EE2L21 EPO-4: Kitt - Final Report

June 18, 2015

Group B6

Martijn Berkers	4223438
Thomas Brasser	4139518
Richard de Jong	4238575
Jeroen van Uffelen	4232690

Delft University of Technology

Abstract

In this report an approach to handle the final challenge given in the EPO-4 project is presented. This includes localization of the car with audio communication, collision detection is done using the data obtained from distance sensors, planning a route with the car and driving it. There are 3 parts of the final challenge, driving from point A to point B, driving from point A to point B with waypoint C, and driving from point A to point B with an obstacle in the route.

Introduction

This report describes the design process of an autonomous driving, wireless charging vehicle. The goal of the design is to make the car drive from point A to point B while avoiding obstacles. To achieve this several problems have to be overcome; wireless communication (Bluetooth connection), object sensing, mapping the location and let the car navigate, devising a control system to steer and drive the car, using audio communication for localization of the car and combining these solutions into a system.

To keep structure the report is divided into several sections. First, the specifications of the car will be determined.
The second section explores localization using audio communication, for this training sequences are discussed. The deconvolution of the measured signal and the reference signal and corresponding peak detection to calculate the Time-difference of arrival(TDOA) between microphones. The localization algorithm used to convert these TDOA's to a position, and the verification of this position.
The third section will discuss the route determination of the car, and the corresponding outputs to the wheels of the car.
The fourth section will discuss the system integration of all these parts and the loop where they are used in, and the GUI used for displaying the position of the car and obstacles.
Lastly a conclusion is constructed about the project and discussion about what could have been done different.

Specifications

The completed system consisting of the car and the software should be able to fulfill predefined tasks and is limited by some constraints. The EPO4 Manual[1] defines the final challenge as a system integration challenge featuring a demonstration to show the various capabilities of the system.

This demonstration consists of 4 succeeding challenges: In the first challenge the car should be able to drive from point 'A' to point 'B', the second challenge adds a waypoint 'C'. The third challenge adds an obstacle to avoid and the last challenge includes a source of interference.

Further constraints are the fully autonomous functioning of the system. Which means that neither the car or the controlling computer can be touched during the challenges. Also, at the start of the challenge, the supercaps should be charged wirelessly, and the challenge ends successfully when the car stops at its destination and the controlling computer plays a sound.

Contents

1 Localization using audio communication

The goal of this module is using audio beacons to accurately determine the position of the car. An audio beacon is placed on the car, the signal is received with five microphones at pre-determined positions. With this information the position of the car can be determined with the TDOA(Time difference of arrival). This time corresponds to the difference in arrival time between microphones. With microphone pairs the position can be determined. To determine when the "sound" is received a training sequence is used and transmitted using the audio beacon. This training sequence is received at all microphones at different times and at intensities, to filter the training sequence from the noise the channel response is used instead of the measured signal. For the estimation of the channel response a known training sequence has to be used. The deconvolution of the inverse of the training sequence and the measured signal gives channel response. There are several implementations for this because the inverse of the training sequence is not easily calculated. The implementation used is the frequency equalization The channel response will ideally have one peak at the point where the measured signal is the same as the training sequence. By comparing the position of the peaks of different microphones the difference of arrival in samples can be found; this is converted to time. In practice the measured signal is not clean, noise and reflections are also measured. To chose the right peak the bad ones need to be filtered. To improve the deconvolution properties of the measured signal, instead of the modelled training sequence the impulse response of the sequence is measured, this can be measured when the beacon is close to the microphone.

1.1 Training sequence

The training sequence that will be transmitted by the audio beacon is a binary sequence which uses "on-off keying". Besides a code word with a maximum of 64 bits, the training sequence is restricted by the parameters in table 1. The code word used is a random sequence of 32 bits, which is "1432AB5C" in HEX. The code is random because it has good convolution properties. The setup used has a sample frequency(Fs) of 48 kHz. That means that the maximum carrier frequency can be $\frac{1}{2} * Fs = 24\,\text{kHz}$. To be on the safe side our Timer0 parameter is 2, which is 15 kHz.
To distinguish the carrier frequency from the code frequency, at least 5 waves are wanted in 1 bit. Because of this, the Timer1 parameter is 4, which corresponds to 3.0 kHz. The Timer3 parameter is 7, which is 8 Hz. This is because there has to be a silent period after the code has been send, to let reflections and the beacon die out. To clarify the parameters, in figure 1 an example of a generated beacon sequence can be found.

Timer index			0	1	2	3	4	5	6	7	8	9
Carrier Freq	(Timer0)	[kHz]	5	10	15	20	25	30	0	0	0	0
Code Freq	(Timer1)	[kHz]	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	0
Repeat Freq	(Timer3)	[Hz]	1	2	3	4	5	6	7	8	9	0

Table 1: Timers frequency configuration table

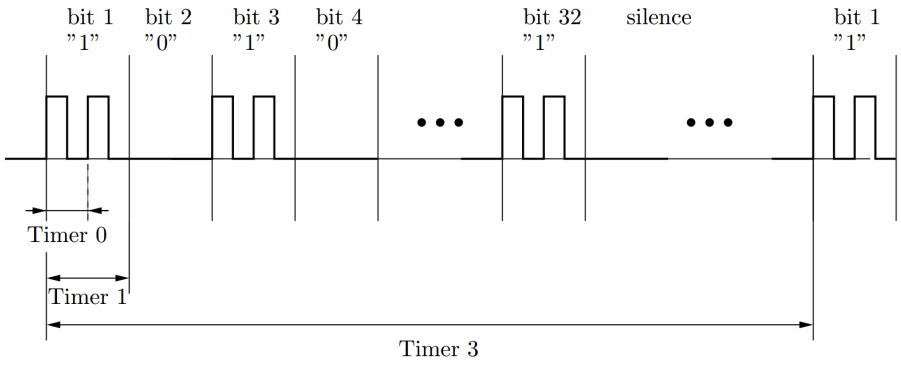

Figure 1: Example of pulses generated by the beacon

1.2 Reference

To have the best deconvolution properties an accurate reference signal has to be used. The modelled signal and the transmitted signal are not equal, because of this the deconvolution properties are not optimal. To get an accurate reference the signal that leaves the speaker needs to be measured. At close distances of around 5 cm, the measured signal will be the channel response $y[n] = h[n]$ where $y[n]$ is the measured signal in samples and $h[n]$ the channel response. Each time the training sequence is changed a new reference has to be measured. The reference used was one of the multichannel measurements, because these microphones are the same as the ones used in the final challenge. The signal was manually shortened to only contain the training sequence, the length of this is given by the settings of the training sequence, the settings used give it a length of 512 samples with a sample rate of $48kHz$. Because it is not clear when the signal begins and ends some data points before the sequence and after have been included in the reference. A plot of the reference can be seen in figure 2

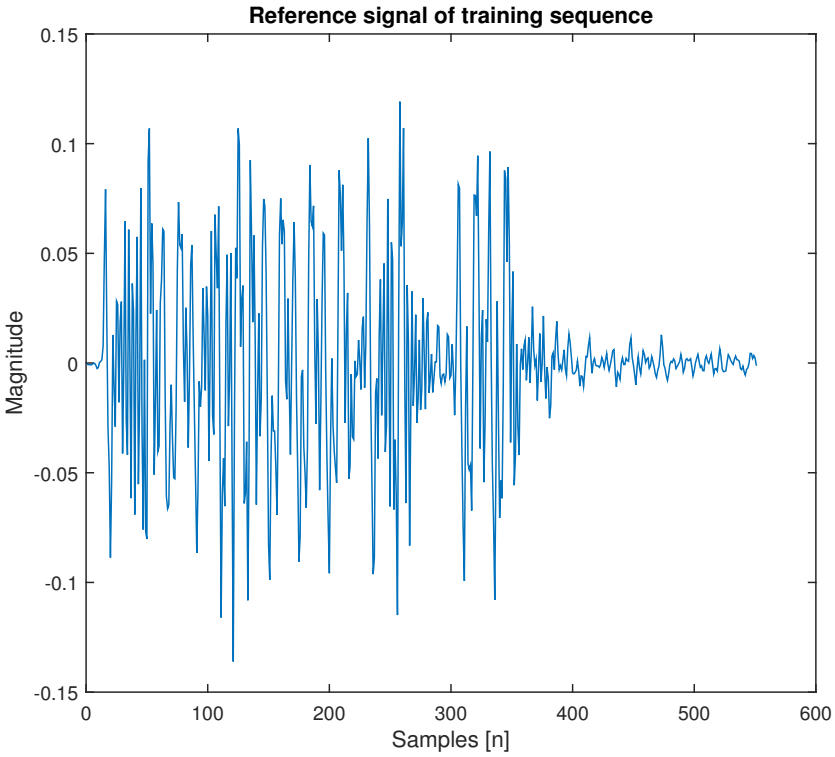

Figure 2: Refsignal

1.3 Deconvolution

The deconvolution of the received audio signal by taking the inverse of the reference signal is computationally complicated. There are three alternative methods which can be used for the deconvolution; the first is the calculation of the Toeplitz matrix, the second is using a matched filter and the third is using frequency equalization. The Toeplitz matrix has a size of the length of the measured signal times the length of the impulse response. The impulse response will be length of y minus the reference. Both of these for our usual measurements are around 18000. Taking the inverse of a matrix this size is not a viable option. Because there was still some doubt about which was better, the matched filter or the frequency equalization, it was decided to implement both. As the tests were done the method with the most reliable results will be the one to choose for the final challenge. The frequency equalization turned out to be the best option and is chosen for the final challenge

The deconvolution function has two inputs, the measured audio signal $y[n]$ and the reference measured training sequence $x[n]$. function will return two estimated channel responses $h[n]$, one for the matched filter and one for the frequency equalization. The matched filter will use the `filter` function in MATLAB, the filter coefficients are determined by the flipped training sequence $x[-n]$. Because this filtered data is not yet yet in the right scale, it is corrected with the magnitude of $x[n]$. The Matched Filter can be written as

$$\hat{h}[n] := y[n] * x[-n] = h[n] * (x[n] * x[-n]) = h[n] * r[n]$$

where $r[n]$ is the autocorrelation of the reference signal. The audio channel response will be smeared, because of the autocorrelation it is expected that there will always be a peak where the signals overlap. Because of this the channel response may be inaccurate but at least the peak will be pronounced and corresponds to the true channel. A plot of the typical impulse response estimated with a matched filter is found in 4. The code of the matched filter can be seen in appendix **??**

An alternative to the Matched Filter is to do frequency equalization. Because the frequency equalization had the best deconvolution properties this is used for the final challenge. $Y(\omega) = H(\omega)X(\omega)$ is derived from $y[n] = h[n] * x[n]$ and hence we can estimate $H(\omega) = Y(\omega)/X(\omega)$. To make this work using the FFT all the sequences need to be of equal length N, where N is the length of y. Since a pointwise division is done in frequency domain the performance will be low for frequencies where $X[k]$ is small. Therefore a threshold ϵ is used to determine which values of $H[k]$ will be set to zero before the Inverse Fast Fourier Transform is applied. The value of epsilon used is calculated in line 10 in Listing 1 after that the indices where X is smaller than ϵ are determined in lines 10-16 and the newly obtained channel estimate is calculated. The entire code can be found in appendix **??**

