EPO4 Final report

Group A2
20/06/2024

EE2L21

Robin Appel (5454042), Mathieu Demeestere (5693691), Gijs Groen (5852005), Alessio Solter (5413567) & Wiktor Tomanek (5625173)

Delft University of Technology

TU Delft

EPO4 Final report

by

Student Name	Student Number
Robin Appel	5454042
Mathieu Demeestere	5693691
Gijs Groen	5852005
Alessio Solter	5413567
Wiktor Tomanek	5625173

Instructor: A. van der Veen and B. Abdikivanani
Project Duration: April, 2024 - June, 2024
Faculty: Faculty of Electrical Engineering, Delft

TU Delft

Contents

1 Introduction **1**

2 Communicating with KITT **3**
2.1 Logging the data . . . 3
2.2 Motor Control Functions . . . 3
2.3 Beacon . . . 4
2.4 Distance sensors reading and calibration . . . 5
2.4.1 Measurements data analysis . . . 5
2.4.2 Measurements analysis . . . 7
2.5 Microphones . . . 7
2.5.1 Connecting to the microphones . . . 7
2.5.2 Reading from the microphones . . . 7

3 Locating KITT using audio communication **8**
3.1 Specifications . . . 8
3.2 Analysis . . . 9
3.3 Design and testing . . . 10
3.3.1 Channel estimation . . . 10
3.3.2 Localization . . . 11
3.3.3 Localization in the field . . . 13
3.3.4 Testing . . . 14
3.4 Results . . . 14

4 Creating the car model **16**
4.1 Velocity model . . . 16
4.1.1 Velocity Equations . . . 16
4.1.2 Drag force: F_d . . . 17
4.2 Steering model . . . 17
4.3 Position model . . . 18
4.4 Tuning the model . . . 18
4.4.1 Acceleration force: F_a . . . 18
4.4.2 Viscous drag coefficient: b . . . 19
4.4.3 Steering angle . . . 19

5 Control **21**
5.1 PID Control . . . 21
5.1.1 PID Template . . . 21
5.1.2 Distance Error . . . 22
5.1.3 Angle Error . . . 22
5.1.4 Tuning the controllers . . . 23
5.2 Pure Pursuit . . . 24

6 System integration **26**
6.1 GUI . . . 27

7 Conclusion **29**

8 Discussion **30**

A Python code and results **32**
A.1 KITT class . . . 32
A.1.1 Joystick Remote Control . . . 36

A.1.2 Sensor Utility . . . 36
A.2 Audio class . . . 37
A.3 TDOA class . . . 38
A.3.1 Channel Estimation (ch3) . . . 38
A.3.2 Localization . . . 39
A.3.3 Localization in the field (Whole TDOA class) . . . 39
A.4 GoldCodeGenerator . . . 41
A.5 PID Controllers . . . 42
A.6 Pure Pursuit . . . 44
A.7 Integrated Code . . . 45
A.7.1 Challenge A - Driving to a point on the field . . . 45
A.7.2 Challenge B - Driving from to a point, then to a second point . . . 49

1 Introduction

This report describes the EE2L21 EPO-4 project ”KITT: autonomous driving challenge”. The aim of the project is to take a remotely operated toy car and make it drive autonomously. The car ”KITT” can be connected via Bluetooth with a base station (PC or laptop), and has a pair of distance sensors as well as an audio beacon on top. The base station takes the data from the sensors and microphones that detect the audio beacon in order to perform location, trajectory, and collision avoidance calculations, such that KITT can drive autonomously from point A to B while avoiding obstacles along the way.

Since the assignment is rather complex, it is split into several sub-blocks:

- Communicating with KITT
- Localization of KITT using microphones
- Virtual car model
- Tracking location and controlling KITT

An overview of the system is shown in Figure 1.1. The virtual car model predicts the position of KITT and tracks the state in order to predict the position without using microphones, which assists the localization algorithm in finding the location as precisely as possible.

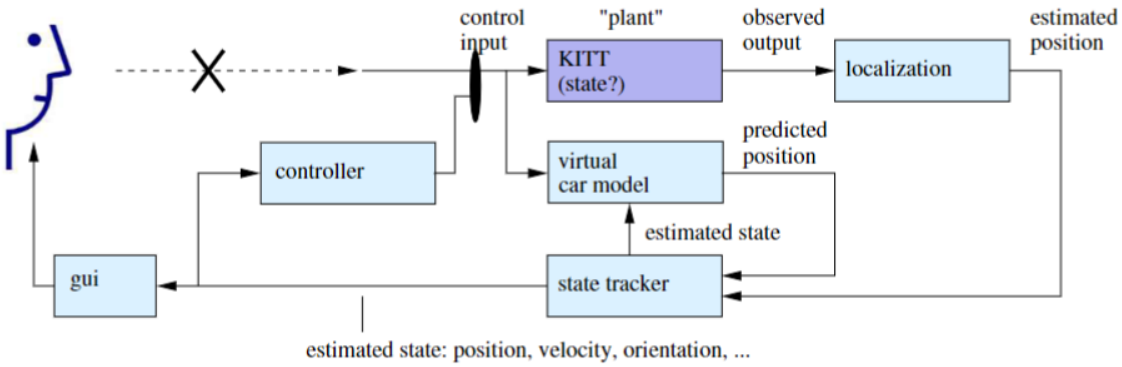

Figure 1.1: Block diagram showing an overview of the system from the Manual [3]

Communicating with KITT consists of setting up a stable RF connection using Bluetooth as well as reading and using the data from the microphones along the field and the sensor data for anti-collision detection. The Bluetooth connection allows the base station to send simple commands to a microcontroller on KITT, which will turn those commands into motor control signals. Further details about the communication with KITT can be found in Chapter 2.

Localization of KITT is done by emitting a repetitive audio signal from the beacon on top of KITT. This signal is picked up by the microphones along the field and deconvolved with the reference signal. Since the recieve-time will differ depending on the distance from the beacon to the microphones, the location of KITT can be determined. Further derivation about the localization of KITT can be found in Chapter 3.

The virtual car model is used to model the behavior of KITT in order to predict its velocity, position, and direction. It consists of an integration between a velocity model and a steering model. The velocity model takes into account all the forces present on KITT while driving and uses this information to determine the velocity. Furthermore, the steering model determines the direction of KITT. If the car makes a turn, it will drive in a circle with a radius R. The relation between the steering angle ϕ of the wheels and the radius of the circle R can then be used to find the direction of KITT. Further details about the virtual car model can be found in Chapter 4.

The goal is ultimately to have KITT drive autonomously from point A to point B while avoiding obstacles it might come across. In order to achieve this, the current location of KITT and possible obstacles should be properly stored and updated when necessary. This will allow for control regarding the direction, steering angle, and leftover distance to be traveled, such that KITT will accurately arrive at its destination while avoiding possible obstacles along the way. Further explanation and derivation about the location tracking and the control of KITT can be found in Chapter 5.

Finally, all the sub-blocks are integrated and combined into a whole. The localization algorithm will work together with the control system to drive KITT from a starting point along a path to the specified endpoint, while the distance sensors allow for the detection and avoidance of potential obstacles. Further explanation and results from the testing can be found in Chapter 6.

The project is divided into 4 graded challenges, namely:

- Challenge A : Driving to a point
- Challenge B : Driving to a point, and continuing to a second point
- Challenge C : Driving to a point with obstacles on the field
- Challenge D : Driving to a point with obstacles and an inactive beacon car on the field.

Results from these challenges are discussed in chapter 6, the conclusion, and the discussion.

2

Communicating with KITT

The initial objectives explained in this section were to communicate with the car, control it, gather information from its sensors reliably, and learn how to capture the audio recordings from the microphone setup that was used for locating the car.

In addition to the implementation of the solutions to the main objectives, some optional extensions were added. That included for example being able to log the data from the sensors into variables and then into a CSV file.

As mentioned in the introduction at the beginning of this project an RC car was provided. It can receive commands from a Virtual Bluetooth Serial Port. The base station sends its requests, which can be either for motor commands or requests for data. KITT responds accordingly by driving the motors or sending data back. To implement this functionality, a Python function was created that initializes all of the necessary variables including the ones for the data logging, checks for the type of operating system to ensure that our code can run on both Linux and Windows, and lastly opens and initializes the comma-separated values (CSV) log file.

It was noted that sometimes the Bluetooth connection does not get established immediately, so the initialization routine tries to connect to the car 5 times.

2.1. Logging the data

To log the data a function ”situation report” (sitrep) was implemented. It sends the status request to the car and either prints or stores the status response. The car responds only with a string of text, so some additional code was required to split the text into variables prepared for logging.

The final implementation of this functionality can be seen in the appendix A.1 as functions print_status, which prints the data to the console, and log_status, which logs the data to a CSV file, which can be used for further testing and development as seen in the chapter chapter 4.

2.2. Motor Control Functions

It was noted that the KITT car system had two ways of affecting its position. A drive motor that would push it either forward or backward based on a given PWM signal that is sent to the driver board, or a servo motor that would turn the front wheels of KITT based on a given PWM signal. The PWM settings were between 135 and 165 for the drive motor, where 135 means full speed back and 165 full speed ahead. For the steering motors, the PWM is between 100 and 200 with 100 full right and 200 full left.

Both had a midpoint of 150 which was the neutral position.

These values need to be controlled using the base station, and as such that is possible with a serial command. Sending a serial command of 'Mxxx$\backslash n$' will set the drive motor to xxx PWM setting, and sending a serial command of 'Dxxx$\backslash n$ will set the steering motor to xxx PWM setting. This functionality was implemented in the set_speed and set_angle functions, which will send the necessary commands and only take in the PWM setting.

To test the validity of these functions two programs were implemented, as seen in the appendix A.1.1. The first one acts as a remote control using the keyboard. It needs to be run in a loop and will check the WASD or arrow keys using the Python keyboard library, and it returns the corresponding speed PWM setting or direction PWM setting. This then needs to be passed into the set functions, which will send it on to the car. To not overload the serial port it is recommended that a delay of at least 0.1 seconds be between each call of the updateDirectionKeyboard function together with its corresponding set functions.

The second function implements joystick control in Linux. updateDirectionStick was used as a learning platform for multiprocessing, a technique that was desired for its ability to accelerate TDOA calculations and make it possible to run them while driving the car. Eventually, it was not used due to the time constraints of the project. The code in appendix A.1.1 is also an example of how to run KITT code and how most of the applications are implemented.

2.3. Beacon

The final subsystem that had to be controlled on the car was the beacon. This is an integral part of this package since it allowed the car to be detected on the field. The first step was to initialize the beacon parameters in the car, namely carrier frequency, bit frequency, repetition count, and beacon code. These need to be set per the system response to make the localization easier.

Carrier frequency is set using the 'Fxxx $\backslash n$' serial command. It was attempted to define this value empirically by running a loop of carrier frequencies through the beacon and plotting the exact system response over multiple frequencies. However, due to a bug in the KITT source code, which only allowed the beacon frequency to be set once per reset, this was not possible. Instead, the value was defined by looking at the microphone response charts and listening to what carrier frequency sounded the best. The carrier frequency was determined to best be set at 5000 Hz.

The bit frequency and repetition count go hand in hand to define the repetition frequency. Again these values were determined by ear and set by a serial command, namely 'Bxxx $\backslash n$' for the bit frequency and 'Rxxx $\backslash n$' for repetition count. Bit frequency was set to 5000, and repetition count was set to the highest value of 2500, this made the transmission time short enough to be able to do faster TDOA, and the repeat frequency high enough so that the measuring interval of 1 second would contain at least 2 transmissions.

