_obj.nodes(i,j);
                    min_distance = grid_obj.nodes(i,j).distance;
                end
            end
        end
    end

    function path = dijkstra(grid_obj, id1, id2, init_dir)
        turning_penalty = 50;
        obstacle_weight = 1e4;
        inaccessible = [];
        inaccessible_weights = [];

        switch(init_dir)
            case 'n'
                direction = [0 1];
            case 'e'
                direction = [1 0];
            case 's'
                direction = [0 -1];
            case 'w'
                direction = [-1 0];
            otherwise
                direction = [0 1];
        end

        for i = 1:grid_obj.GRID_SIZE_Y
            for j = 1:grid_obj.GRID_SIZE_X
                grid_obj.nodes(i,j).distance = inf;
                grid_obj.nodes(i,j).is_still_in_set = 1;
                grid_obj.nodes(i,j).shortest_path = 0;
            end
        end

        origin = grid_obj.find_node(id1);
        destination = grid_obj.find_node(id2);

        origin.distance = 0;
        origin.direction = direction;

        for i = 1:length(origin.connected_nodes)
            current_node = origin.connected_nodes(i);
            x = current_node.x - origin.x;
            y = current_node.y - origin.y;
            rv = [x y];
            d = dot(rv, direction);
            prod = length(rv) * length(dir);
            tetha = acosd(d / prod);
            if (tetha >= 90)
                inaccessible = [inaccessible current_node];
                inaccessible_weights = [inaccessible_weights origin.connected_nodes_weights(i)];
                grid_obj.set_weight(obstacle_weight, id1, current_node.id);
            end
        end

        while(1)
            node = grid_obj.find_minimum_distance_node();
            node.is_still_in_set = 0;

            if (node.equals(destination))
                break
            end

            for i = 1:length(node.connected_nodes);
                current_node = node.connected_nodes(i);
                current_weight = node.connected_nodes_weights(i);
                x = current_node.x - node.x;
                y = current_node.y - node.y;
                rv = [x y];
```

```
                    current_node_dir = rv / norm(rv);

                    % flag to check directions of connected nodes
                    if (sum(node.direction == current_node_dir) == 2)
                        dir_equal = true;
                    else
                        dir_equal = false;
                    end

                    if (dir_equal)
                        total_weight = node.distance + current_weight;
                    else
                        total_weight = node.distance + current_weight + turning_penalty;
                    end

                    if (total_weight < current_node.distance && current_node.is_still_in_set)
                        current_node.distance = total_weight;
                        current_node.shortest_path = node;
                        current_node.direction = current_node_dir;
                    end
                end
            end

            % construct shortest path
            path = [];
            while (node.shortest_path ~= 0)
                path = [node.id path];
                node = node.shortest_path;
            end
            path = [id1, path];

            for i = length(inaccessible)
                grid_obj.set_weight(inaccessible_weights(i), id1, inaccessible(i).id);
            end
        end
    end
end
```

Listing A.10: Matlab code to run dijkstra's algorithm

```
close all
clear all
clc

% dimensions of the room in cm
ROOM_SIZE_X = 600;
ROOM_SIZE_Y = 600;

% max turning radius in cm
max_radius = 67.5;

% desired distance between nodes in cm
dx = max_radius;
dy = max_radius;

% number of nodes in x and y directions
GRID_SIZE_X = floor(ROOM_SIZE_X / dx);
GRID_SIZE_Y = floor(ROOM_SIZE_Y / dy);

% real distances between nodes
w_x = ROOM_SIZE_X / GRID_SIZE_X;
w_y = ROOM_SIZE_Y / GRID_SIZE_Y;
w_d = sqrt(w_x^2 + w_y^2);

% generate grid
grid = grid(GRID_SIZE_X, GRID_SIZE_Y, ROOM_SIZE_X, ROOM_SIZE_Y);
```

Bibliography

[1] EE2L21 Project EPO-4: Autonomous driving challenge, Alle-Jan van der Veen. https://blackboard.tudelft.nl/bbcswebdav/pid-2422827-dt-content-rid-8391321_2/courses/34309-141504/epo4manual_24apr.pdf 8