Listing 1: Calculating Epsilon

```
10 eps = 0.1*max(abs(X)); %Make epsilon depend on X
11 G = X; %make G the same length as X
12 ii = find(abs(X) <= eps);
13 G(ii) = 0;
14 ii = find(abs(X) > eps);
15 G(ii) = 1;
```

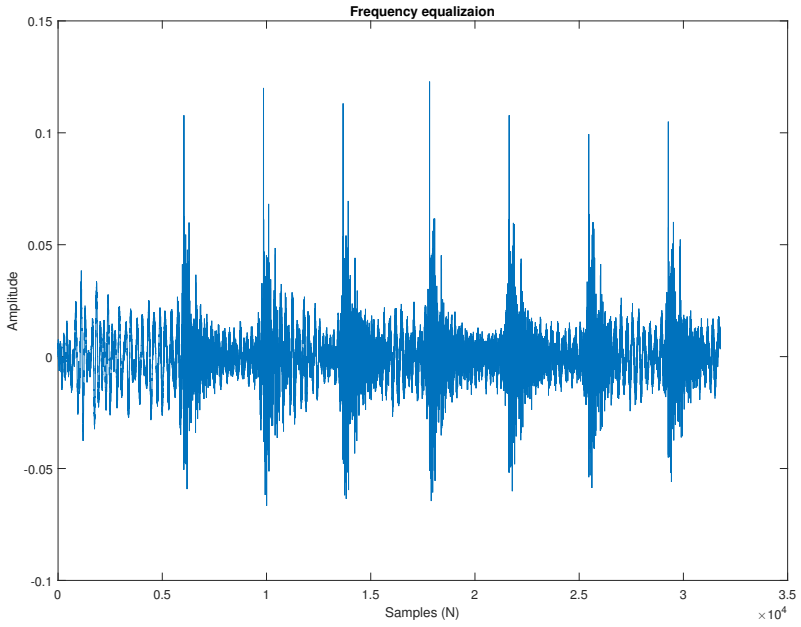

Figure 3: Typical impulse response of frequency equalization

```
16 H = H.*G;
```

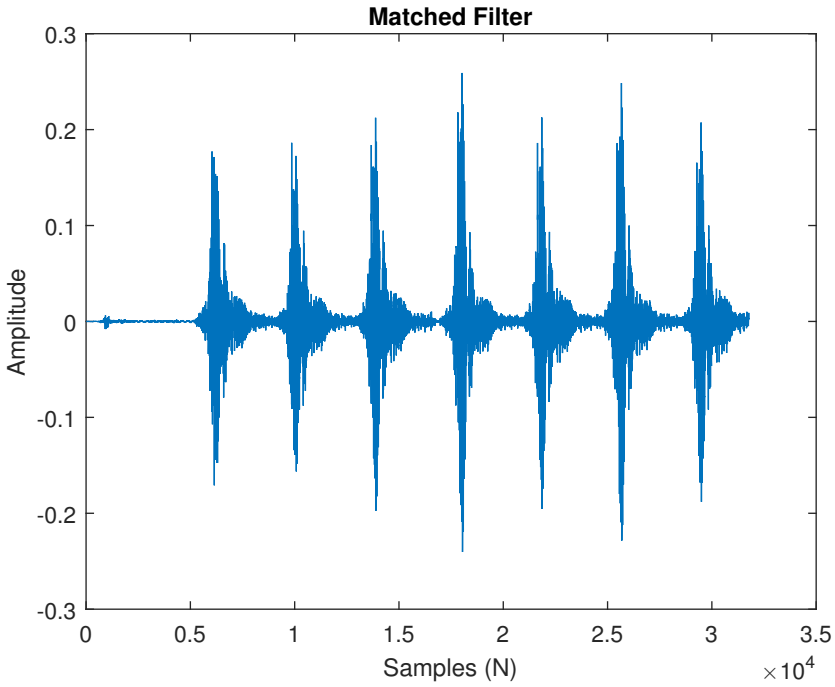

Figure 4: Typical impulse response of a matched filter

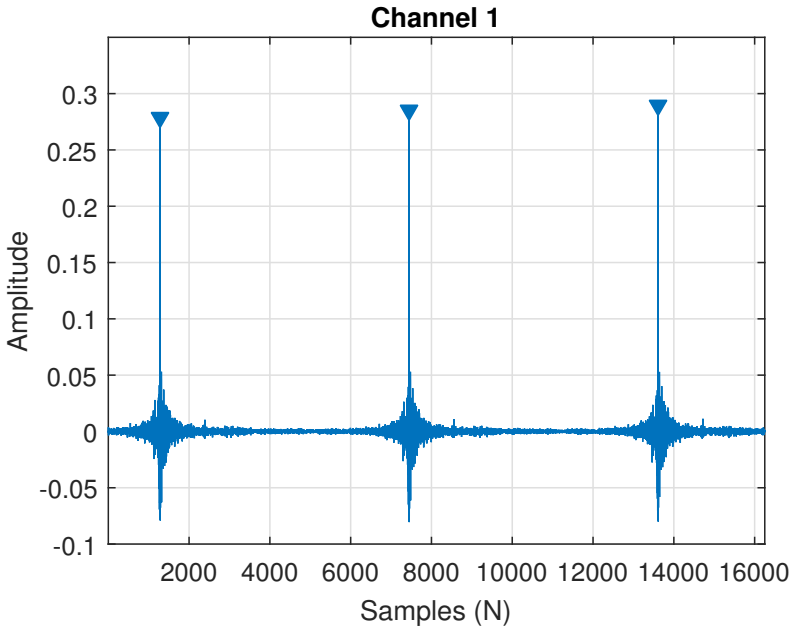

Figure 5: Peak detection channel 1

1.4 Peak detection

When the channels are estimated the position of the peaks of the estimated channels need to be determined. The peaks of the channels are determined using the `findpeaks` command in matlab. This function returns the height of the peaks and the location of the peaks based on the parameters given to the function as can be seen in Listing 2. The code used calculates the first peak of the signal after a 1200 samples silence period, which corresponds to the maximum delay that should appear in the signal, and a 6000 samples minimum peak distance range. The maximum delay found is across the diagonal of the field and therefore this delay in samples is calculated as $\sqrt{6^2 + 6^2}/340 * 48000 = 1.1979 * 10^3$ samples. Where $340m/s$ is the speed of sound and $48000Hz$ is the sample rate of the system. The minimum peak distance is based on the repetition timer of the beacon setting. An example of the calculated peaks of the first channel is shown in figure 5. The first peak found after the silence period of 1200 samples is used as reference for the other peaks. The entire code/function used to calculate the first peak is found in appendix **??**

Listing 2: Peak position

```
6 Ts = 1200; %Search window
7
8 %Determine the location of the largest peak after a silence period
9 [peaks_f, locations_f] = findpeaks(h1(Ts:end), 'MinPeakDistance',
      6000);
```

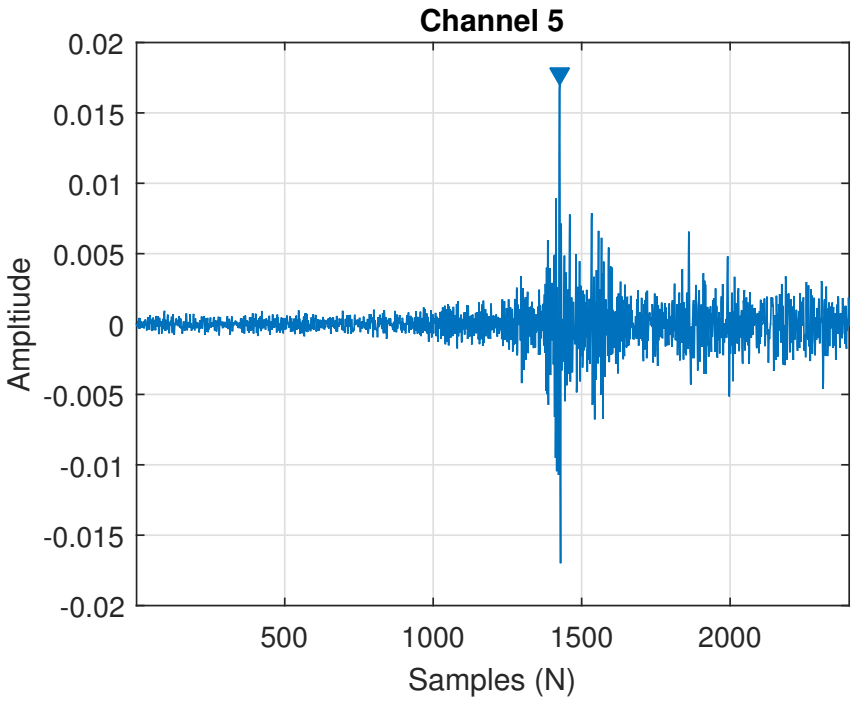

Figure 6: Peak detection channel 5

When the peak of the first channel is found this peak location is used as a reference for the other peak locations as can be seen in Listing 3 and is illustrated in figure 7. The position of peak 1 is the position of $nmax$ in the figure. A search window is applied to the second channel ranging from ± 1200 samples from the position of the peak of the first channel. The location of the highest peak in this interval is returned and corresponds to the delayed or leading peak from the first channel. The resulting window is shown in figure 6 for channel 5 of the test measurement. The search window has a size of ± 1200 samples and the corresponding peak position is found. This same process is done for the remaining channels and all the peak positions are returned so the TDOA can be calculated.

Listing 3: Peak position channel 2

```
15 loc_peak_1 = loc_peak1;
16
17 [peaks_2_f,locations_2_f] = findpeaks(h2([loc_peak_1-Ts:loc_peak_1+
       Ts]), 'MinpeakDistance', 2*Ts-1);
```

1.5 TDOA

After the peak detection all the matching peak positions are known for each microphone. With these peak positions the differences between microphones can be determined. For use in the localization algorithm they will be returned in a row shape in. To convert the samples to cm it is divided by sample rate, the sample rate used is 48 kHz. With this time in seconds, the distance in meters can be recovered with the speed of sound. An estimate of 340 $\mathrm{m\,s^{-1}}$ is used for the speed of sound, this distance in meters multiplied will be the difference distance between microphones in cm. For the final challenge five microphones are present, the TDOA matrix will have the size of 5x5. The portion that calculates the TDOA is seen in listing 4. In table 1.5 the measurement used as a example in the peak detection subsection are used to display the tdoa matrix, the distance differences in cm are displayed.