The code was determined using a gold code generator. This code was adapted from Dasika Sri Bhuvana Vaishnavi [2] and can be found in appendix A.4. This code returns a binary sequence that is theoretically a gold code. However, later consideration discussed that the energy of this gold code is probably not correct since this function assumes +1 and -1 energy content and the beacon only allows 0 and 1. It was still decided to continue with this code however and as such the code used in the end was 0xEB79D549 and set using the serial command 'Cxxx $\backslash n$'.

Carrier Frequency	5000 [Hz]
Bit Frequency	5000 [Hz]
repetition count	2500
code	0xEB79D549

Table 2.1: Beacon parameters

2.4. Distance sensors reading and calibration

To read the distance sensors and perform measurements on them a separate minimalist implementation of the KITT class and its application was implemented in the sensors_claibration_utility.py, as seen in the appendix A.1.2. This program contains only the necessary functions to read and print the sensor data to the standard output. The last line in the code clears the standard output buffer, so the new data always appears on the top of the screen.

This minimalist approach allowed for the testing of the maximum speed at which the data could be pulled from the car through the Bluetooth interface. The idea was to reduce the delay in line 35 of the sensors_claibration_utility.py until the system breaks, so the data request is sent faster than the previous response is being received. The delay was reduced to 8ms and the output stopped appearing. It was hard to decide if that was a data request issue, or the terminal emulator (KDE Konsole) not being able to clear the output fast enough, however, what can be seen from that is the fact that most of the delay in the main routine will be caused by other Python function and the car's movement control, which was estimated to be close to 250ms based on the steps in the data shown in the figure <ref> and attached in the appendix <ref>. Consequently, with multithreading, the sensor data can potentially be received very fast.

2.4.1. Measurements data analysis

A set of static and dynamic measurements was performed to characterize the sensors' response. The first one was simply placing the car on the ground, putting an obstacle in front of it, and comparing the distance recorded by the sensors with the one measured. The observations are summarized in the table 2.2. It can be noted that the sensors seem to be accurate to one centimeter which is more than required for obstacle detection.

Real	Measured	Difference
20	20	0
50	49	1
100	100	0
200	200	0
300	300	0

Table 2.2: Distance measurements

The second measurement was estimating the maximum range of the sensors. This was done by putting the car at the front of the wall and moving it manually as far back as possible until the sensors stopped increasing the reported value. The maximum stable recorded value was 713 cm which is more than the size of the track for the car, which means that the maximum distance measurement is also not a limitation for collision detection.

Lastly, an estimation of the beam angle was performed. This was done by placing the car on the ground and marking a parallel line extending its axis. This line was used to mark the point at the edge of the view cone of the given sensor. Those points were found by moving an obstacle outwards the view cone until the sensors started reporting an overflow value instead of the accurate measurement.

The results of the measurements can be found in the table 2.3 below, where USRL, USRR, USLL, and USLR are the distances of the edge points from the central line. Accordingly right sensor's left edge, the right sensor's right edge, the left sensor's left edge, and the left sensor's right edge. Distance (X) is the distance between the sensor and the points of measurement alongside the line. The angles were calculated from the distances using the formulas 2.1. 17 and 17.5 in the formulas represent the distance from the center line of the car to the location of the sensor on the beam. The average result equals to 15.9°which is relatively narrow.

Distance (X)	USRL	USRR	USLL	USLR	Angle Left	Angle Right
300	-53	39.5	-60.5	27.5	16.7°	17.4°
100	-3	25.5	-24	-1	13°	16.1°
50	5	21.5	-22	-10	13.7°	18.6°

Table 2.3: Angle measurements

$$\alpha_L = \arctan\left(\frac{|USLL| - 17.5}{X}\right) + \arctan\left(\frac{USLR + 17.5}{X}\right) \tag{2.1}$$

$$\alpha_R = \arctan\left(\frac{USRR - 17}{X}\right) + \arctan\left(\frac{|USRL| + 17}{X}\right) \tag{2.2}$$

Figure 2.1: Formulas for calculating the view angles of the sensors.

The above measurements were part of Task 2 in section 4.3.1 of the manual[3], thus they were the static measurements. Task 3 focused on the dynamic measurements. To characterize the dynamic behavior of the sensors we recorded the data while driving away from the wall. Figure 2.2 shows the plotted data for a specific period and a zoomed-in fragment of those data.

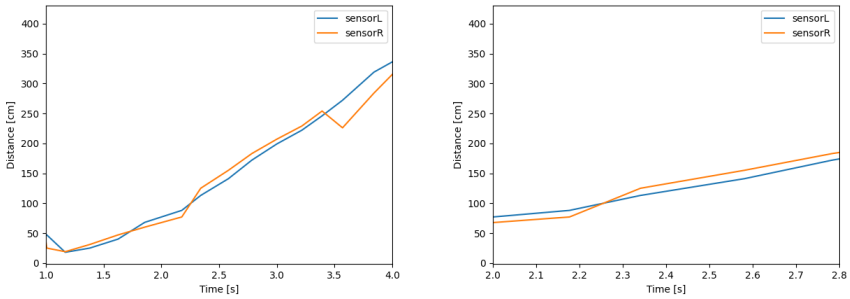

Figure 2.2: Distance measurements while driving from the wall

It can be noticed that the data is not completely smooth due to a delay caused by the Python code and the time it takes for the car to respond with a new data value. In total, the delays between the new data requests equal to about 250ms, which can be seen in the timestamps in the data in the appendix <ref>. All the time measurements were performed using Python's time.monotonic function, as seen in the line <ref> in the code in the appendix <ref>. There are also some discrepancies between the right and the left sensors caused by the angles at which the sensors see the obstacle, their viewing cones, and the fact that they measure alternatingly.

2.4.2. Measurements analysis

In conclusion, the sensors have high accuracy of the measurement but their viewing cone is limited, which has to be taken into account for the collision avoidance. Their response is also very fast, however, due to the delay between every sample point and changes in the distance caused by the change of the angle between the surface of the obstacle and the sensor, the velocity estimation based on their data will not be accurate. Sometimes sensors also ”lose” the emitted ultrasonic wave, or receive the bounced wave, and jump to a very high value, which has to be filtered, otherwise, the derivative will be very high.

2.5. Microphones

In this project, there are 5 microphones located around a field, and connected to the PreSonus Audiobox 1818VSL for detecting the position of the car. The audio recorded by the microphones needs to be used to carry out the TDOA described in the next chapter. It is important that the output of this class is formatted so that it can be used by the TDOA class.

The Pyaudio library is used for reading the information from the microphones together with a class that was made to handle the audio data processing needed before the TDOA class receives it.

This section will explain how this class works, how to use its basic functionality and the nice-to-haves that this package includes. The code for this class can be found in appendix A.2.

2.5.1. Connecting to the microphones

Connecting to the microphones is done by connecting the USB cable from the soundcard, then calling the class and storing it as an object. The ’init’ function initializes the microphones and holds internally all handlers needed for operation. If the microphones are not connected, the program will retry 5 times before returning an error and exiting the program.

The code needs to find the correct microphone. This is done with the detect_microphone-function. It searches the audio device that contains ’Microphone’ and ’Audiobox 1818’ and uses this device.

2.5.2. Reading from the microphones

Reading from the microphones is done by using the library Pyaudio. This can be done in two ways, by reading the steam directly and by using a callback function. Both were tried to implement in this class, but using a callback function turned out to be a bit difficult and not necessary for reading the audio. The report will only describe the first way mentioned.

By calling Audio(), the class defaults to a non-callback version. Reading the microphones is then done by calling the sample function with the amount of samples wanted. This will then return N * 5 samples for the 5 microphones at the default sampling frequency of 44100 [Hz]. This will however block the code until all the data has been read. This is undesirable since that time could be used to calculate TDOA from the old sample, so multiprocessing could be used to accelerate this code in the future. The code below shows an example of how to sample all 5 microphones for 2 seconds.

Listing 2.1: Example of how to use the non-callback microphones

```
1 from Audio import Audio
2 mics = Audio()
3 N = 44100 * 2
4 samples = mics.sample(N)
```

3

Locating KITT using audio communication

In this chapter, the localization of KITT will be described. Real-time recordings are used in combination with 5 microphones to localize the car. The microphones along the field record the beacon signal from the car. Afterwards, deconvolution is used to find the channel estimate of the recordings and the highest peak is selected. The peaks from the different recordings are compared to each other, which results in a relative delay. Finally, this relative delay results in a time delay in order to locate the car.

3.1. Specifications

The objective of the localization part of the project is to be able to locate the car in the field. The car is sending out a beacon signal. This signal is recorded by the microphones that are on the corners of the field and in the middle of one side. Due to the difference in distances between the car that sends out the beacon signal and the five microphones, the microphones record the signal at different times. By using the speed of sound, the relative difference in distances and eventually the coordinates of the car in the field can be calculated.

The field is 4.60 by 4.60 meters. Four microphones stand on the corners of the field. One microphone stands in the middle of one side of the field at a different height. This can be seen in figure 3.1. The height of microphones one to four is 0.5 meters. The height of microphone 5 is 0.8 meters. The x-axis is defined from microphones 1 to 4 and the y-axis from microphones one to two. Coordinates are to be interpreted as (x,y). In table 3.1 all the locations of the microphones can be found.

To develop the code for the localization seven pre-made recordings are given. For these pre-made recordings also a high-quality reference recording is given and the coordinates of these recordings are given. Using these recordings the code for the localization can be made and next the code can be tested using the car that sends out a beacon signal in the field. The last step is to implement the code for the localization with the other parts of the project. This will be described in chapter 6. The microphones in the pre-made recordings are placed on a field of 4.80 by 4.80 meters instead of 4.60 by 4.60 meters. The locations of the pre-made recordings can be found in table 3.2.

Microphone	**x** (cm)	**y** (cm)	**z** (cm)
1	0	0	50
2	0	460	50
3	460	460	50
4	460	0	50
5	0	230	80

Table 3.1: Microphone locations

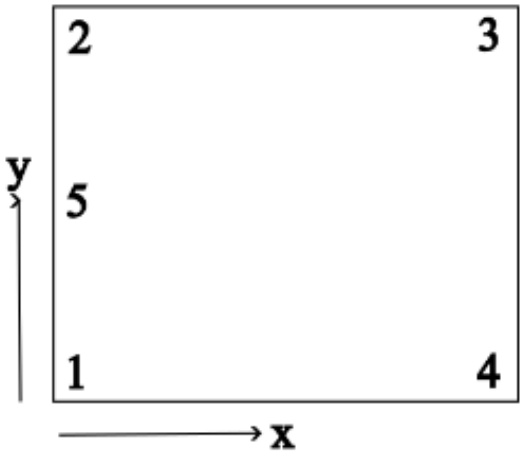

Figure 3.1: Microphone location and definition of axes

Recording	x (cm)	y(cm)
1	64	40
2	82	399
3	109	76
4	143	296
5	150	185
6	178	439
7	232	275

Table 3.2: Locations of the pre-made recordings

3.2. Analysis

First, a reference recording of the audio beacon has been made by placing the beacon directly underneath a microphone, which results in signal $x[n]$. Afterward, the car is placed somewhere inside the field with the audio beacon turned on. The microphones along the field will record this signal and store it, which results in 5 different measurements stored inside signal $y[n]$. Given the audio signal of the beacon $x[n]$ and the measured signals at the microphones $y[n]$, the audio channel h[n] can be found using via the convolution $y[n] = x[n] * h[n]$.