Listing 4: TDOA calculation with microphone positions

```
27 %Calculate all the differences in distances
28 TDOA = zeros(5);
29 for x = 1:5
30     for y = 1:5
31         TDOA(y,x) = position_mic(x) - position_mic(y);
32     end
33 end
34 %Calculate the TDOA in centimers
35 TDOA = TDOA ./ 48000;
36 TDOA = TDOA .* speed_of_sound*100;
37 %Make an array from the data for the function localize
38 TDOA_2 = [TDOA(2,1);TDOA(3,1);TDOA(4,1);TDOA(5,1);TDOA(3,2);TDOA
       (4,2);TDOA(5,2);TDOA(4,3);TDOA(5,3);TDOA(5,4)];
```

distance	mic 1	mic 2	mic 3	mic 4	mic 5
mic 1	0	388.1667	434.2083	187.0000	160.0833
mic 2	-388.1667	0	46.0417	-201.1667	-228.0833
mic 3	-434.2083	-46.0417	0	-247.2083	-274.1250
mic 4	-187.0000	201.1667	247.2083	0	-26.9167
mic 5	-160.0833	228.0833	274.1250	26.9167	0

Table 2: TDOA matrix with data of measurement of peak detection figure 5.

1.6 Localization

In EPO-4 manual, a solution given to localize the car with the TDOA data. This algorithm is partially implemented. The range difference r_{ij} consist of the difference d_i from microphone i to the car and the difference d_j from microphone j to the car. With the range difference, a hyperbolic curve can be drawn. When all the range difference are drawn, there will be one point where all lines will cross. That is the point where the beacon is. To get this point, a set of quadratic equations need to be solved. Luckily these equations can be transformed to a set of linear equations. This is done in the following way:

$$r_{ij} = d_i - d_j \Rightarrow (r_{ij} + d_j)^2 = d_i^2 \Leftrightarrow r_{ij}^2 + d_j^2 + 2r_{ij}d_j = d_i^2 \tag{1}$$

Inserting

$$d_i^2 = ||\mathbf{x} - \mathbf{x}_i||^2 = ||\mathbf{x}||^2 + ||\mathbf{x}_i||^2 - 2\mathbf{x}_i^T\mathbf{x}$$
$$\text{and } d_j^2 = ||\mathbf{x} - \mathbf{x}_j||^2 = ||\mathbf{x}||^2 + ||\mathbf{x}_j||^2 - 2\mathbf{x}_j^T\mathbf{x}$$

into equation 1, $||\mathbf{x}||^2$ can be eliminated. Here is $\mathbf{x}$ the position of the beacon and $\mathbf{x}_i$, $\mathbf{x}_j$ are positions of the microphones. The resulting equation in vector notation is

$$\begin{bmatrix} 2(\mathbf{x}_j - \mathbf{x}_i)^T & -2r_{ij} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ d_j \end{bmatrix} = r_{ij}^2 - ||\mathbf{x}_i||^2 + ||\mathbf{x}_j||^2$$

This can be done for all microphone pairs, which results in the following matrix of the form $\mathbf{Ay} = \mathbf{b}$

$$\begin{bmatrix} 2(\mathbf{x}_2 - \mathbf{x}_1)^T & -2r_{12} & & & & \\ 2(\mathbf{x}_3 - \mathbf{x}_1)^T & & -2r_{13} & & & \\ 2(\mathbf{x}_4 - \mathbf{x}_1)^T & & & -2r_{14} & & \\ 2(\mathbf{x}_5 - \mathbf{x}_1)^T & & & & -2r_{15} & \\ 2(\mathbf{x}_3 - \mathbf{x}_2)^T & & -2r_{23} & & & \\ 2(\mathbf{x}_3 - \mathbf{x}_2)^T & & & -2r_{24} & & \\ 2(\mathbf{x}_3 - \mathbf{x}_2)^T & & & & -2r_{25} & \\ 2(\mathbf{x}_4 - \mathbf{x}_3)^T & & & -2r_{34} & & \\ 2(\mathbf{x}_5 - \mathbf{x}_3)^T & & & & -2r_{35} & \\ 2(\mathbf{x}_5 - \mathbf{x}_4)^T & & & & -2r_{45} & \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ d_2 \\ d_3 \\ d_4 \\ d_5 \end{bmatrix} = \begin{bmatrix} r_{12}^2 - ||\mathbf{x}_1||^2 + ||\mathbf{x}_2||^2 \\ r_{13}^2 - ||\mathbf{x}_1||^2 + ||\mathbf{x}_3||^2 \\ r_{14}^2 - ||\mathbf{x}_1||^2 + ||\mathbf{x}_4||^2 \\ r_{15}^2 - ||\mathbf{x}_1||^2 + ||\mathbf{x}_5||^2 \\ r_{23}^2 - ||\mathbf{x}_2||^2 + ||\mathbf{x}_3||^2 \\ r_{24}^2 - ||\mathbf{x}_2||^2 + ||\mathbf{x}_4||^2 \\ r_{25}^2 - ||\mathbf{x}_2||^2 + ||\mathbf{x}_5||^2 \\ r_{34}^2 - ||\mathbf{x}_3||^2 + ||\mathbf{x}_4||^2 \\ r_{35}^2 - ||\mathbf{x}_3||^2 + ||\mathbf{x}_5||^2 \\ r_{45}^2 - ||\mathbf{x}_4||^2 + ||\mathbf{x}_5||^2 \end{bmatrix}$$

With this the location of the car can be calculated. This can be done by computing the pseudo-inverse of $\mathbf{A}$, i.e. $\mathbf{y} = \mathbf{A}^\dagger\mathbf{b}$.
To confirm that this algorithm works, a function is written to generate TDOA data. The function `make_tdoa` can be found in appendix A.1.4. This function calculates the range from all the microphone to the car. With this the difference between the microphone pairs is calculated. With this data the localization was tested and confirmed to work most of the times. When the given position is on the middle axis of the field, the deviation is maximum of 220 cm in a field of 600 cm × 600 cm. To counter this, the least squares solution is used. Least squares is also not perfect, but the deviation is maximum 13 cm.

location [x, y]	pseudo inverse	LS
300, 300	300, 300	300, 300
300, 150	300, 175	300, 152
300, 0	300, 30	300, 3
300, 450	300, 381	300, 445
300, 600	300, 420	300, 589
150, 300	220, 300	154, 300
600, 300	380, 300	587, 300

Table 3: Example of deviations from original position

1.7 Checking calculated position

To check if the found location is correct, it will be checked for validity. This is done by using the previous (verified) location. If the car drives straight for a certain amount of time, the traveled distance cannot be larger than a known value. For instance, if the car drives for 1 second the travelled distance is about 0.75 m. If the found location is more than 1 m away, it cannot be correct and will be rejected. Also the cars orientation is kept and checked. For instance, if the car is known to be driving parallel to the positive x axis, and to get from the verified location to the found location, a sharp turn has to be taken, the found location cannot be correct and will be rejected. Because the location algorithm is not accurate in the middle of the playing field, every location found within a meter from the middle will be rejected. In figure 8 the rejection zone van be seen. When the car is in the middle it will drive as long as necessary to get out of the middle. When a location outside the 1 meter radius from the middle is found, this location is tested for validity. Also a location more then 1 meter outside the playing field is rejected. The matlab code for localization and checking of positions can be found in appendix ??.

1.8 Resulting Design

The combined design for determining the position of the car is implemented in the control loop. First the function `TDOA` in listing 5 is to record audio and pass this on to the `ch5_tdoa_final` function. As seen in this function audio will be recorded from 5 channels with a sample rate of 48 kHz, the time to record will be 0.25 s. This time is enough to guarantee that at least 2 samples will be within the recorded sample, one of these samples will be complete enough to use for the deconvolution.

Listing 5: function TDOA

```
function [ TDOA_data ] = TDOA
%TDOA runs the tdoa and returns a column with distances differences
Fs = 48000;
Trec = 0.25;
disp('Sampling');
measured_data = pa_wavrecord(3, 7, Fs*Trec, Fs, 'asio');
disp('TDOA calculating');
[TDOA_matrix, TDOA_data] = ch5_tdoa_final(measured_data);
end
```

Within `ch5_tdoa_final` the data will be split in data of each microphone, this data will be the input of the channel estimation. The channel estimation will return a estimated channel response, this response will be the input of the peak detection each microphone will receive their own channel response. The peak detection will pass the found position to a simple loop that creates the TDOA matrix. The function `ch5_tdoa_final` can be found in appendix ??. This matrix will be the input of the localization algorithm, this algorithm determines the position of the car in x,y and z coordinates. Only the x and y coordinates are of importance, these will be updated in the global variable `position`. In the localization function the determined position will be checked if it is correct. If it does not "pass" the TDOA will be measured again, after three failed attempts to determine the position the car will first move before attempting again.

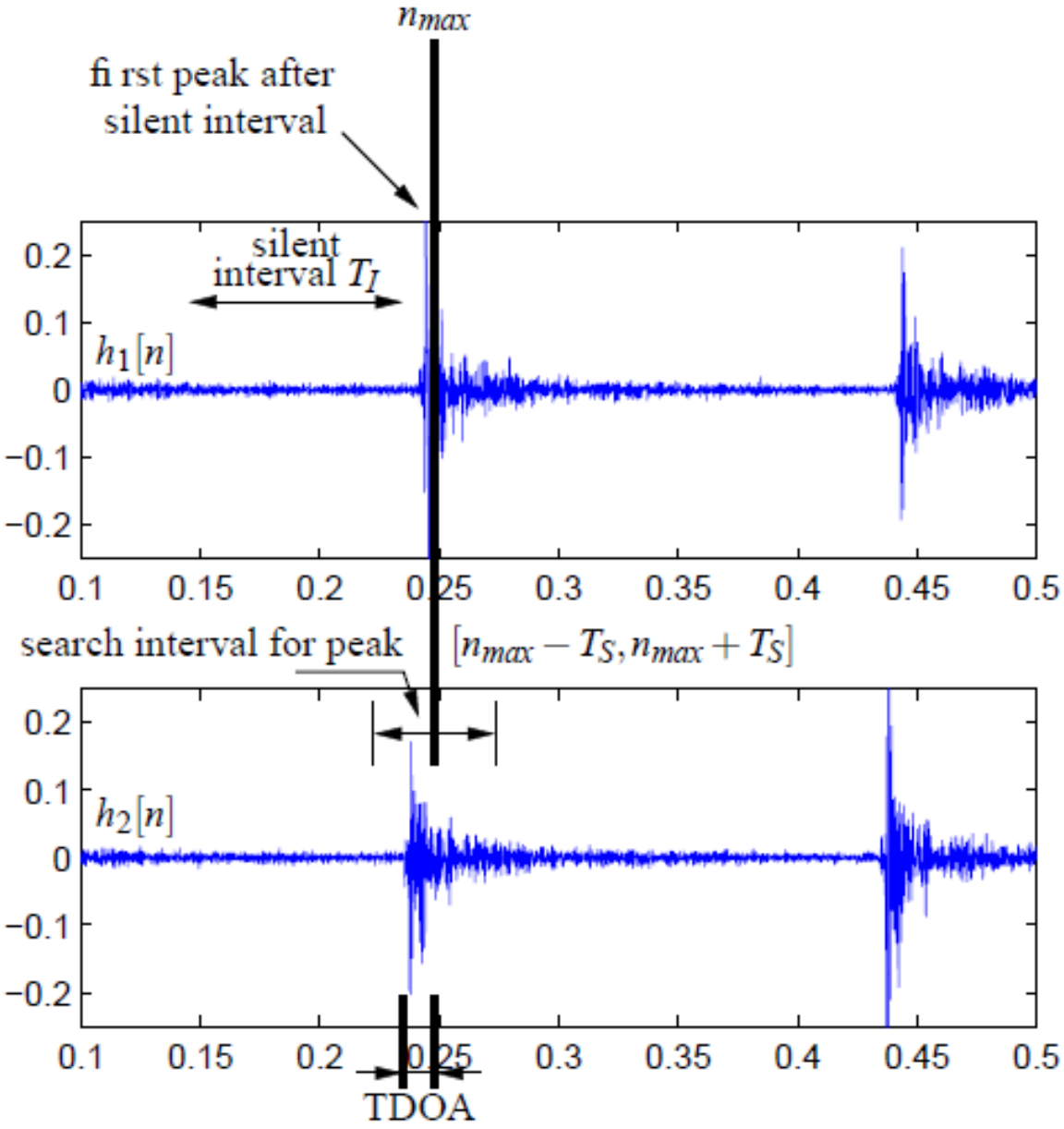

Figure 7: TDOA search window

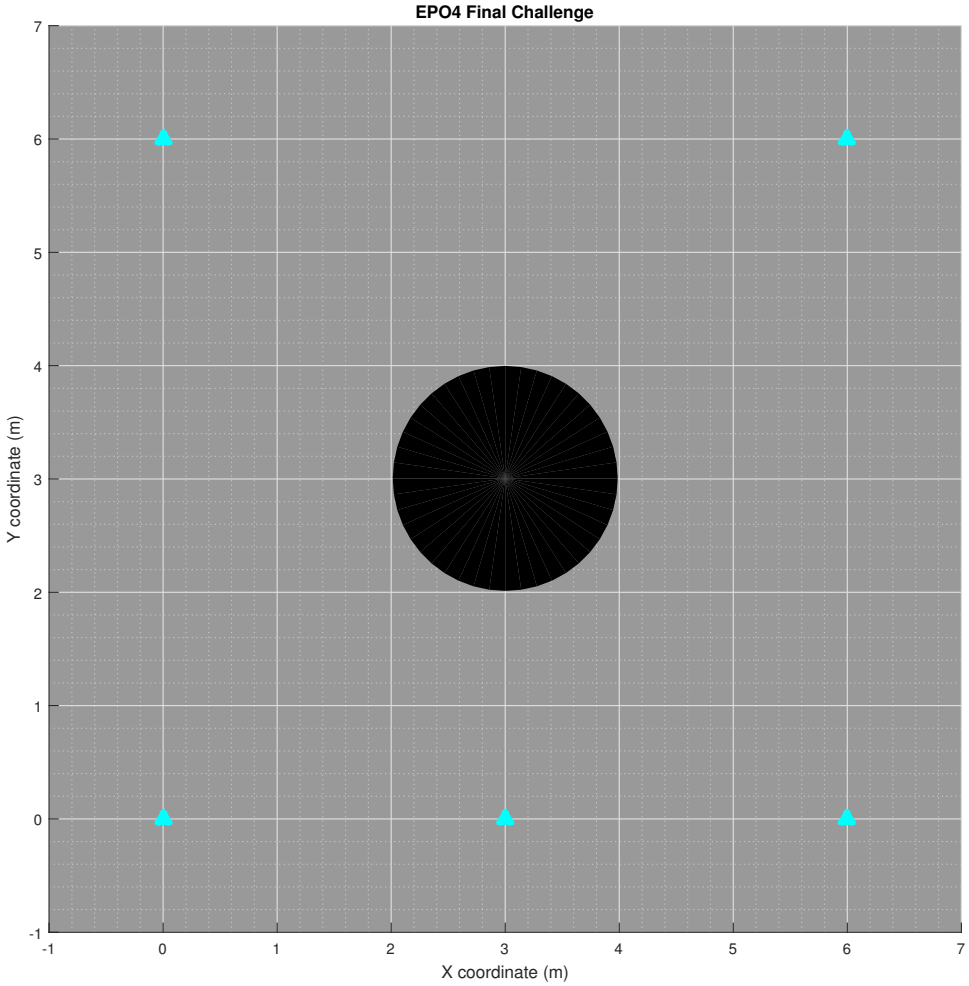

Figure 8: Zone of all locations that will not be put through planner cause of uncertainty in the localization, instead the car will drive a bit further for better localization results.

1.9 Testing

For the testing of the multichannel localization, a test setup was used. In table 4 the bounds of the field are the positions of the microphones one through four. With this field ten measurements are done, the calculated position of these measurements can be seen in table 5. The reference used by the deconvolution is a measurement at one of the microphones. Because of the small distance to the microphone this can be seen as the impulse response of the training sequence through the microphone.

Looking at the results in table 5 the localization is not accurate and does not always work. During the test it was discovered that the first localization algorithm used does not provide an accurate location when the beacon is close to the middle of the field. This is because two of the TDOA values are zero, this happens when microphones are at equal distance from the beacon. For the horizontal middle line the distance from microphone 1,4 and 2,3 to the beacon is the same. This makes it hard to calculate a correct x position, the y position will be calculated correctly. For the vertical middle line the microphone pairs 1,2 and 3,4 are the same, this prevents the localization to correctly calculate the y position.

Setting of the beacon parameters:	Nbits: 32 Timer0: 2 Timer1: 4 Timer3: 7
(x,y,z) locations of the microphones:	Mic 1: 0, 0, 30 Mic 2: 413, 0, 30 Mic 3: 413, 210, 30 Mic 4: 0, 210, 30 Mic 5: 173, 0, 77
Measurement 1:	Beacon at Mic 1
Measurement 2:	Beacon at Mic 2
Measurement 3:	Beacon at Mic 3
Measurement 4:	Beacon at Mic 4
Measurement 5:	Beacon at Mic 5
Measurement 6:	Location(x,y,z): (A) 102, 70, 26
Measurement 7:	Location(x,y,z): (B) 155, 105, 26
Measurement 8:	Location(x,y,z): (C) 216, 88, 26
Measurement 9:	Location(x,y,z): 203, 210, 26
Measurement 10:	Location(x,y,z): 355, 75, 26

Table 4: Measurement position of test setup.

measurement	1	2	3	4	5
x	-0.7041	250.8521	393.7024	3.8097	167.8806
y	-9.9271	394.9473	199.2680	201.5119	1.4720
z	-16.3226	1.1500e+03	19.9370	10.4180	229.1798

measurement	6	7	8	9	10
x	109.1353	216.1429	220.9459	204.5990	136.2869
y	73.1398	105.0000	75.3259	63.1943	176.8470
z	-10.6656	-594.9538	367.3060	-529.2243	-217.4669

Table 5: Localization results for tests.

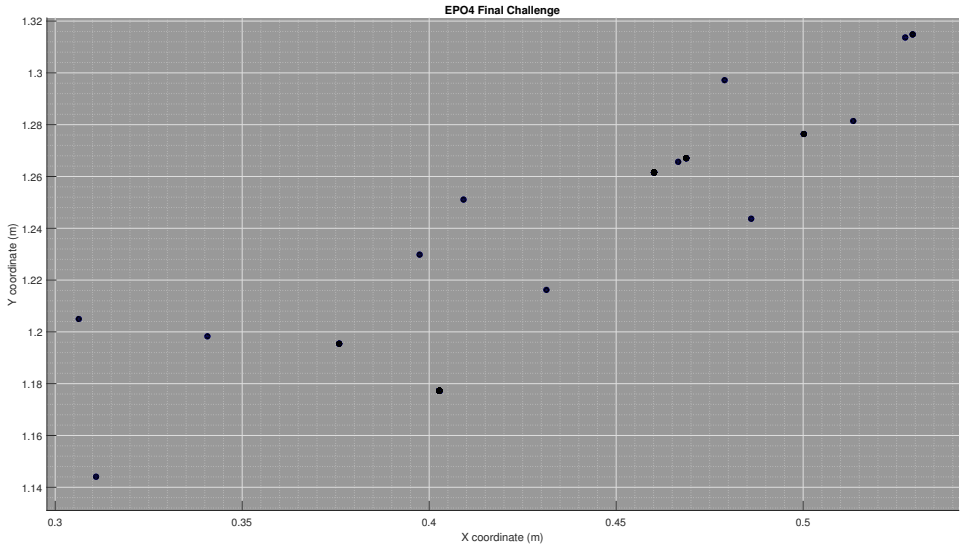

Figure 9: Positions detected by the code on the final test field

1.9.1 Field testing

The TDOA and localization code is also tested on the field that will be used for the final challenge. The car stopped at a position and the code ran several times. In figure 9 the results are visualized, the black dots represent the positions detected. The localization of the car is determined with an accuracy of $\pm$ 10 cm. This should be enough for the car to reach it's destination within 30 cm, which is required for the final challenge.

1.10 Conclusion

The localization of the car is done with a reasonable accuracy and it should be high enough for the car to reach it's goals. However with a sample rate of 48kHz the location of the car could be determined more precise. Sound travels around $1/(48\,\text{kHz}) \cdot 340\,\text{m}\,\text{s}^{-1} = 0.7$ cm per sample so it should be possible to determine the position of the car with a higher resolution than $\pm$10 cm. The accuracy of the position is reduced when the car approaches the positions where x = 3m or y = 3m. This issue is now solved through filtering the positions and driving a bit further in case the car is near the mid lines ($x = 3\,\text{m}$ or $y = 3\,\text{m}$) of the field and no possible correct position is detected. It could be resolved though by using a different algorithm for the localization of the car.

2 Route planning

2.1 Introduction

The role of the route planner is to decide how the car should move given its objective. This can be a number of different things, for example, making a turn, driving a straight line or stopping all-together.

2.2 Design

The only variable that is really fixed is the `route` variable, containing the waypoints (if there are any) and the destination. All other variables and measurements are approximations and are prone to errors and uncertainties. That's why the design of the planner is made with compensation and self-correcting features in mind from the start.
This is the reason for the main design choice, namely that there is no fixed trajectory which is tracked. Instead, the `planner` is executed 'fresh' after each successful localization attempt by the Control Loop, which is explained in further detail in section 3. This means that the only thing that the planner needs to know in addition to the aforementioned `route` variable, is the current position and orientation of the car.

2.2.1 Specifications and Goals

The information available to the planner is summed up in table 6.

Name	**Explanation**
`position`	matrix containing all the known positions and orientations of the car, with the first entry being the start position and the last entry being the current position.
`car.voltage`	contains the current voltage of the car in Volts.
`static_positions.route`	contains the waypoints (if there are any) and the destination of the car in x and y coordinates.
`static_positions.point`	keeps track of which waypoint/destination the car is traveling to.
`car.steer_straight`	contains the steer setting for driving straight.