In this case, a deconvolution is required involving the Fast Fourier Transform (FFT) in the frequency domain, which is more efficient than a convolution in the time domain: a convolution in the time domain results in a pointwise multiplication in frequency, thus for a deconvolution in the frequency domain, only a pointwise division is required followed by an Inverse FFT to receive the time domain channel impulse response h[n]. The result is a channel estimate for the distance to each microphone, but since the transmission time is unknown, only relative propagation delays are obtained. By taking the difference between microphone delays, the Time Difference of Arrival (TDOA), as an amount of samples for each microphone pair can be obtained, removing the unknown transmit time.

Finally, the location can be obtained from the TDOA. The matrix calculation to find the location can be seen below in Equation 3.1. $\mathbf{x} = [x, y]^T$ is the location of the car, $\mathbf{x}_1$ to $\mathbf{x}_5$ are the locations of the microphones and τ_{12} until τ_{15} are the TDOA between the microphones, where τ_{12} is the TDOA between microphone 1 and 2. This matrix equation can be solved by using a pseudo-inverse in Python in order to obtain the location **x** and d_1, the distance from microphone 1 to the car.

$$\begin{bmatrix} \mathbf{x}_1 - \mathbf{x}_2 & \tau_{12} \\ \mathbf{x}_1 - \mathbf{x}_3 & \tau_{13} \\ \mathbf{x}_1 - \mathbf{x}_4 & \tau_{14} \\ \mathbf{x}_1 - \mathbf{x}_5 & \tau_{15} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ d_1 \end{bmatrix} = \begin{bmatrix} 0.5(\mathbf{x}_1^2 - \mathbf{x}_2^2 + \tau_{12}^2) \\ 0.5(\mathbf{x}_1^2 - \mathbf{x}_3^2 + \tau_{13}^2) \\ 0.5(\mathbf{x}_1^2 - \mathbf{x}_4^2 + \tau_{14}^2) \\ 0.5(\mathbf{x}_1^2 - \mathbf{x}_5^2 + \tau_{15}^2) \end{bmatrix} \tag{3.1}$$

3.3. Design and testing

The design of the localization of the KITT can be divided into three steps. First, a channel estimating algorithm was designed. Secondly, the code to find the coordinates for the pre-made recordings was made. And lastly, this code was adjusted and improved to be able to locate the car in the field. After each of the three steps, the testing of that part is shown.

3.3.1. Channel estimation

As described above in chapter 3.2, Analysis, the channel estimation is made by using the FFT. In Python, the FFT can be made by using the function scipy.ftt from the library SciPy. To do the pointwise division in the frequency domain, $x[n]$ and $y[n]$ need to be of the same length. This is ensured by adding zeros at the end of $x[n]$ to make it of the same length as $y[n]$. Then the FFTs are taken of $x[n]$ and $y[n]$, the deconvolution is carried out and $H[f]$ is obtained. All values of $H[f]$ for which the absolute value of $X[f]$ is less than a threshold value $epsi$ times the maximum absolute value of $X[f]$ are set to zero to avoid blow-ups during the inversion. Next, the Inverse FFT is obtained to get $h[n]$.
The code for this Channel Estimation algorithm, called ch3, can be found in appendix A.3.1. This algorithm was already designed in an earlier course.

Testing

The testing of the channel estimation function was already done in an earlier course. To show the working of the function the channel estimation for the first microphone of the pre-made recording at (64,40) has been plotted, as can be seen in figure 3.2. The plot clearly shows a peak where the beacon signal was recorded by the microphone. This peak can be used, together with the peaks that are recorded by the other microphones to calculate the coordinates of the car.

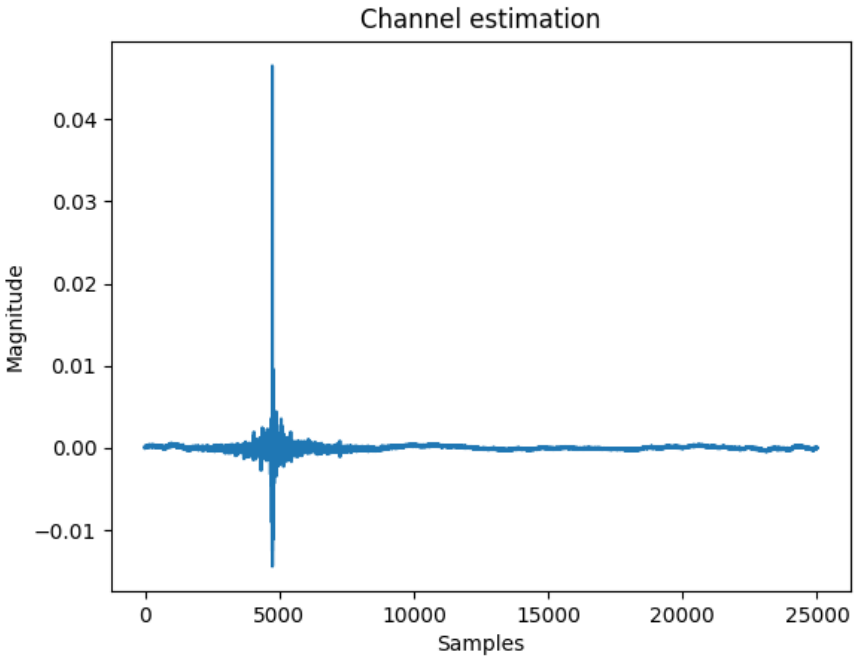

Figure 3.2: Channel estimation microphone 1 at (64,40)

3.3.2. Localization

The localization of the car is done by using TDOA (time difference of arrival). First of all some variables are defined, like the threshold value $epsi$ and the speed of sound. Then the $x[n]$- and $y[n]$-signals are cropped to contain only the last peak that has been sent before they are put into the channel estimation algorithm. The $y[n]$-signal contains the recordings of all five microphones. Each of the recordings are stored in a column of $y[n]$. So, $y[n]$ has five columns. For every column, a channel estimation is made. Using these channel estimations the time difference is calculated. The difference in peaks, calculated by taking the maximum value of the absolute channel estimate, between the first microphone and all other microphones, times the speed of sound and divided by the sample rate, gives the time difference of arrival. Then the locations of the microphones are specified. Lastly, to calculate the coordinates equation 3.1 is implemented in Python. A and C are defined and B is calculated using the least-squares method. As output only the coordinates are selected. The code for this part can be found in appendix A.3.2.

At first, only four microphones were used when developing the code for the localization. This turned out to give an error that was too large. By adding the fifth microphone the error in the coordinates was acceptable. As can be seen in the code all coordinates for the microphones are taken in 2D. This results in an error that gets bigger when the car gets closer to a microphone because the vertical difference relative to the horizontal difference gets larger. However, calculating everything as if it were in 2D gives results that are accurate enough for the car. The results and errors can be found in chapter 3.3.2.1.

Testing

At first, the code where only four microphones were used was tested using the pre-made recordings. This gave the following results, which are also depicted in figure 3.3:

Table 3.3: Localization results using TDOA with four microphones

Recording number	**True x** (cm)	**True y** (cm)	**Measured x** (cm)	**Measured y** (cm)	**Error** (cm)
Recording 1	64	40	49	17	27
Recording 2	82	399	97	376	27
Recording 3	109	76	90	47	34
Recording 4	143	296	171	273	37
Recording 5	150	185	96	145	67
Recording 6	178	439	189	406	35
Recording 7	232	275	239	244	32

The error was calculated using the following formula:

$$Error = \sqrt{(x_{true} - x_{measured})^2 + (y_{true} - y_{measured})^2} \tag{3.2}$$

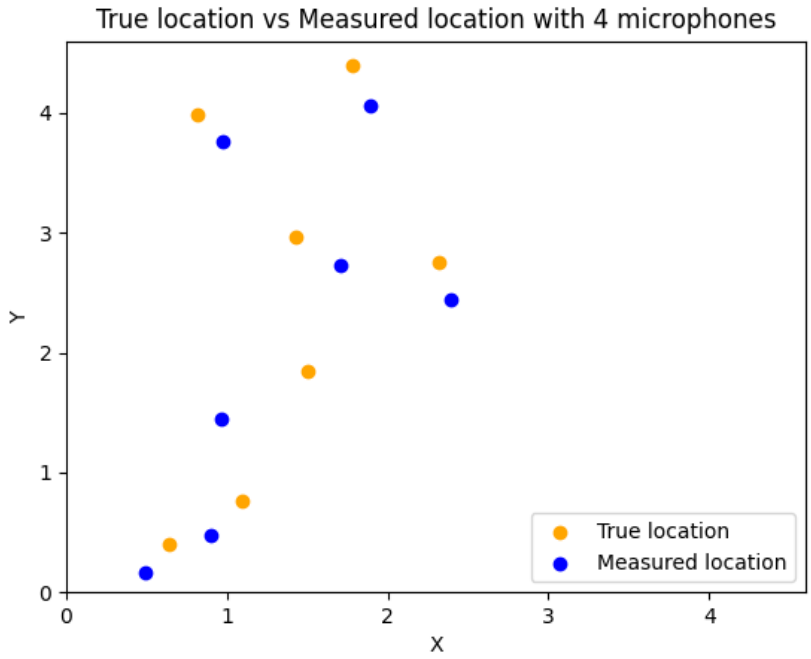

Figure 3.3: True coordinates compared to the Measured coordinates with four microphones

As the errors for the pre-made recordings were still too large and the pre-made recordings were presumably of a higher quality than the recordings that would be made with the car in the field, the code that uses only four microphones was not tested in the field.

Adding the fifth microphone resulted in a smaller error for most pre-made recordings. Only the error for the recordings where the car was placed relatively close to one of the microphones stayed practically the same. These testing results can be seen in table 3.4 and are shown in figure 3.4:

Table 3.4: Localization results using TDOA with five microphones

Recording number	**True x** (cm)	**True y** (cm)	**Measured x** (cm)	**Measured y** (cm)	**Error** (cm)
Recording 1	64	40	50	19	26
Recording 2	82	399	89	386	15
Recording 3	109	76	97	58	21
Recording 4	143	296	144	288	8
Recording 5	150	185	150	181	4
Recording 6	178	439	184	426	15
Recording 7	232	275	234	273	3

The error was calculated with equation 3.2.

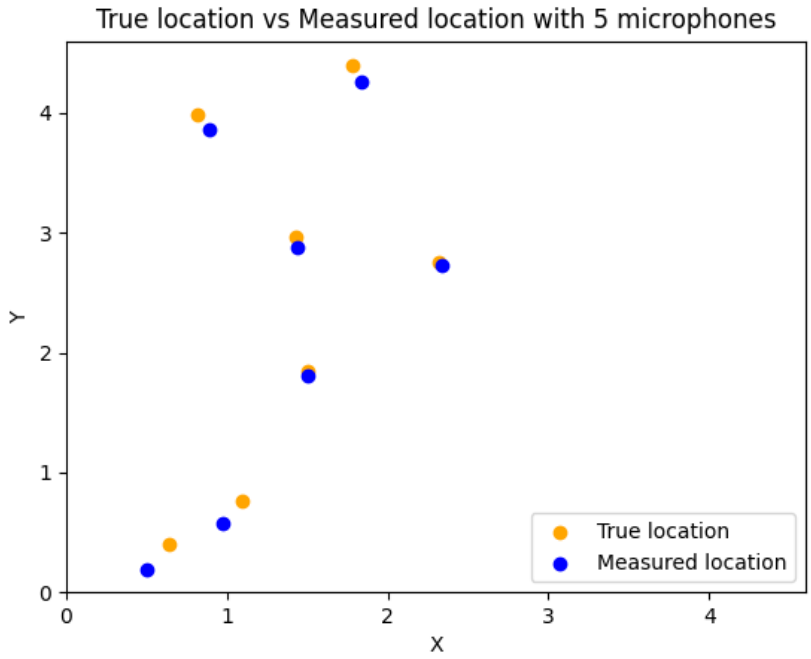

Figure 3.4: True coordinates compared to the Measured coordinates after adding the fifth microphone

As explained before, the error gets larger when the car is closer to a microphone. The addition of the fifth microphone drastically improved the measurements compared to the results while only using 4 microphones.