Table 6: Information and variables available to the planner.

The data and information that the planner should provide to the Control Loop is available in table 2.2.1.

Name	Explanation
`speed`	the cars needed speed setting including voltage drop compensation.
`car.d_theta`	difference in angle between the car and the next waypoint/destination.
`car.v_factor`	factor between `0` and `1` depending on the starting voltage and the current voltage of the car.
`steer`	steer setting of the car depending on whether to make a turn or not and which way.
`car.status`	status of the car, `1` when at a waypoint, `2` when at the destination and `0` elsewhere.
`car.did_turn`	toggles between `0` and `1` reflecting whether the last driven action of the car was a turn or not.
`time`	time in seconds the car has to move, this movement can be a turn, driving straight or standing still, depending on the other output variables.

Table 7: Information and variables returned to the Control Loop.

2.2.2 Code

The code of the Planner, which can be found in appendix A.2, consists of two parts. In the first part all the needed information is generated and in the second part a decision tree build out of `if`-statements figures out what needs to be done with that information.

speed setting adjustment
When driving with supercaps instead of a batterypack there is a significant voltage drop over time, to compensate for this the car will drive at different speed settings at different voltage levels. In addition to this, every car behaves differently, therefore the values in this code are determined by trial and error, which is more generally explained in paragraph 2.3.1.

Listing 6: Speed setting adjustment for different Voltages

```
8  if car.voltage >= 17 % speed compensation for voltage drop
9      speed = 158; % IMPORTANT, CALIBRATE FOR 1.3s for 1m
10     disp('Voltage level: normal')
11 elseif car.voltage >= 14 % adjust by hand acordingly
12     speed = 162;
13     disp('Voltage level: low')
14 else
15     speed = 165;
16     disp('Voltage level: drained')
17 end
```

generation of information
First, the number of fixed points is extracted from the `static_positions.route` variable. Then the angle that the car needs to turn and distance it has to travel is calculated using basic trigonometry and the Pythagorean theorem. Finally the voltage compensation factor is determined using the current car voltage.

Listing 7: Generation of needed information

```
21 [~,numberofpoints] = size(static_positions.route); % number of
       points (waypoints + destination)
22 d_x                = static_positions.route(1,static_positions.
       point)-position(1,end); % delta x
23 d_y                = static_positions.route(2,static_positions.
       point)-position(2,end); % delta y
24 desired_theta      = atan2d(d_y,d_x); % desired orientation of the
       car
25 car.d_theta        = desired_theta - position(3,end); % difference
       between car orientation and desired orientation
26 distance           = sqrt(d_x^2 + d_y^2); % distance from car to
       destination/waypoint
27 test_data.dtheta   = [test_data.dtheta, car.d_theta];
28 car.v_factor       = 1; %max car voltage / car.voltage;
```

Decision tree consisting of nested `if`-statements.
This part of the code performs the necessary checks to determine what drive settings need to be generated. The first check that occurs is whether the car has arrived within 30 cm of its current destination, which can either be a waypoint or the final destination. When this is the case the `planner` outputs variables to the Control Loop to stand still and informs whether it's a waypoint or destination.

Listing 8: Check 1: Destination/Waypoint check

```
32 if distance <= 30
33     steer = car.steer_straight;
34     speed = 150; % standing still
35
36     if static_positions.point >= numberofpoints
37         time = 40;
38         disp('ARRIVED AT DESTINATION')
39         car.status = 2;
40     else
41         static_positions.point = static_positions.point + 1; % keep
               track of where we're going
42         disp('ARRIVED AT A WAYPOINT')
43         time = 6;
44         car.status = 1;
45     end
46 else
```

When that check returns false the car needs to move, first the code checks whether the previous movement was a turn. This is done because the cars orientation needs to be calculated and this can only be done when a straight line has been driven from its previous to its current location. So if the previous movement was a turn, the car is ordered to drive a bit in a straight line so that the next time it performs this check it will not have made a turn.

Listing 9: Check 2: turn check

```
47     car.status = 0;
48     if car.did_turn == true
49         time          = 0.6 / car.v_factor;
50         steer         = car.steer_straight;
51         car.did_turn = false;
52         car.did_last_turn = true;
53         disp('Driving straight a bit to provide data for
               orientation after the turn.')
54     else
```

The next part checks whether the car needs to make a turn, if it doesn't need to make a turn the code differentiates between 2 possible situations. Situation 1: the car is almost at its current destination. If that is the case, the function `straight`, found in appendix A.2.3, returns the time it takes to drive to that destination. Situation 2: the car has quite a long way to go before reaching its destination, so it drives a predefined amount of time in a straight line. It does this because, due to the aforementioned inaccuracies and uncertainties, it can occur that it is slightly of its course. Therefore this will all resolve itself by running the `planner` again after getting a bit closer to the goal destination.

Listing 10: Driving in a straight line towards the waypoint/destination.

```
57          if abs(car.d_theta) <= 5 % no turn needs to be made
58              disp('No turn!');
59              if distance <= 150 % the car is close to the waypoint /
                    destination.
60                  time    = straight(distance);
61                  steer   = car.steer_straight;
62                  disp('Almost there.')
63              else % distance is greater than 1.5m
64                  time    = 1 / car.v_factor;    % return a 1s drive
65                  steer   = car.steer_straight;  % straight
66                  disp('Not there yet.')
67              end
68          else % turn is needed
```

If it has to make a turn it calculates the time it needs to do that using the `turn` function, found in appendix A.2.2, and provides the correct steer setting depending on whether the variable `car.d_theta` is positive or negative.

Listing 11: Making a turn.

```
55              disp('Lets turn')
56              car.did_turn    = true; % make sure that we drive a bit
                    in a straight line after a turn is made
57              if car.d_theta <= 0
58                  if d_x <= 0 % fix when driving from right to left
59                      steer = 200;
60                      disp('to the left!')
61                  else
62                      steer = 100;
63                      disp('to the right!')
64                  end
65              else
66                  if d_x <= 0 % fix for when driving from right to
                        left
67                      steer = 100;
68                      disp('to the right')
69                  else
70                      steer = 200;
71                      disp('to the left!')
72                  end
73              end
74              time = turn(car.d_theta);
75          end
76      end
77  end
```

2.3 Testing

To implement the `turn` and `straight` functions mentioned in the previous code section, several tests were done to determine the time it takes to make a certain turn or drive in a straight line for a certain length. Because testing with supercaps was unnalowed, testing was done with the batterypacks instead. This meant that no detailed effects of the voltagedrop on the timings could be found. To simplify things further, linear approximations were used, it was decided that as long as the timings were not causing overshoot, the final goal should be reached because of the compensating and self-correcting behaviour of the system.

To determine the linear approximation of our turn timings the measurementdata shown in table 8 was inter- and extrapolated. This resulted in the following code in the `turn` function (found in appendix A.2.2).

Listing 12: The `turn` function.

```
8  if d_theta > 5
9      time = (0.01422222222*d_theta +0.5) / car.v_factor;
10 elseif d_theta < -5
11     % negative d_theta
12     time = (-0.0142222222*d_theta +0.5) / car.v_factor;
13 else
14     time = 0;
15 end
```

Angle (`d_theta`)	`time`
`-180`	3.45 s
`-90`	2.05 s
`0`	0 s
`90`	2.11 s

Table 8: Measurement data for linear approximation of turn timings.

To determine the linear approximation of the timings for driving in a straight line we calibrated the speed setting to produce a 1.3 s drive for covering 1 m. The resulting `straight` function can be found in appendix A.2.3.

To get an idea of the behaviour of the `planner` when included in the Control Loop, small messages were added to the code. These messages show up in the console while driving with updates on what decisions are made was chosen and in conjunction with saved test data, errors or unexpected behaviour was easy to spot. However as has become apparent while doing the final challenge, there was an error that slipped through, which will be mentioned in section **??**.

2.3.1 Trial and Error & Findings

Between each run small changes to the variables were made to ensure a viable 'maximum car voltage' performance. These found settings would degrade when the voltage would start to drop but the code is able to compensate for that by rerunning itself untill the car arrives at the destination. This means that in optimal state the car would make only one turn per waypoint and more turns when the voltagedrop makes the turns less precise.

During both standalone and 'as part of the control loop' testing, unexpected behaviour was found. It turned out, for example, that when driving from the right to the left side, instead of the usual left-to-right route, the car would turn left when it needed to make a right turn and vice-versa. This has been fixed by first checking the driving direction before determining whether a turn should be to the right or to the left.

2.4 Conclusion

This implementation of route planning has proven to be useful due to its ability to recover from errors caused by invalid localization or other uncertanties. Unfortunately it failed the distance check as mentioned in listing 8 and 10 because of a conversion error during the final challenge, but other than that, the design and implementation proved to be simple and effective.

To improve on this design the linear approximations and guessed factors could be removed and replaced by measured curves for different voltage levels. Also it could be expanded further to include collision detection handling, which could include driving backwards and logging found obstacles.

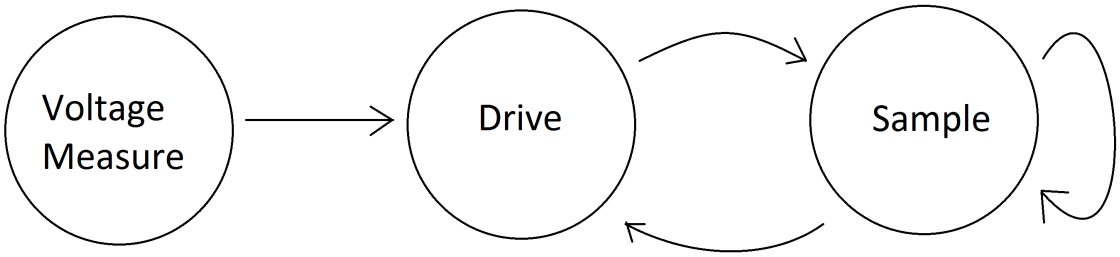

Figure 10: State machine implemented in the control loop

3 Control Loop

The control loop is the main function. All other functions are called from this control loop.

3.1 Overview

To implement the control loop, a MATLAB `timer` is used. The timer object is very useful because it handles a few things which are hard to implement in a normal while loop. A timer has 4 main functions, which are `StartFnc`, `TimerFnc`, `StopFnc` and `ErrorFnc`. To initialize the timer, the control loop function is called. Here are `StartFnc`, `TimerFnc` etc. assigned. Also, 5 global variables are initialized. These variables are global so they don't have to be in the input and/or output of a function. For instance, one of these globals is to keep test data. From within every function this variable can be altered.
The `timer_startFcn` is the `StartFnc` of the timer. This is used to initialize the figure to map the car. This function is called just before the car starts driving. The `timer_loop` is the `TimerFnc` and is the main loop. In this function the car will drive and all the functions are called. A state machine is used to keep everything clear and structured. The 3 states are VoltageMeasure, Drive and Sample. These states will be explained in later. The `timer_stopFcn` is the `StopFnc` of the timer. It is used to clean when the function is done driving the car. Last but not least is the `timer_error`, which is the `ErrorFnc`. This function can handle every error thrown by one of the other functions. When an error is thrown, this function will send a command to the car to stop driving. This is so the car wont drive without control and drive against an object.

3.2 VoltageMeasure

For the final challenge the car has to be charged wireless, this is done with a air-core DC/DC converter. The super capacitors need to be charged to 17 V. Via the wireless communication the current voltage can be read. The voltage measured fluctuates heavily, because of this an outlier will count as 17 V. To make sure the super caps are charged to 17 V the measurement will count each time a value above 17 V is measured, if the count is five it will assume the car is charged. After the car is charged the next state will be `Drive`.

3.3 Drive

First thing that this state does is call the planner function. In the planner the parameters for the car are calculated. If the car is near its target, the planner will give a signal to the control

loop. When this signal is given, a sound will be played by the computer to indicate that it has arrived. When all the parameters are determined, a drive command has to be sent to the car to make it drive. A separate function is made to do this to be sure all the parameters are determined before transmitting it to the car. The function drive_car.m, in appendix **??**, has as inputs a speed setting, steer setting and a time to drive. It executes the drive command for the time needed using a pause command to wait, before transmitting a roll out command which is a speed setting of 150 and a steer setting that corresponds to driving a straight line. After this is done, the next state will be Sample.

3.4 Sample

First the TDOA data will be determined. With this data, a location will be determined with the function localize. If the found location passes all tests, the next state will be `Drive`. But if the found location does not pass the tests, a fail code will be given. When the localization fails, the next state will be `Sample` and new TDOA data will be determined. This will be done until 3 tests fail. When 3 tests fail, the car will drive a little bit forward to find a better spot for a measurement. When the tests fail 9 times in a row, a different action is taken depending on the failed test. When the found location fails because it is too far away, after 9 times it is possible that it is actually correct. So this location will then be accepted. If the location fails because it is at the wrong angle, the car will drive straight until is finds the correct angle. If it does not pass the test within 10 tries, the position will be accepted and a new orientation will be determined.

3.5 Mapping the car

To visualize the location of KITT the matlab class EPO4figure from blackbloard, appendix **??**, is used to map the car location, mic positions, waypoint, destination and obstacle locations.

The class has 5 functions: `setMicLoc`, `setWayPoint`, `setDestination`, `setKITT` and `setObstacle` which all require a location in some form as input. A variable `start` is added to the function `setKITT` to prevent the figure to load the last position from a previous test. The last position of the car is shown as a blue dot and the past positions of the car are shown as a black dot, the mic positions are the light blue dots and the violet ones are the destination and waypoint positions as can be seen in figure 11.

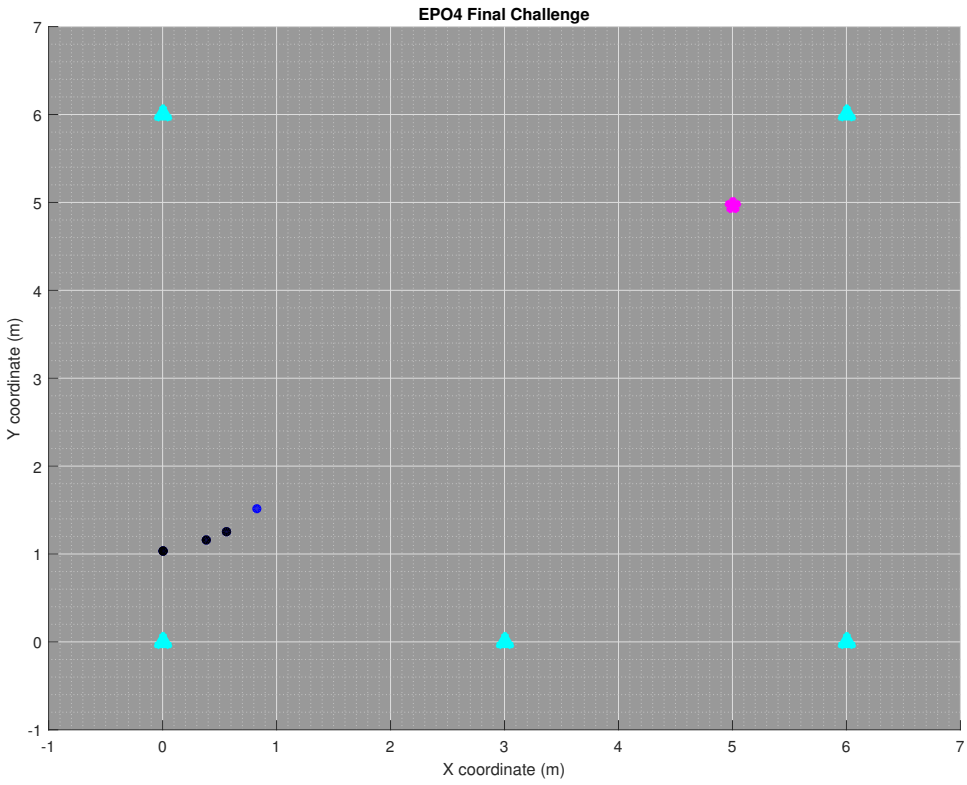

Figure 11: Mapping the locations

4 The Final Challenge

This section describes the performance of our system during the Final Challenge. The car didn't succeed in stopping at the destination and therefore it didn't meet our expectations however we are still satisfied with the code we created. Unfortunately some small mistakes were made and these will be explained below.

4.1 Objective

As already mentioned the first challenge was to drive from point A (0, 103) to point B (500,497).

4.2 Training sequence

The training sequence used for the final challenge is Timer1: 2, Timer2: 0, Timer: 9. These settings are used due to the limited availability of working cars.

4.3 Reference

A different reference signal was used for the final challenge. For completeness of the report this reference signal is shown in figure 12. To create this reference signal a measurement was done close (5cm) to microphone 1 and the desired pulse was cut from this data. The start point of the reference is the first point after the silence period of the signal. At the end a small silence part is included.

4.4 What went wrong and how was it solved?

In figure 13 the accepted car positions are shown. Near the destination, pictured by the pink dot, a strange turn can be observed. The car actually steers away from the destination. The test data showed that the sign of the calculated angle was inversed when the car drives from right to left. This bug was easily solved by checking the driving direction before issuing a steer setting command, as can be seen in listing 11.

In the final run of the final challenge the car actually reached the destination for the first time, the only problem was that it didn't pause there but continued driving. After inspection of the code it became clear that the distance was not calculated correctly and was off by a factor 100. This behaviour can be seen in figure 14. The car still tried to make a turn to approach the destination again but it hit a table on the outside of the field while turning and therefore we stopped the run. It is highly unlikely the cars position would have gotten within 0.3 cm accuracy of the goal and therefore it would have made turns around the destination until the batteries were empty.

In figure 14 the accepted positions are shown for the final run.