3.3.3. Localization in the field

To implement the code from above, so it can be used to actually locate the car, a few things need to be added. First, a function called tdoa_input is made to ensure that the localization function receives the right input. The recordings of the microphones are all in different arrays. To create one array with five columns, each for one microphone, the arrays are stacked. Initially, the code from above was used to locate the car in the field. This, however, gave errors that were too large to be able to accurately locate the car. Only in the quarter of the field that is closest to microphone one, the coordinates did not have a large error. To solve this problem, the first idea was to make a reference recording not only at microphone one and use this for the channel estimation of all microphones, but for every microphone, so five in total. It turned out that this did not solve the error, it only made it worse. The next idea was to determine which microphone was closest to the car and take this as reference microphone one in the matrix calculation. This means that every microphone gets its own matrix calculation. This could work because for microphone one the results already were accurate enough.

To implement this, the localization function was altered. The localization function was split up into three functions that are contained in one function. The first function contains the code to make the channel estimations with the ch3-function from section 3.3.1. It returns the five-channel estimates for the five microphones. Then a new function is made, the closest_mic-function. This function determines which of the first four microphones is closest to the car using the channel estimates. It makes an array of the indices of the maximum value of each of the four channel estimates. The fifth microphone is not taken into account for this, because the field is divided into four sections. The function then takes the indices of the minimal value in this newly made array. It adds one to these indices because the array starts with indices zero and the microphone numbers start at one. The third and last function in the localization part is the matrix calculation. For each of the four microphones, a matrix calculation is defined. In every calculation, the microphone that is closest is taken as a reference. So, the time differences of arrival for when microphone one is closest are defined from microphone one to microphone two, to microphone three, to microphone four and to microphone five. When microphone two, however, is closest, the time

differences of arrival are defined from microphone two to microphone one, to microphone three, to microphone four, and to microphone five. This is done for every one of the four microphones to which the car can be closest. Which of the now four matrix calculations has to be used is determined by using an if-statement that checks what the closest microphone was according to the closest_mic-function and then carries out the right matrix calculation. It returns the coordinates and the function around the three functions mentioned returns also those coordinates. The code for this section can be found in appendix A.3.3

3.3.4. Testing

For this last part of the design, testing has been done in the field. The results of this testing are also the results of this whole chapter. They can be found in the next section, section 3.4.

3.4. Results

Finally, the localization algorithm was linked to the microphones and testing could be done. For the tests, KITT has been placed in known locations spread across the field to get an accurate representation of the measurement quality. The results of these tests can be found in table 3.5 and figure 3.5. The measurements with direct integration with the microphones show similar results compared to testing with the given reference recordings; when the beacon gets close to a microphone, the error increases and when the beacon gets further away, the measured location gets closer to or is approximately the same as the true location.

Table 3.5: Localization results from direct integration with microphones

True x (cm)	**True y** (cm)	**Measured x** (cm)	**Measured y** (cm)	**Error** (cm)
50	80	38	64	20
110	30	95	17	28
20	105	11	89	18
120	130	112	124	10
60	230	48	210	23
90	300	93	289	11
40	440	63	419	31
125	415	114	406	14
200	360	205	351	10
230	230	231	229	1
290	180	287	176	5
220	45	209	47	11
420	430	454	459	44
380	400	393	421	24
320	275	317	271	5
445	300	468	312	26
415	45	452	24	42
400	100	423	89	25

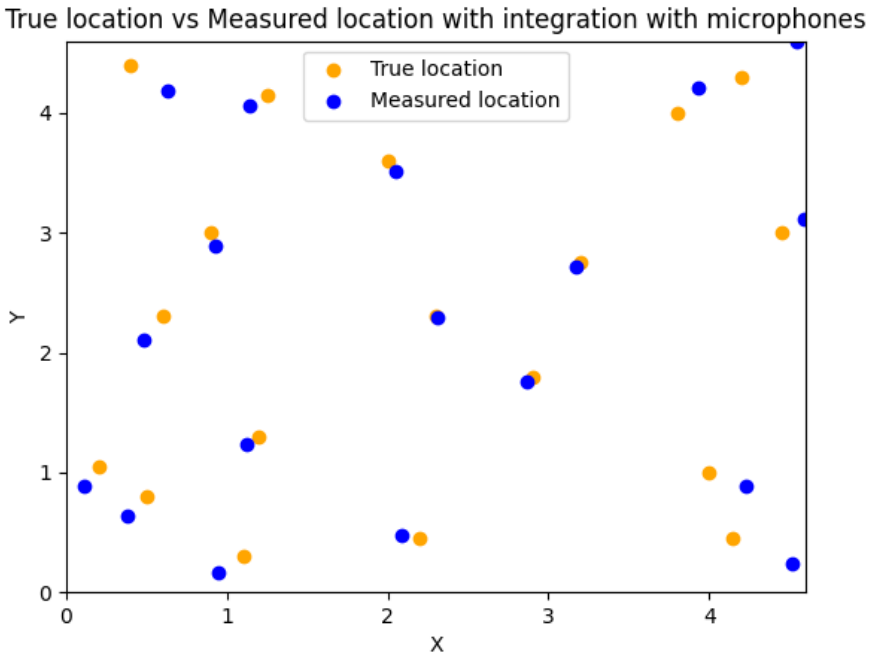

Figure 3.5: True locations vs Measured locations using direct integration with microphones

4

Creating the car model

Understanding KITT and its behavior is crucial for the understanding of the entire system in this project. Therefore, in this chapter, a virtual model of KITT that simulates its behavior is created. The model is split into two parts - velocity and steering. The decoupling of both components results in an approximate linearization of the system and therefore makes it significantly easier to work with.

4.1. Velocity model

The velocity model that is implemented is based on the addition of the different forces on the car. These forces are the drag force, the acceleration force, and the brake force. A schematic overview of the forces and the way they act on the car can be seen in figure 4.1.

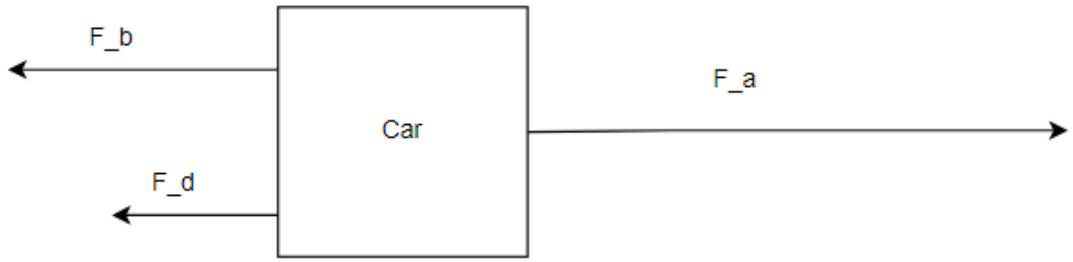

Figure 4.1: Forces on KITT

4.1.1. Velocity Equations

The velocity of an object can be derived from the sum of forces that act upon the body. In the case of KITT, two forces affect it in uniform rectilinear motion. Namely the motor Force [F_m] and the drag force [F_d]. With the acceleration and brake force being reduced to positive and negative motor force. The acceleration can be derived from Newton's second law of motion, namely:

$$F = ma \tag{4.1}$$

$$a = \frac{F}{m} \tag{4.2}$$

$$\frac{\delta v}{\delta t} = \frac{F}{m} \tag{4.3}$$

By discretely integrating this equation, the velocity can be derived. By integrating again the distance traveled is found.

$$v = \frac{F}{m} * t \tag{4.4}$$

This is implemented as:

$$v(t + \delta t) = v(t) + (\frac{F}{m} * \delta t) \tag{4.5}$$

with:

$$F = F_m + F_d \tag{4.6}$$

4.1.2. Drag force: F_d

The drag force, F_d, is comprised of two different forces: air friction and viscous friction. This can be calculated using the following formula:

$$F_d = b|v| + cv^2 \tag{4.7}$$

Where b is the viscous friction constant, $|v|$ is the magnitude of the velocity and c is the air drag coefficient. Typical values of streamlined bodies go as low as 0.04, as such, an approximation of c value was taken as $c = 0.1$, with a maximum velocity of $v = 2.1m/s$, this would lead to a force of 0.21N, this was decided to consider negligible which linearizes the drag model, significantly reducing the complexity of the model. Because of this, the formula for the drag force becomes:

$$F_d = b|v| \tag{4.8}$$

The derivation for b can be found in subsection 4.4.2

4.2. Steering model

The steering model for KITT is more complex than the velocity model. To begin we must define the steering angle, ϕ. This is the angle between the front axis and the wheels. If the car where to turn in a circle then the radius, R, of this circle would be dependent on ϕ. Additionally, the car can be better defined with the following parameters. L will be the distance between the front and rear axle. Thus, making the location of the rear axle $[0, 0]$ and the front axle $[L, 0]$. Finally, the angle that we wish to determine must be defined. This angle, θ, is the orientation of the car, or in other words the angle of the car relative to the positive x-axis. These paramaters can be visualized in the following figure (4.2): [3]

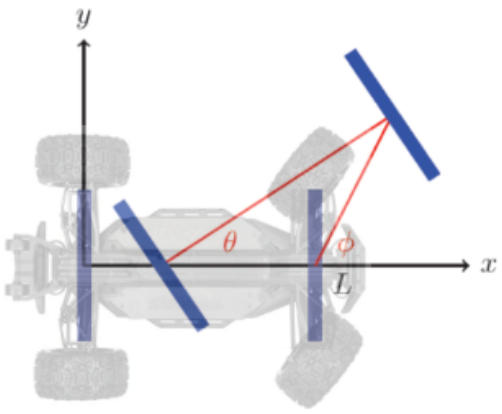

Figure 4.2: Visualization of steering parameters

There is a relationship between the angular velocity and the radius of a circle. Therefore, by combining these relationships it is possible to determine a relationship between the steering angle, ϕ, and the

radius of a circle, R. This can be seen in the following formula:

$$R = \frac{L}{tan(\phi)} \tag{4.9}$$

Additionally, it is known that the angular velocity, ω, is equivalent to the change in the angle over a certain time[1]. In other words the following can be derived:

$$\omega = \frac{v \cdot tan(\phi)}{L} = \frac{v}{R} \tag{4.10}$$

By utilizing both equations 4.9 and 4.10 it is possible to determine the orientation of the car. However, for this to be possible a relation between the steering input of the car and the steering angle must be determined. This can be found in subsection 4.4.3.

4.3. Position model

With both velocity and heading angle known, the position of the car can now be tracked. The position at time $t + \delta t$ can be found by adding a scaled direction vector to the current position vector.
To make this entire process simple a Python class was implemented. When this class is called, starting values are passed to it and initialized, usually to 0, however, variables like the position can be changed before runtime. This class contains the update function which will in order update velocity, heading and position, and return position and heading angle.

When the position update function is called it takes the current velocity and heading of the car, previously updated by the other two models. First, the function creates a unit-vector in direction θ (car heading) per Equation 4.11. Scaling this function by the distance traveled in the current time interval, one finds the travel vector, and adding this travel vector to the old position vector defines the new position vector. This ends in Equation 4.12.

$$\vec{D} = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \tag{4.11}$$

$$\vec{P}(t + \delta t) = \vec{P} + (\vec{D} * v * \delta t) \tag{4.12}$$

4.4. Tuning the model

4.4.1. Acceleration force: F_a

The motor does not take in pure Force values, as such the force needs to be measured in relation to the PWM setting, and over different PWM settings to account for non-linear behavior.
By Equation 4.13, when the velocity of the car doesn't change, the sum of forces is 0. This happens when the car is standing still, but also when the car is at maximum velocity. By measuring the maximum velocity and filling that into Equation 4.14, one can find the force exerted by the motor. For this experiment, $b = 4.16$ from subsection 4.4.2 was used.

$$a = \frac{F_m - b|v|}{m} \tag{4.13}$$

with a = 0

$$F_m = b|v| \tag{4.14}$$

Measuring the maximum speed was achieved by using a camera recording at 60 frames per second, and counting the frames that it took the car to cross a space of 1 meter. The test setup has been illustrated in Figure 4.3. From the number of frames, n, the velocity can be found by $v = \frac{60}{n}$. The final derived values are presented in Table 4.1. In Figure 4.4, a graph of the function that was derived from these measurements. The function implements linear interpolation between the measured points. This graph shows PWM on the Y-axis and the resulting force of that PWM on the X-axis.

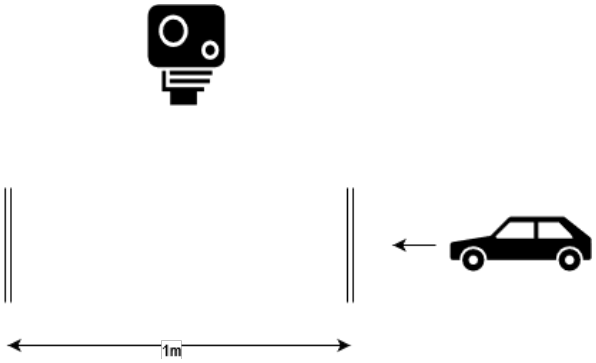

Figure 4.3: Velocity Measuring setup

PWM	Frames	velocity [m/s]	Force [N]
165	28	2.14	8.91
160	50	1.20	4.99
156	240	0.25	1.04
150	inf	0	0
144	92	0.65	-2.7
140	40	1.49	-6.24
135	25	2.39	-9.98

Table 4.1: Measured Force Values

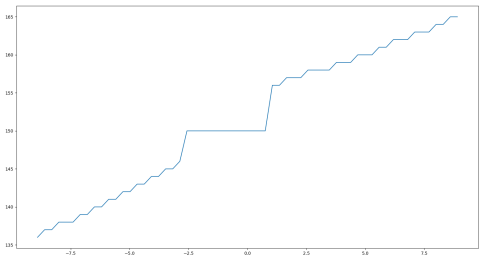

Figure 4.4: Graph with PWM on the y-axis and force on the x-axis, with interpolation

4.4.2. Viscous drag coefficient: b

The drag coefficient can be measured from the rolling distance of the car after a known velocity, by Equation 4.15. Rolling distance in this instance means the distance the car travels when no inputs are given.

$$x = v_o * t - \frac{1}{2}|v|bt^2 \tag{4.15}$$

b Was then found by inputting the v0 into the model and adjusting b until the rolling distance matched up.

PWM	V0 [m/s]	Rolling Distance [m]	b
165	2.72	2.62	4.16

4.4.3. Steering angle

In order to relate the steering angle of the car to the power-mode inputs of the car the car was made to drive in circles with varying different radii. From this, it was determined that the relationship between the

steering angle and the power mode was approximately linear. This can be seen in table 4.2. Because of the linearity of the system, Equation 4.16 was derived to calculate the PWM value from a given radian input.

$$PWM = \frac{900}{\pi} * (radians + \frac{\pi}{9}) + 100 \tag{4.16}$$

Steering input	Angle in degrees	Radius (m)
100	19.15	0.96
110	15.89	1.17
120	12.23	1.54
130	8.50	2.24
140	4.30	4.45
150	0	
160	4.5	4.26
170	8.56	2.23
175	10.94	1.73
180	13.75	1.37
185	16.21	1.15
190	18.05	1.03
w 200	22.12	0.82

Table 4.2: Steering input to angle

5

Control

In order to be able to actually make the car drive it must be constantly given new power settings. To determine which settings are required at each given moment a state tracking and control system is needed. This system should be able to determine what power settings the car needs to use in order to get to the desired destination. Two different methods where considered for this. These being a simple PID controller and a pure pursuit model. Both were worked out however, eventually, the PID was chosen to be used.

5.1. PID Control

For the control of the car, two PID control loops are running concurrently. The Force PID and the angle PID controller. Both determine their output based on input the same way, but the input error is different and what is done with the output is different.

The data for these controllers is stored in the PID class that can be seen in appendix A.5. It operates by calling the update function with the car state and the setpoint, which returns a motor force and a steering angle.

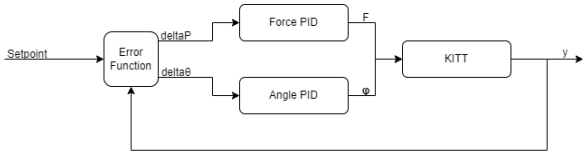

Figure 5.1: Illustration of the PID control loop

5.1.1. PID Template

A PID controller has 3 parts, as the name suggests, a proportional controller, an integral controller, and a derivative controller. This implementation of the PID controller uses discrete versions of the integral [Equation 5.1] and derivative [Equation 5.2] functions and then inputs those into the final equation, Equation 5.3 to determine the output of the system.

$$errorIntegral(t + \delta t) = errorIntegral(t) + (error(t + \delta t) * \delta t) \tag{5.1}$$

$$errorDerivative(t + \delta t) = \frac{error(t + \delta t) - error(t)}{\delta t} \tag{5.2}$$

$$Output = (Kp * error) + (Ki * errorIntegral) + (Kd * errorDerivative) \tag{5.3}$$

This function runs twice per loop, once for the angle and once for the force.

5.1.2. Distance Error

The distance error is quite simple to calculate using Pythagoras's theorem. First, find the vector that points from the car towards the set point. This vector is determined by Equation 5.4 and illustrated in Figure 5.2. The length of this vector is determined by Equation 5.5. This length is also the error for the Force PID controller.

$$\vec{E} = \vec{SP} - \vec{C} \tag{5.4}$$

$$|\vec{E}| = \sqrt{x^2 + y^2} \tag{5.5}$$

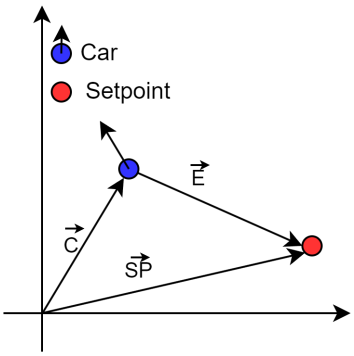

Figure 5.2: Illustration of the Error Vector $\vec{E}$

5.1.3. Angle Error

By far the hardest part of designing the PID controllers was determining a consistent and reliable angle error. It was decided to use the same error vector from Figure 5.2. Finding this angle from the positive X-axis is quite simple by using the arctan function described in Equation 5.6. The arctan function is nice to use in Python since the numpy function automatically deals with edge cases and divides by zero errors that may arise. Since θ, the car heading is given as an input, this can be used to derive to find the total error angle by Equation 5.7.

$$\phi = \arctan(\frac{y}{x}) \tag{5.6}$$

$$\Delta\theta = \phi - \theta \tag{5.7}$$

A couple of edge case preventions are implemented. If $\Delta\theta$ ends up outside of $[-\pi, \pi]$. Lastly, there is a function implemented to make KITT detect points that are behind it. This function detects angle errors that are greater than $\frac{\pi}{2}$, sets the distance error to negative, such that the distance PID starts sending negative forces such that KITT reverses back onto the set-point if it overshoots it. In theory, this function also allows the car to drive backward, however, in testing, this often did not work and the error was narrowed down to a bug in the car model that didn't steer correctly when negative velocities were applied. Due to limited time, this was never fixed.

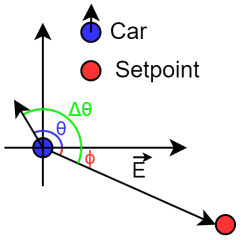

Figure 5.3: Angle derivation illustration

5.1.4. Tuning the controllers

The controllers were tuned based on a mixture of both real-life testing and simulation based on chapter 4. Kp was increased until the desired convergence speed was a little undershot, then Kd was increased to decrease convergence time and finally, Ki was increased until the model predicted low error. Ki was purposefully kept low since increasing it any further quickly introduced instability in both systems.

Force			Angle	
Kp	1.2		Kp	0.8
Ki	0.2		Ki	0.1
Kd	1.0		Kd	0.2

Table 5.1: PID Gains

One of the simulation tools for tuning is shown in Figure 5.4. This tool was designed to tune the force component of the PID loop. It shows the distance traveled over time, with the orange line being a 1m set-point. The behavior of both PID loops working together is modeled in Figure 5.5. In the top left, there is a top-down view of the field, the bottom two graphs are representations of the x and y values of the car, with the orange lines being the set-point, and the blue lines being the simulated position. Finally, the top right graph shows the Angle PID output, This is because this tool was mostly used to tune the angle controller and error function.

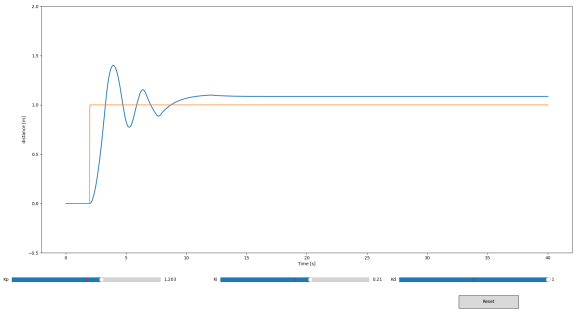

Figure 5.4: PID Force tuning Utility

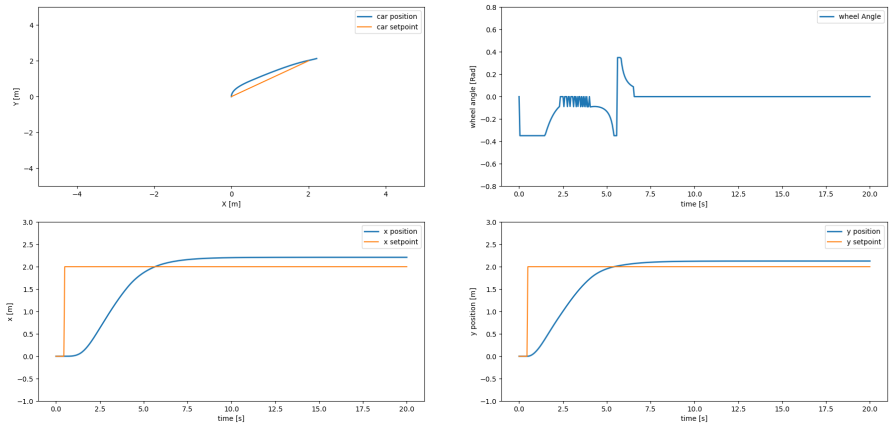

Figure 5.5: Behaviour of PID on the field

5.2. Pure Pursuit

Pure pursuit works by first drawing a line between the initial position of the car and the final point the car needs to go. Afterwards, a circle is drawn around the car with radius L_d. This is the look-ahead distance. This circle then creates intersections with the line that was drawn previously. These intersections then become target points, TP. Next, the angle, α, needs to be chosen such that the trajectory of the car gets to the target point, TP. This can be seen in figure 5.6. The code implementation can be seen in Appendix A.6