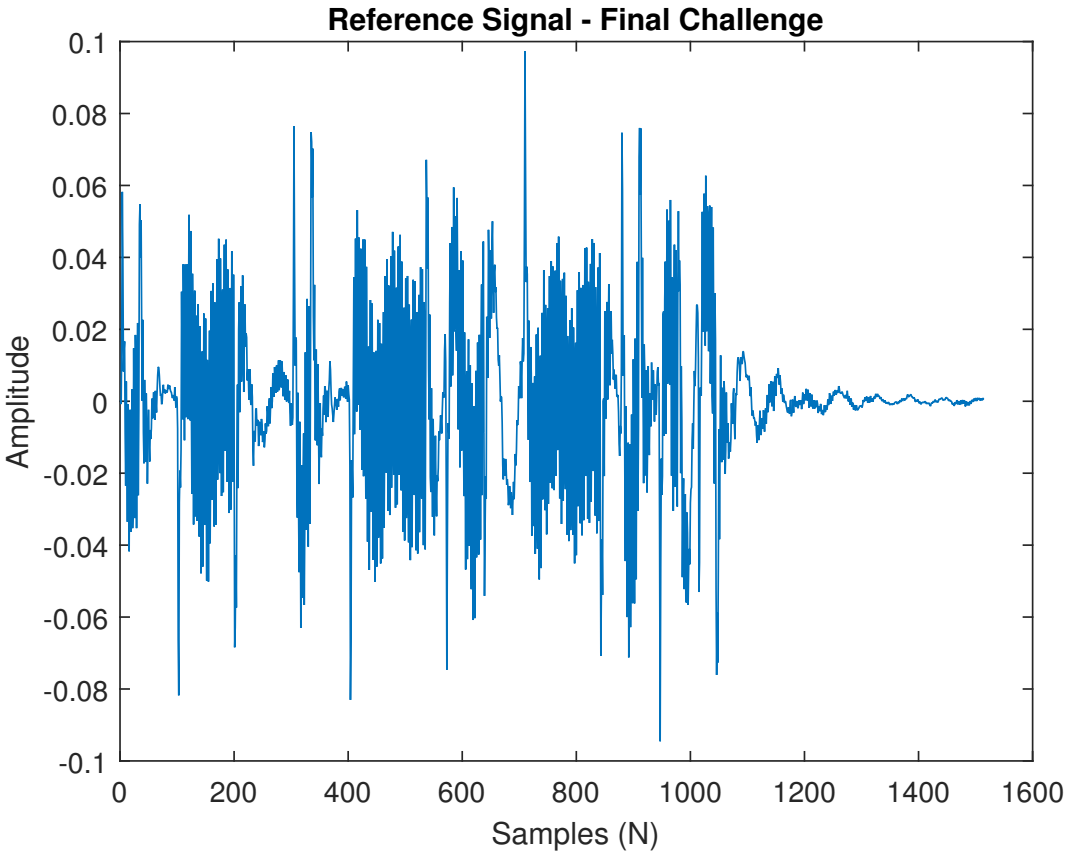

Figure 12: Reference signal used for the Final Challenge

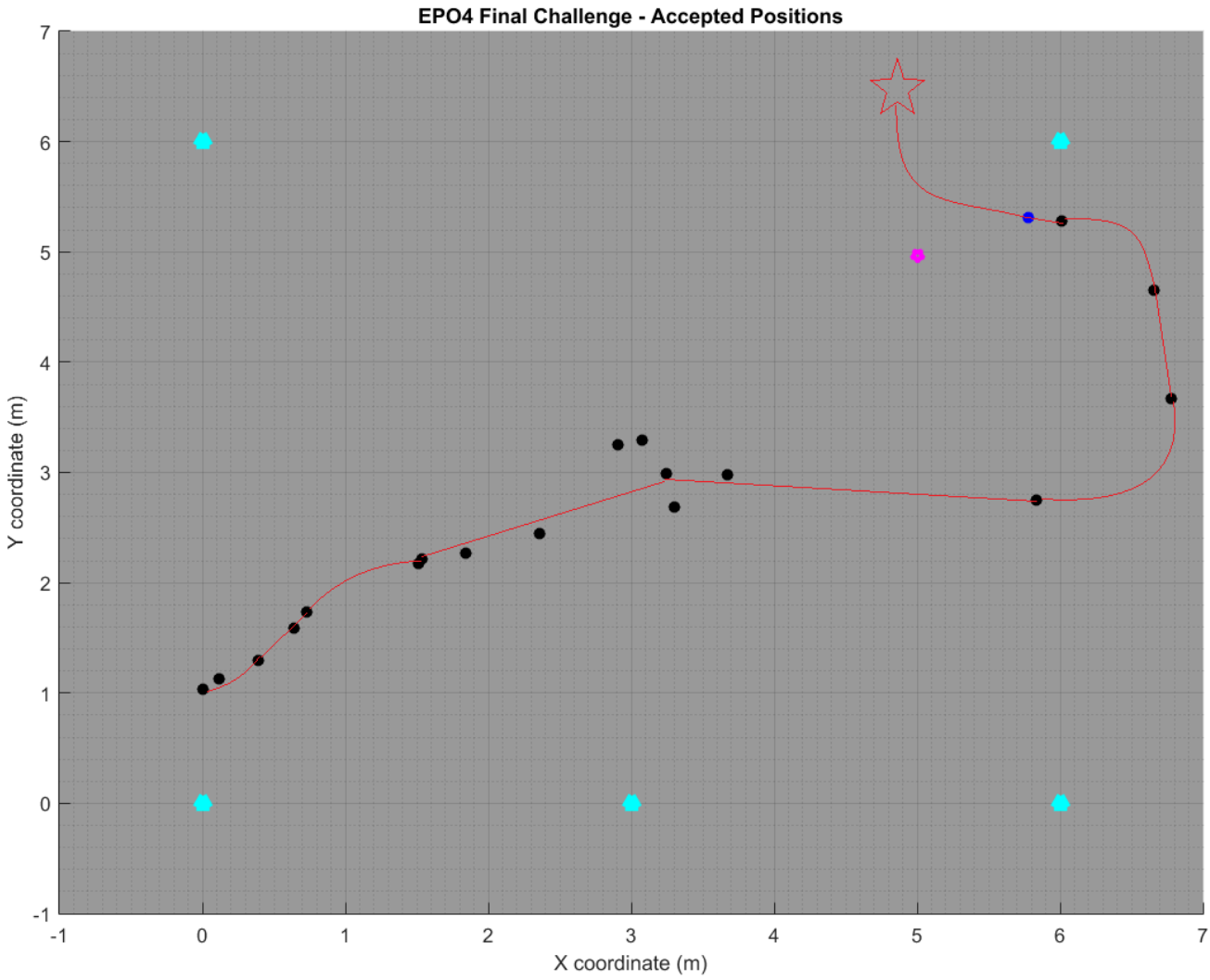

Figure 13: Map of the accepted car locations during a run, the red line is the observed route the car took

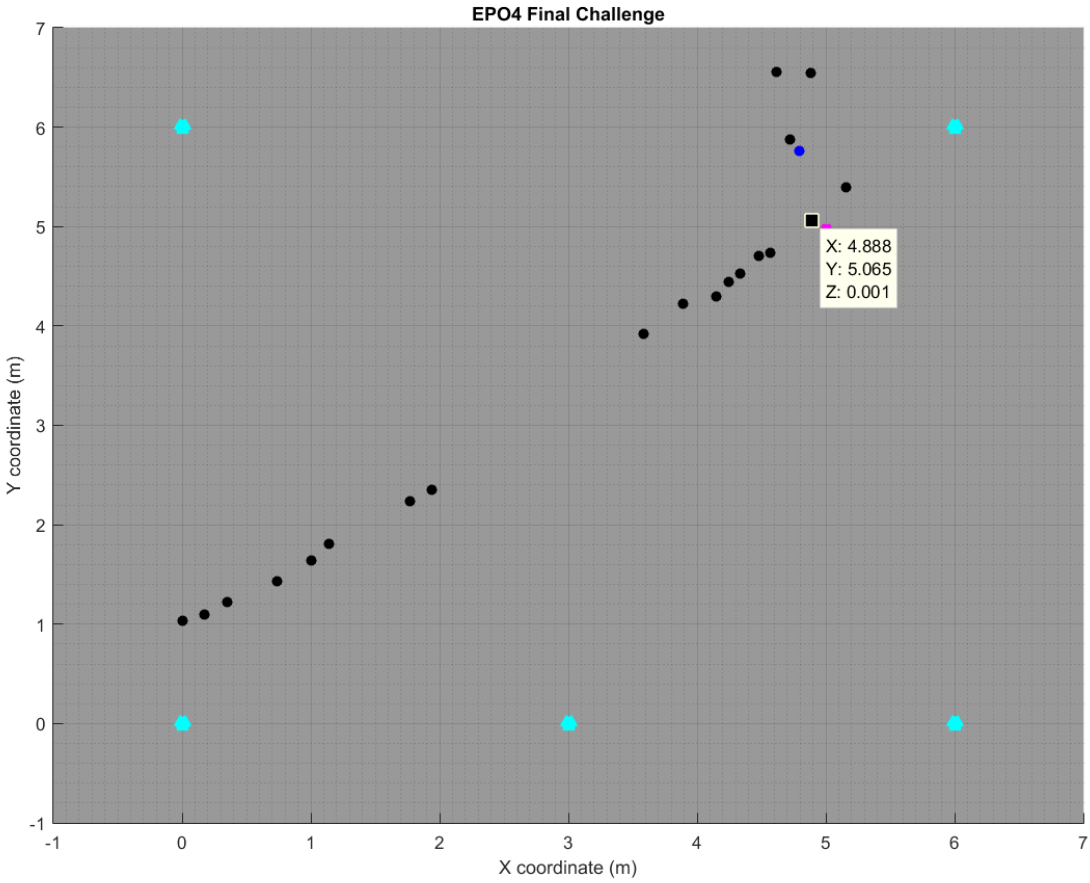

Figure 14: Map of the accepted car locations at the final run, the position selected is in stopping range of the destination

5 Conclusion and Discussion

5.1 Conclusion

In the end the control loop operated well. The TDOA and Localization algorithm were able to determine the position within ± 10 cm accuracy. The route planning worked aswell. We are convinced it will complete the first two challenges if given another chance.

5.2 Discussion

As mentioned before we didn't succeed in completing the objectives on the moment of truth, the final challenge, this was due to some small mistakes made in the code. These mistakes could have been detected before the challenge if the individual codes were tested more thoroughly. The localization algorithm used determines the location of the car with reasonable accuracy. When the car is in the middle of the field the location gets less precise for the mid lines (x = 3m or y = 3m) of the field the location is determined with less accuracy as well. This could have been solved by using a more advanced algorithm for the location estimation. The obtained resolution on all positions with the current algorithm is high enough to complete the challenges and therefore we didn't elaborate this further. To get a higher precision with the current algorithm it would be an option to make a reference signal for every microphone because the microphones have their own slightly different characteristics this should result in a better channel estimation and therefore a better TDOA and better localization.

6 Planning and Teamwork

6.1 Planning

The planning of the final challenge is divided in four weeks. The first three lab day sessions are used for localization and multichannel measurements. the three lab day sessions after are meant for system integration, putting all parts together and writing functions to plan the route and control the car. The last few lab days are used for testing with the final setup. The final report deadline is planned for Friday 19 June noon, for this it was decided that two days before this deadline at 18:00 the individual works are finished and afterwards the written pieces will be corrected and if something is missing written. The final report was divided into parts and each individual will write the pieces they worked on, this can be reviewed on the next page.

6.2 Teamwork

Introductie		Martijn
	Abstract	Martijn
	Introduction	Martijn
Specs		Thomas
	Specs	Thomas
Planning/Teamwork		Martijn
	Planning	Martijn
	Teamwork	Martijn
Plaatsbepaling		Martijn
	Introduction	Martijn
	Training sequence	Jeroen
	Reference	Martijn
	Deconvolution	Martijn
	Peak Detection	Richard
	Localization	Jeroen
	Checking calculated position	Jeroen
	Testing	Richard
	Conclusion	Richard
Object detection?		Martijn
	Introduction	Martijn
	Design	Martijn
	Testing	Martijn
	Conclusion	Martijn
Route planning		Thomas
	Introduction	Thomas
	Design	Thomas
	Testing	Thomas
	Conclusion	Thomas
Loop		Jeroen
	Timer Function	Jeroen
	Determining Voltage	Martijn
	Driving	Richard
	Plotting	Richard
	Challenges	Jeroen
	Error	Jeroen
Conclusie		Richard
Discussie		Richard
Appendix		Thomas

References

[1] A.-J. van der Veen, *EE2L21 Project EPO-4: Autonomous driving challenge*, 24th ed. TU Delft Faculty of Electrical Engineering, Mathematics and Computer Science, 2015. [Online]. Available: https://blackboard.tudelft.nl/bbcswebdav/pid-2422827-dt-content-rid-8391321_2/courses/34309-141504/epo4manual_24apr.pdf

A Appendix

A.1 Localization

A.1.1 Open_com.m

```
comport = '\\.\COM3'; %Bluetooth Module: 3215
EPOCommunications('close');
result = EPOCommunications('open', comport);
status = EPOCommunications('transmit', 'S');
EPOCommunications('transmit','A1');
```

A.1.2 orientation.m

```
%Start Orientation is known. Use this as 0
%For info on atand2d see
%http://nl.mathworks.com/help/matlab/ref/atan2d.html

function Orientation%(position)
%Function to keep track of the orientation of the car
global position

    %Take positive y axis as 0 degree

    x_diff = position(1, end) - position(1, end-1);
    y_diff = position(2, end) - position(2, end-1);

    angle = atan2d(y_diff,x_diff); % * 180 / pi;

    position(3,end) = angle;

end
```

A.1.3 matched_f.m

```
function h = matched_f(x,y)
Ny = length(y);
Nx = length(x);
L = Ny-Nx +1;
xr = flipud(x);
h = filter(xr,1,y);
h = h(Nx:end);
alpha = x' *x;
h = h/alpha;
end
```

A.1.4 matched_f.m

```
function tdoa = make_tdoa(matrix_mics_position, car_location)
% Generates test TDOA data
% input: mic positions, some field position
% output: TDOA data
    [row, col] = size(matrix_mics_position);
    dimensions = length(car_location);
    if(col ~= dimensions) % check for same dimensions
        error('Mics location dimensions dont match with x_location
           dimensions');
    end
    elements = (row * (row - 1))/2;
    tdoa = zeros(elements, 1);
    mic1 = 1;
    mic2 = 2;
    for i = 1:elements
        tdoa(i, 1) = (norm(matrix_mics_position(mic1, 1:col) -
           car_location) - norm(matrix_mics_position(mic2, 1:col) -
            car_location));
        mic2 = mic2 + 1;
        if(mic2 > row)
            mic1 = mic1 + 1;
            mic2 = mic1 + 1;
        end
    end
end
```