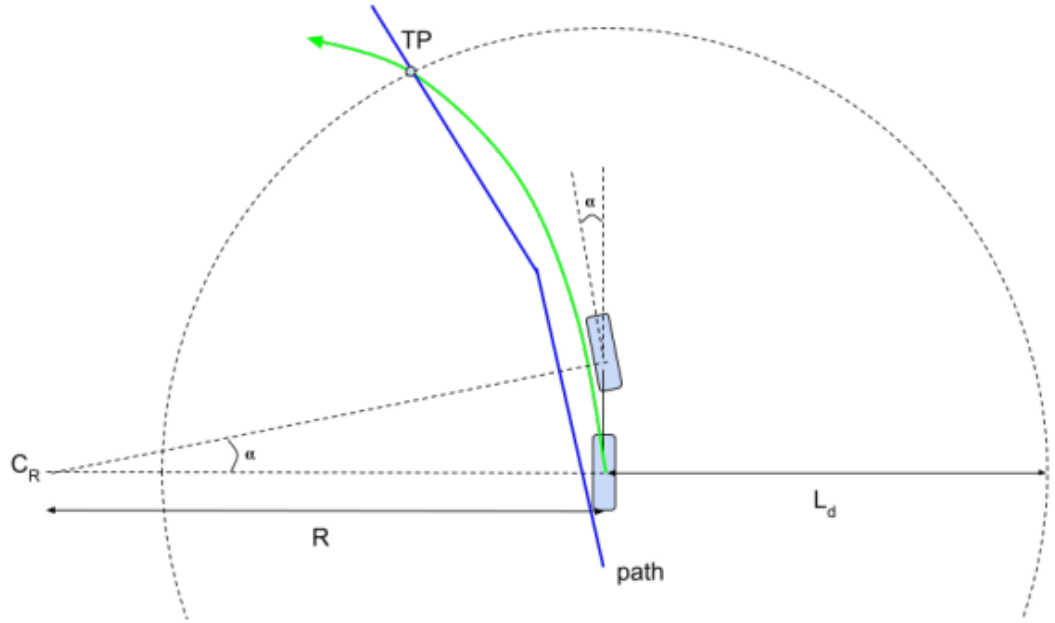

Figure 5.6: Pure Pursuit Model

In order to determine the steering angle, α, a triangle can be drawn between C_R, TP, and the center of the car. This can be seen in figure 5.7.

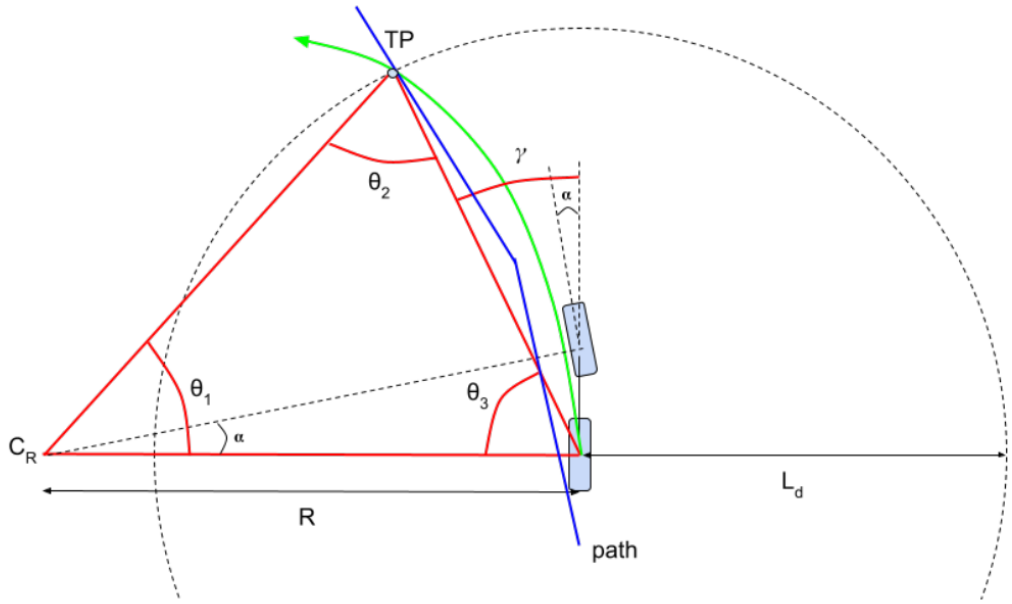

Figure 5.7: Pure Pursuit Model

The distance between C_R and TP is the same as the distance between C_R and the car. Therefore, the triangle is an isosceles triangle. Knowing this the following is true:

$$\theta_2 = \theta_3 = 90° - \gamma \tag{5.8}$$

$$180 = \theta_1 + \theta_2 + \theta + 3 = \theta_1 + (90° - \gamma) + (90° - \gamma) \tag{5.9}$$

Therefore, it can be said that $\theta_1 = 2\gamma$. Therefore, the law of sins gives:

$$\frac{L_d}{sin(\theta_1)} = \frac{R}{sin(\theta_2)} \tag{5.10}$$

$$\frac{L_d}{sin(2\gamma)} = \frac{R}{sin(90 - \gamma)} \tag{5.11}$$

Next, $sin(90-\gamma)$ can be replaced with $cos(\gamma)$, and $sin(2\gamma)$ with $2sin(\gamma)cos(\gamma)$. Thus giving the following:

$$R = \frac{L_d}{2sin(\gamma)} \tag{5.12}$$

Finally, this equation can be combined with equation 4.9, giving the final equation to be used in order to calculate the steering angle.

$$\alpha = arctan(\frac{2Lsin(\gamma)}{L_d}) \tag{5.13}$$

After implementing the pure pursuit algorithm it ended up not being very accurate. Therefore, it was chosen to use the PID controller instead.

6

System integration

Everything needed to be integrated and synchronized after all the modules were written and tested separately. A top-level file was designed and created for this, as seen in the appendix A.7.1. It consolidates the Python classes from chapter 2, chapter 3, chapter 4 and chapter 5. The pseudo-code seen in algorithm block 1 represents the implementation of this code for challenge A. The actual implementation can be found in the appendix A.7.1

$target \leftarrow input$;
$carState \leftarrow input$;
INIT PID, TDOA, KittModel, KITT;
while *True* **do**
  if $timeSinceLastUpdate < 1s$ **then**
    $carInput \leftarrow PID(carState, target)$;
    $KITT \leftarrow carInput$;
    $carState \leftarrow KittModel(carInput)$;
  end
  if $timeSinceLastUpdate > 1.5s$ **then**
    $carState \leftarrow TDOA()$;
    $timeSinceLastUpdate \leftarrow 0$;
  end
  if $carState$ *is close to target* **then**
    $carState \leftarrow TDOA()$;
    if $carState$ *is close to target* **then**
      $return \leftarrow carState$
    end
  end
end

Algorithm 1: Top level pseudo-code

At the beginning, the algorithm asks the user to input the target position and the initial position of the car. After that, it initializes all of the necessary variables and classes. Class initializations naturally cause the system to connect to the car via Bluetooth and to the microphones via the USB sound interface. After all of the initialization was done the system starts an infinite loop function that runs with a 100ms delay in every iteration. If less than 10 iterations of the loop pass (1 sec) then the car uses the KITTModel to predict its current position, however, the model is not ideal and will drift over time. That is why every 15 iterations (1.5 sec) the car stops and uses beacon localization to overwrite its current position.

It was noted during testing that the KITTModel is inaccurate for a motor force equal to 0 and causes

the simulated car to ”free wheel” forward as if there was very little stopping friction, which causes the predicted position to reach the final target vicinity before it happens to the real car. To mitigate that the KITTModel is not updated between 10 and 15 iterations. Lastly, if the car’s predicted position is close to the target the car stops and measures the actual position of the beacon, since the prediction might still have been based on the inaccurate simulation. If it is still the case that the car is within a predefined threshold of the target then the entire program stops.

Challenge B consists of driving to a point, and then to a second point, thus the chosen approach was to use the same algorithm as for challenge A, but twice. Additionally, a Graphical User Interface (GUI) was implemented for the convenience of inputting the data, as discussed in section 6.1. The full implementation of the challenge B can be found in the appendix A.7.2

6.1. GUI

For a convenient use of the program, two Graphical User Interfaces were developed. One of them was used to input the targets and the initial position for challenge B, as seen in fig 6.1. The second one was used to see the simulated position of the car, which was very helpful during the testing phase as it helped to discover the ”free-wheeling” problem of the car simulation when the force was equal to 0. The most probable reason for this was the fact that the frictions and drags are nonlinear, and quickly rise close to 0 force, while the simulation assumes a fixed constant value for them.

The simulation GUI can be seen in figure 6.2. A blue dot represents the car and the arrow represents the direction it is heading. The red dot represents the target. Both of these move and update during a run such that the behavior of the car and the behavior of the model can be compared. Figure 6.2 illustrates an example of this interface, with the car at $(0.3, 0.3)$ with heading $\theta = \frac{\pi}{2}$, and the target at $(2, 2)$.

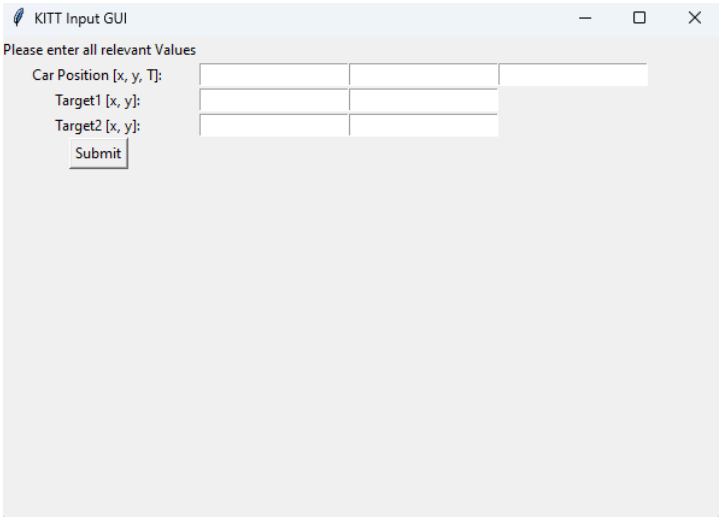

Figure 6.1: Input GUI

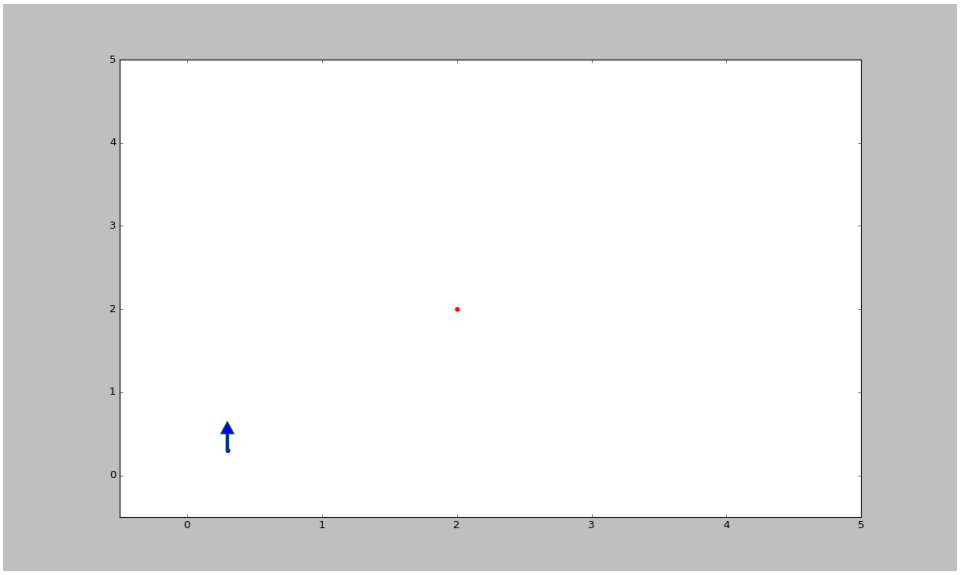

Figure 6.2: Example of the Run GUI

7

Conclusion

As described in the previous chapter, the car was able to drive to a location in the field. The car stopped at about 10 cm from the exact coordinates it was supposed to drive to. The TDOA algorithm was able to locate the car accurately enough. The car model describing the behavior of the car and the controllers that enabled the car to drive on itself in the field were developed correctly. Integration of the system has resulted in a car that can drive from a given start location to any location in the field with a relatively small error.

8

Discussion

When looking back on the project there are a lot of things that went right, but there are also a lot of improvements to make. Most of these improvements were not carried out because there was not enough time left after finishing the project.

First of all, more testing would have made it possible to, besides driving to a location in the field, avoid objects and drive to a location with a waypoint in between. Possible code solutions for challenge C, driving with an obstacle in the field and D, driving with another, stationary, but active beacon car and other obstacles in the field, were discussed but never implemented due to time limitations.

For C a quick detection algorithm was implemented using a 460x460 array, where each index represented a 10cm x 10cm area in the real world. By using a direction vector scaled by the distance that either sensor detected something, this array could have been filled with locations where possible obstacles have been detected. On this a path-finding algorithm could've been applied, which then uses the working challenge A code to go to a final destination.

”Gold codes”, a necessary part of challenge D were implemented and discussed, but challenge D was not implemented any further.

The error on the coordinates where the car stops, could be improved by doing more iterations of testing and improving the code. Also, taking time to implement the TDOA using 3D coordinates would have resulted in a smaller error. A faster version of the algorithm of multi-threaded version could also have been developed which would have decreased the reliance on the car model, and allowed for other optimizations, for example, not having to stop the car during the challenge to measure position.

References

[1] pressbooks.bccampus.ca. *Rotation Angle and Angular Velocity*. 2024. URL: https://pressbooks.bccampus.ca/douglasphys1107/chapter/9-1-rotation-angle-and-angular-velocity/ (visited on 2024-05-24).

[2] Dasika Sri Bhuvana Vaishnavi. *Gold Code Generators*. 2024. URL: https://medium.com/@dasikavaishnavi/gold-code-generators-d784e2b87984 (visited on 2024-06-12).

[3] A. van der Veen and B. Abdikivanani. *Autonomous driving challenge*. Delft, The Netherlands: Delft University of Technology, 2024.

A

Python code and results

A.1. KITT class

Listing A.1: KITT class

```
import serial
import keyboard
import sys
import os
import time
import csv
from datetime import datetime
import subprocess
import threading
import queue
import collections

class KITT:
    def __init__(self, port, baudrate=115200):
        self.serial = None
        self.EstopF = False
        self.BeaconFlag = False

        self.beacon = False
        self.c = int(0)
        self.f_c = int(0)
        self.f_b = int(0)
        self.c_r = int(0)
        self.dir = int(0)
        self.mot = int(0)
        self.l = int(0)
        self.r = int(0)
        self.batt = float(0)

        self.speed = int(150)
        self.angle = int(150)

        self.history = []

        if os.name == 'nt':
            n = len(os.listdir('utilities/data'))
            self.filename = f'utilities/data/report_log{n}.csv'
        else:
            n = len(os.listdir('../../utilities/data'))
            self.filename = f'../../utilities/data/report_log{n}.csv'

        with open(self.filename, 'x') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(['Time', 'Sensor_L', 'Sensor_R', 'Battery_V', 'One_Direction', '
                Motór'])

```

```
46          for i in range(5):
47              try:
48                  self.serial = serial.Serial(port, baudrate, rtscts=True)
49                  break
50              except serial.serialutil.SerialException:
51                  print(i)
52          if self.serial == None:
53              raise Exception
54
55          self.setBeacon(carrier_freq = 5000, bit_frequency = 5000, repition_count = 2500, code
                = 0xB00B1E50)
56
57      def send_command(self, command):
58          self.serial.write(command.encode())
59
60      def set_speed(self, speed):
61          self.speed = speed
62          self.send_command(f'M{speed}\n')
63
64      def set_angle(self, angle):
65          self.angle = angle
66          self.send_command(f'D{angle}\n')
67
68      def stop(self):
69          self.set_speed(150)
70          self.set_angle(150)
71
72      def setBeacon(self, carrier_freq = 1000, bit_frequency = 5000, repition_count = 2500,
            code = 0xEB79D549):
73          carrier_freq = carrier_freq.to_bytes(2, byteorder= 'big')
74          self.serial.write( b'F' + carrier_freq + b'\n')
75          bit_frequency = bit_frequency.to_bytes(2, byteorder= 'big')
76          self.serial.write(b'B' + bit_frequency + b'\n')
77          repition_count = repition_count.to_bytes(2, byteorder= 'big')
78          self.serial.write(b'R' + repition_count + b'\n')
79          code = code.to_bytes(4, byteorder= 'big')
80          self.serial.write(b'C' + code + b'\n')
81
82      def setFreq(self, freq):
83          freq = freq.to_bytes(2, byteorder= 'big')
84          self.serial.write( b'F' + freq + b'\n')
85
86      def startBeacon(self):
87          if self.BeaconFlag == True:
88              return
89          else:
90              self.serial.write(b'A1\n')
91              self.BeaconFlag = True
92
93      def stopBeacon(self):
94          if self.BeaconFlag == True:
95              self.serial.write(b'A0\n')
96              self.BeaconFlag = False
97          else:
98              return
99
100     def sitrep(self):
101         self.serial.write(b'S\n')
102         status = self.serial.read_until(b'\x04')
103         return status
104
105     def print_status(self):
106         string = str(self.sitrep())
107         i = 0
108         while i < len(string):
109             if string[i] == "\\" and string [i+1] == "n":
110                 print()
111                 i = i + 1
112
113             else:
114                 print(string[i], end='')
```

```
115
116             i = i + 1
117         print()
118
119     def log_status(self):
120         # This routine splits the string and puts it
121         # into variables that can be then logged
122         string = str(self.sitrep())
123
124         i = 0
125         while i < len(string):
126             if string[i] == ':' and string[i-1] == 'n' and string[i-2] == 'o':
127                 if (string[i+2] == 'o' and string[i+3] == 'n'):
128                     self.beacon = True
129                 else:
130                     self.beacon == False
131
132             if string[i] == ':' and string[i-1] == 'c' and string[i-2] == '␣':
133                 # This has to converted from hex to dec
134                 self.c = string[i+4:i+11]
135
136             if string[i] == 'c' and string[i-1] == '_' and string[i-2] == 'f':
137                 self.f_c = int((string[i+3:i+9].split('\\'))[0])
138
139             if string[i] == 'b' and string[i-1] == '_' and string[i-2] == 'f':
140                 self.f_b = int((string[i+3:i+9].split('\\'))[0])
141
142             if string[i] == 'r' and string[i-1] == '_' and string[i-2] == 'c':
143                 self.c_r = int((string[i+3:i+9].split('\\'))[0])
144
145             if string[i] == 'r' and string[i-1] == 'i' and string[i-2] == 'D':
146                 self.dir = int((string[i+3:i+9].split('\\'))[0])
147
148             if string[i] == 't' and string[i-1] == 'o' and string[i-2] == 'M':
149                 self.mot = int((string[i+3:i+9].split('\\'))[0])
150
151             if string[i] == 'L' and string[i-1] == '␣' and string[i-2] == '.':
152                 self.l = int((string[i+2:i+9].split('␣'))[0])
153
154             if string[i] == '␣' and string[i-1] == 'R' and string[i-2] == '␣':
155                 self.r = int((string[i+1:i+9].split('\\'))[0])
156
157             if string[i] == 't' and string[i-1] == 't' and string[i-2] == 'a':
158                 try:
159                     self.batt = float((string[i+2:i+9].split('␣'))[0])
160                 except ValueError:
161                     self.batt = 0.0
162
163
164             i += 1
165
166         if self.EstopCondition():
167             self.Estop()
168
169         # For now it only logs the sensors l, r, and a timestamp
170         now = datetime.now()
171         current_time = now.strftime("%M:%S.%f")[:-3]
172         with open(self.filename, 'a', newline='') as csvfile:
173             writer = csv.writer(csvfile)
174             writer.writerow([current_time, self.l, self.r, self.batt, self.dir, self.mot])
175
176         l = [self.l, self.r, self.batt, self.dir, self.mot]
177         self.history.append(l)
178         return l
179
180     def updateDirectionKeyboard(self):
181         speed = 150
182         angle = 150
183
184         if keyboard.is_pressed('up') or keyboard.is_pressed('w'):
185             speed += 15
```

```
186
187         if keyboard.is_pressed('left') or keyboard.is_pressed('a'):
188             angle += 35
189
190         if keyboard.is_pressed('down') or keyboard.is_pressed('s'):
191             speed -= 15
192
193         if keyboard.is_pressed('right') or keyboard.is_pressed('d'):
194             angle -= 35
195
196         if keyboard.is_pressed('q'):
197             self.startBeacon()
198
199         if keyboard.is_pressed('e'):
200             self.stopBeacon()
201
202
203         return speed, angle
204
205     def initJoystick(self):
206         # Run the binnary as a separate process that will read the joysticks
207         self.proc = subprocess.Popen("../../utilities/bin/joystick", stdout=subprocess.PIPE)
208         # os.set_blocking(self.proc.stdout.fileno(), False)
209         self.q = collections.deque(maxlen=1)
210         t = threading.Thread(target=read_output, args=(self.proc, self.q.append))
211         t.daemon = True
212         t.start()
213
214     def updateDirectionStick(self):
215         # Read from the stdout of the binary and update the joysticks
216         # tmp = self.proc.stdout.readline()
217         try:
218             tmp = self.q[0]
219         except IndexError:
220             tmp = 0
221
222         if len(str(tmp)) < 8:
223             speed = 0
224             angle = 0
225
226         else:
227             angle = int(str(tmp)[2:5])
228             speed = int(str(tmp)[-6:-3])
229
230         return speed, angle
231
232     def EstopCondition(self): ## Should ESTOP be initiated automatically // TODO
233         derR,derL = self.DistanceDerivative()
234         if derR  == 0:
235             derR = 1e-33
236         elif abs(derR) > 20:
237             derR = 1e-33
238
239         if derL == 0:
240             derL = 1e-33
241         elif abs(derL) > 20:
242             derL = 1e-33
243         TimeToImpact = -min(self.l/derL, self.r/derR)
244         if 0 < TimeToImpact < 3:
245             return True
246         else:
247             return False
248
249     def DistanceDerivative(self):
250         timediff = self.time_new - self.time_old
251         if timediff == 0:
252             timediff = 1e-33
253         derL = (self.l - self.l_old) / timediff
254         derR = (self.r - self.r_old) /  timediff
255         return derL, derR
256
```

```
257
258     def Estop(self):
259         speed = -self.speed + 300
260         self.set_speed(speed)
261         time.sleep(1)
262         self.stop()
263         self.EstopF = True
264
265
266 def read_output(process, append):
267     for line in iter(process.stdout.readline, ""):
268         append(line)
```