A.2 Planner

A.2.1 planner.m

```
function [time,steer,speed] = planner
%input: (global)position(x,y,orientation),(global)next_position(x,y
   )
global car;
global position;
global test_data;
global static_positions;
% compensation for voltage drop of supercaps
if car.voltage >= 17 % speed compensation for voltage drop
    speed = 158; % IMPORTANT, CALIBRATE FOR 1.3s for 1m
    disp('Voltage level: normal')
elseif car.voltage >= 14 % adjust by hand acordingly
    speed = 162;
    disp('Voltage level: low')
```

```
14 else
15     speed = 165;
16     disp('Voltage level: drained')
17 end
18 % end of compensation for voltage drop of supercaps
19
20 % begin computation of needed information
21 [~,numberofpoints] = size(static_positions.route); % number of
       points (waypoints + destination)
22 d_x                = static_positions.route(1,static_positions.
       point)-position(1,end); % delta x
23 d_y                = static_positions.route(2,static_positions.
       point)-position(2,end); % delta y
24 desired_theta      = atan2d(d_y,d_x); % desired orientation of the
       car
25 car.d_theta        = desired_theta - position(3,end); % difference
       between car orientation and desired orientation
26 distance           = sqrt(d_x^2 + d_y^2); % distance from car to
       destination/waypoint
27 test_data.dtheta   = [test_data.dtheta, car.d_theta];
28 car.v_factor       = 1; %max car voltage / car.voltage;
29 % end of computation of needed information
30
31 % start if-based decision tree
32 if distance <= 30
33     steer = car.steer_straight;
34     speed = 150; % standing still
35
36     if static_positions.point >= numberofpoints
37         time = 40;
38         disp('ARRIVED AT DESTINATION')
39         car.status = 2;
40     else
41         static_positions.point = static_positions.point + 1; % keep
               track of where we're going
42         disp('ARRIVED AT A WAYPOINT')
43         time = 6;
44         car.status = 1;
45     end
46 else
47     car.status = 0;
48     if car.did_turn == true
49         time          = 0.6 / car.v_factor;
50         steer         = car.steer_straight;
51         car.did_turn = false;
52         car.did_last_turn = true;
```

```
53          disp('Driving straight a bit to provide data for
                orientation after the turn.')
54      else
55          car.did_last_turn = false;
56          disp('Check whether a turn has to be made.')
57          if abs(car.d_theta) <= 5 % no turn needs to be made
58              disp('No turn!');
59              if distance <= 150 % the car is close to the waypoint /
                    destination.
60                  time    = straight(distance);
61                  steer   = car.steer_straight;
62                  disp('Almost there.')
63              else % distance is greater than 1.5m
64                  time    = 1 / car.v_factor;    % return a 1s drive
65                  steer   = car.steer_straight;  % straight
66                  disp('Not there yet.')
67              end
68          else % turn is needed
69              disp('Lets turn')
70              car.did_turn    = true; % make sure that we drive a bit
                    in a straight line after a turn is made
71              if car.d_theta <= 0
72                  if d_x <= 0 % fix when driving from right to left
73                      steer = 200;
74                      disp('to the left!')
75                  else
76                      steer = 100;
77                      disp('to the right!')
78                  end
79              else
80                  if d_x <= 0 % fix for when driving from right to
                        left
81                      steer = 100;
82                      disp('to the right')
83                  else
84                      steer = 200;
85                      disp('to the left!')
86                  end
87              end
88              time = turn(car.d_theta);
89          end
90      end
91  end
92  % end of decision tree
93  end
```

A.2.2 turn.m

```
function [ time ] = turn(d_theta)
global car;
%TURN Summary of this function goes here
%   returns time to make a certain turn of d_theta

% Moet nog getweaked / gecalibreerd worden
% positive d_theta
if d_theta > 5
    time = (0.01422222222*d_theta +0.5) / car.v_factor;
elseif d_theta < -5
    % negative d_theta
    time = (-0.0142222222*d_theta +0.5) / car.v_factor;
else
    time = 0;
end
end
```

A.2.3 straight.m

```
function [ time ] = straight(distance)
global car;
%TURN Summary of this function goes here
%   returns time to drive a certain distance lower than 1.5m (for
   now)

% Moet nog getweaked / gecalibreerd worden
time = (2/1.5) * distance %car.v_factor;

end
```

A.3 Loop

A.3.1 control_loop.m

```
function t = control_loop() %Timer functie met een acceleratie tijd
    en een remtijd

global position
global voltage
global car  % need this don't delete
global static_positions
global test_data

fail_factor = 0;
```

```
drive_counter = 0;
middle_counter = 0;

test_data.pass = 0;
test_data.TDOA = [0;0;0;0;0;0;0;0;0;0];
test_data.measured = zeros(1,12000,5);
test_data.dtheta = 0;
test_data.cartime = [0 0 0];
test_data.pos_tdoa = [0;0;0];
static_positions.origin = [0;103;0]; %start position
static_positions.destination = [500;497;0];
static_positions.waypoint = [0;0;0];
%static_positions.waypoint = [193;400;0];
static_positions.point = 1; % need this in planner
[sound, Fs] = audioread('ObjectiveComplete.mp3');
player = audioplayer(sound,Fs);
if static_positions.waypoint == [0;0;0]
    static_positions.route = [static_positions.destination];
else
    static_positions.route = [static_positions.waypoint,
       static_positions.destination];
end
static_positions.mic_positions = [0 0 40; 600 0 40; 600 600 40; 0
   600 40; 300 0 46];

car.did_turn = false;
car.did_last_turn = false;
car.steer_straight = 150;
position = static_positions.origin; %Postion in centimeters

state = States.VoltageMeasure;
t = timer;
t.TimerFcn = @timer_loop;
t.StartFcn = @timer_startFcn;
t.StopFcn = @timer_stopFcn;
t.ErrorFcn = @(~,~)timer_error;
t.StartDelay = 0;
t.Period = 0.01;
t.ExecutionMode = 'fixedRate';


    function timer_startFcn(timerObj, timerEvent)
        disp('Started - Load figure');
        EPO4figure; %Load the figure
        EPO4figure.setMicLoc(static_positions.mic_positions/100) %
           Update Mic Positions
```

```
53          EPO4figure.setDestination(static_positions.destination/100)
                ;
54          EPO4figure.setKITT([position(1,end)/100 position(2,end)
                /100], 1); %Update car position
55          if static_positions.waypoint == [0;0;0]
56              disp('No waypoint');
57          else
58              EPO4figure.setWayPoint(static_positions.waypoint/100);
59          end;
60          EPOCommunications('transmit','A0');
61      end
62
63
64      function timer_loop(timerObj, timerEvent)
65          switch(state)
66              case States.VoltageMeasure
67                  disp('Measuring Voltage')
68                  measure_voltage;
69                  if(voltage.done == true)
70                      state = States.Drive;
71                  end
72              case States.Drive %Example, states and flow can be
                    altered.
73                  disp('Driving straight');
74                  fail_factor = 0;
75                  drive_counter = 0;
76                  %Status request
77                  status_update;
78                  %Planner
79                  [car.time, car.steer, car.speed] = planner;
80                  test_data.cartime = [test_data.cartime, car.time];
81                  if(car.status == 1)
82                      %Car is at waypoint
83                      play(player)
84                      disp('WAYPOINT')
85                  elseif(car.status == 2)
86                      %Car is at destination
87                      play(player)
88                      disp('At destination!')
89                      EPOCommunications('transmit','A1');
90                      stop(timerObj);
91                  end
92
93                  drive_car(car.speed, car.steer, car.time);
94                  state = States.Sample;
95
96              case States.Sample
```

```
97                  disp('Sampling after straight');
98                  %Sample
99                  %TDOA
100                 TDOA_data = TDOA;
101                 disp('TDOA afgerond');
102                 test_data.TDOA = [test_data.TDOA ,TDOA_data];
103                 %Localize
104
105                 pass = localize_5ch(TDOA_data, ((100/1.3) * car.
                        time) + (15 * drive_counter) + (50 *
                        middle_counter));
106                 %pass = 1;
107                 if(pass ~= 8)
108                     middle_counter = 0;
109                 end
110                 test_data.pass = [test_data.pass; pass];
111
112                 if(pass == 1)
113                     EPO4figure.setKITT([position(1,end)/100
                            position(2,end)/100], 0); %Update car
                            position
114                     state = States.Drive;
115                     Orientation; %function without nonglobal
                            arguments
116                 else
117                     state = States.Sample;
118
119                     fail_factor = fail_factor + 1;
120                     if(pass == 8)
121                         %Too close to middle, drive straight.\
122                         fail_factor = 0;
123                         middle_counter = middle_counter + 1;
124                         drive_car(car.speed, car.steer_straight,
                                0.7);
125                     end
126                     if(fail_factor >= 3)
127
128                         fail_factor = 0;
129                         if(drive_counter >= 3)
130                             switch(pass)
131                                 case 2
132                                     disp('Case 2');
133                                     %This will hopefully never
                                            happen. In
134                                     %this case just trow an error.
                                            Maybe
```

```
135                                     %later drive backwards and
                                            reset position?
136                                     error('Car outside microphone
                                            range');
137                                 case 3
138                                     disp('Case 3');
139                                     drive_counter = 100;
140                                 case 4
141                                     drive_counter = drive_counter +
                                            1;
142                                     drive_car(car.speed, car.
                                            steer_straight, 0.2);
143                                     if(drive_counter > 10)
144                                         car.did_last_turn = true;
145                                         localize_5ch(TDOA_data,
                                                1000);
146                                         car.did_last_turn = false;
147                                     end
148                                 case 5
149                                     disp('Case 5');
150
151                                 case 6
152                                     disp('Case 6');
153                                     localize_5ch(TDOA_data, 1000);
154                                     state = States.Drive;
155                                     Orientation; %function without
                                            nonglobal arguments
156                                 case 7
157                                     disp('Case 7');
158                                     localize_5ch(TDOA_data, 1000);
159                                     state = States.Drive;
160                                     Orientation; %function without
                                            nonglobal arguments
161
162
163                             end
164                         else
165                             drive_counter = drive_counter + 1;
166                             drive_car(car.speed, car.steer_straight
                                    , 0.2);
167                         end
168                     end
169                 end
170         end
171     end
172
173
```

```
174     function timer_stopFcn(timerObj, timerEvent)
175         disp('Timer is gestopt');
176         delete(timerObj);
177     end
178     function timer_error
179         disp('Error');
180         drive(150, 150);
181         EPOCommunications('transmit','A1');
182     end
183 end
```

A.3.2 drive_car.m

A.3.3 states.m

```
1  classdef States
2     enumeration
3        VoltageMeasure, Drive, Sample
4     end
5  end
```

A.3.4 status_update.m

```
1  function status_update
2  global car;
3  inc_data = EPOCommunications('transmit', 'S');
4  raw = strsplit(inc_data, {'D', 'U', 'A', 'udio ', '\n', ' '});
5  data = str2double(raw);
6  car.LeftSensor = data(4);
7  car.RightSensor = data(5);
8  car.voltage = data(6) / 1000;
9  car.AudioStatus = data(7);
10 end
```

A.4 Final Challenge