A.1.1. Joystick Remote Control

Listing A.2: Joystick remote control

```
1  #!/bin/python3.10
2  import keyboard
3  import sys
4  import os
5  import time
6
7  sys.path.append(os.path.join(os.path.dirname(sys.path[0]), 'inc'))
8  from KITT import KITT
9
10 if os.name == 'nt':
11     comPort = 'COM13'
12 else:
13     comPort = '/dev/rfcomm0'
14
15 def tick():
16     speed,angle = kitt.updateDirectionStick()
17
18     if (speed != 0 and angle != 0):
19         kitt.set_speed(speed)
20         kitt.set_angle(angle)
21
22 def alt_c():
23     global WhichInput
24     WhichInput = not WhichInput
25     if WhichInput:
26         print('joystick␣Selected')
27     else:
28         print('keyboard␣Selected')
29
30 WhichInput = False
31
32 if __name__ == '__main__':
33     keyboard.add_hotkey('alt+c', alt_c)
34
35     kitt = KITT(comPort)
36     kitt.initJoystick()
37
38     while True:
39         try:
40             tick()
41             #kitt.print_status()
42             kitt.log_status()
43
44         except KeyboardInterrupt:
45             kitt.serial.close()
46             break
47
48     kitt.serial.close()
```

A.1.2. Sensor Utility

Listing A.3: sensor calibration utility

```
1      #!/bin/python3.10
```

```
import serial
import time
import os

class KITT:
    def __init__(self, port, baudrate=115200):
        self.serial = serial.Serial(port, baudrate, rtscts=True)

    def sitrep(self):
        self.serial.write(b'Sd\n')
        status = self.serial.read_until(b'\x04')
        return status

    def print_status(self):
        string = str(self.sitrep())

        i = 0
        while i < len(string):
            if string[i] == "\\" and string [i+1] == "n":
                print()
                i = i + 1

            else:
                print(string[i], end='')

            i = i + 1

        print()

if __name__ == '__main__':
    kitt = KITT('/dev/rfcomm0')
    while(1):
        kitt.print_status()
        time.sleep(0.008)
        os.system('clear')
```

A.2. Audio class

Listing A.4: Audio

```
class Audio:
    def __init__(self, Fs = 44100, callback = False, noAudio = False):
        if not noAudio:
            self.handle = pyaudio.PyAudio()
            self.Fs = Fs
            index = self._detect_microphones()[0]
            if callback == False :
                self.microphones = self.handle.open(input_device_index= index,
                                                    channels=5,
                                                    format = pyaudio.paInt16,
                                                    rate = self.Fs,
                                                    input = True,
                                                    )
            else:

                self.microphones = self.handle.open(input_device_index = index,
                                                    channels=5,
                                                    format = pyaudio.paInt16,
                                                    rate = self.Fs,
                                                    input = True,
                                                    stream_callback= self.callback
                                                    )

            self.callback_data = None

    def _detect_microphones(self):
        j = []
        for i in range(self.handle.get_device_count()):
            devive_info = self.handle.get_device_info_by_index(i)
            if "AudioBox␣1818" in devive_info['name'] and "Microphone" in devive_info['name'
```

```
                ]:
                j.append(i)

        if len(j) == 0:
            print("AudioBox 1818 not found, please try again")
            sys.exit(-1)
        print(j)

        return j

    def callback(self,in_data, frame_count, time_info, status):
        self.callback_data = np.frombuffer(in_data, dtype='int16')

        return in_data, pyaudio.paContinue

    def sample(self, N):
        return np.frombuffer(self.microphones.read(N), dtype='int16')

    def split_data(self,data):
        mic1 = data[0::5]
        mic2 = data[1::5]
        mic3 = data[2::5]
        mic4 = data[3::5]
        mic5 = data[4::5]

        return [mic1,mic2,mic3,mic4,mic5]

    def play_sound(self, array):
        play = self.handle.open(format=pyaudio.paInt16, channels=1, rate=self.Fs, output=True
            )
        play.write(array.tobytes())
        play.stop_stream()
        play.close()

    def writeAlltoWave(self, mic1, mic2,mic3,mic4,mic5, fileappendix = ''):
        write(f'{fileappendix}mic1.wav', self.Fs, mic1.astype(np.int16))
        write(f'{fileappendix}mic2.wav', self.Fs, mic2.astype(np.int16))
        write(f'{fileappendix}mic3.wav', self.Fs, mic3.astype(np.int16))
        write(f'{fileappendix}mic4.wav', self.Fs, mic4.astype(np.int16))
        write(f'{fileappendix}mic5.wav', self.Fs, mic5.astype(np.int16))

    def write1toWav(self, data, filename = "File.wav"):
        write(filename, self.Fs, data)

    def saveToPickle(self, data, filename = "File.pkl"):
        with open('runTest.pkl', 'wb') as outp:
            pickle.dump(data, outp, pickle.HIGHEST_PROTOCOL)

    ## deleted writeto1wav because it didnt work

    def close(self):
        self.microphones.close()
        self.handle.terminate()

    def __del__(self):
        self.microphones.close()
        self.handle.terminate()
```

A.3. TDOA class

A.3.1. Channel Estimation (ch3)

Listing A.5: ch3

```
def ch3(self,x,y,Lhat,epsi):
        Nx = len(x)
        Ny = len(y)
        Nh = Lhat

        x = np.concatenate((x, np.zeros(Ny-Nx)))

```

```
        Y = fft(y)
        X = fft(x)
        H = Y/X

        H[np.absolute(X) < epsi*max(np.absolute(X))] = 0
        h = np.real(ifft(H))

        return h
```

A.3.2. Localization

Listing A.6: Localization

```
def localization(self, x1, y, Fs):
        epsi = 0.01     #threshold value used in ch3
        v = 343.21      #speed of sound
        Lhat1 = len(y) - len(x1) + 1    #equal length of x and y

        x1 = x1[len(x1)-25000:]         #crop x to the last pulse(s)

        y = y[len(y)-25000:]            #crop y to the last pulse(s)

        h0 = self.ch3(x1, y[:, 0], Lhat1, epsi)     #create the channel estimates
        h1 = self.ch3(x1, y[:, 1], Lhat1, epsi)     #for all microphones
        h2 = self.ch3(x1, y[:, 2], Lhat1, epsi)
        h3 = self.ch3(x1, y[:, 3], Lhat1, epsi)
        h4 = self.ch3(x1, y[:, 4], Lhat1, epsi)

        tau12 = ((abs(h0).argmax() - abs(h1).argmax())*v/Fs) #calculate the difference
        tau14 = ((abs(h0).argmax() - abs(h3).argmax())*v/Fs) #between peaks in the
        tau13 = ((abs(h0).argmax() - abs(h2).argmax())*v/Fs) #channel estimates
        tau15 = ((abs(h0).argmax() - abs(h4).argmax())*v/Fs)

        p1 = np.array([0, 0])       #define microphone positions
        p2 = np.array([0, 4.60])
        p3 = np.array([4.60, 4.60])
        p4 = np.array([4.60, 0])
        p5 = np.array([0, 2.30])

        A = np.array([[p1[0]-p2[0],p1[1]-p2[1],tau12],
                      [p1[0]-p3[0],p1[1]-p3[1],tau13],
                      [p1[0]-p4[0],p1[1]-p4[1],tau14],
                      [p1[0]-p5[0],p1[1]-p5[1],tau15]])

        C = np.array([[0.5*(p1[0]**2-p2[0]**2+p1[1]**2-p2[1]**2+tau12**2)],
                      [0.5*(p1[0]**2-p3[0]**2+p1[1]**2-p3[1]**2+tau13**2)],
                      [0.5*(p1[0]**2-p4[0]**2+p1[1]**2-p4[1]**2+tau14**2)],
                      [0.5*(p1[0]**2-p5[0]**2+p1[1]**2-p5[1]**2+tau15**2)]])

        B = np.linalg.lstsq(A, C, rcond=None)[0][:2].flatten()  #matrix calculation
                                                                #to get coordinates

        return B
```

A.3.3. Localization in the field (Whole TDOA class)

Listing A.7: TDOA class

```
class TDOA:
    def ch3(self,x,y,Lhat,epsi):
        Nx = len(x)
        Ny = len(y)
        Nh = Lhat

        x = np.concatenate((x, np.zeros(Ny-Nx)))

        Y = fft(y)
        X = fft(x)
        H = Y/X

        H[np.absolute(X) < epsi*max(np.absolute(X))] = 0
```

```
        h = np.real(ifft(H))

        return h

    def localization(self, x1,x2,x3,x4,x5, y, Fs):
        def channel_estimate(x1,x2,x3,x4,x5,y,Fs):
            epsi = 0.01     #threshold value used in ch3
            Lhat1 = len(y) - len(x1) + 1    #equal length of x and y

            x1 = x1[len(x1)-25000:]         #crop x to the last pulse(s)

            y = y[len(y)-25000:]            #crop y to the last pulse(s)

            h0 = self.ch3(x1, y[:, 0], Lhat1, epsi)     #create the channel estimates
            h1 = self.ch3(x1, y[:, 1], Lhat1, epsi)     #for all microphones
            h2 = self.ch3(x1, y[:, 2], Lhat1, epsi)
            h3 = self.ch3(x1, y[:, 3], Lhat1, epsi)
            h4 = self.ch3(x1, y[:, 4], Lhat1, epsi)

            return h0, h1, h2, h3, h4

        def closest_mic(h0,h1,h2,h3):
            distances = np.array([[abs(h0).argmax()], [abs(h1).argmax()], [abs(h2).argmax()],
                 [abs(h3).argmax()]]) #array of all peaks
            min_pos = np.argmin(distances) + 1  #takes the indice of peak with the lowest
            print(min_pos)                      #sample number, adds 1 to get the
            return min_pos                      #corresponding mic number

        def matrix_calc(h0,h1,h2,h3,h4,cl_mic):
            v = 343.21      #speed of sound
            p1 = np.array([0, 0])       #define microphone positions
            p2 = np.array([0, 4.60])    ####PAY ATTENTION TO 480/460###
            p3 = np.array([4.60, 4.60])
            p4 = np.array([4.60, 0])
            p5 = np.array([0, 2.30])

            if cl_mic == 1:
                tau12 = ((abs(h0).argmax() - abs(h1).argmax())*v/Fs) #calculate the
                tau14 = ((abs(h0).argmax() - abs(h3).argmax())*v/Fs) #difference between
                tau13 = ((abs(h0).argmax() - abs(h2).argmax())*v/Fs) #peaks in the channel
                tau15 = ((abs(h0).argmax() - abs(h4).argmax())*v/Fs) #estimates

                A = np.array([[p1[0]-p2[0],p1[1]-p2[1],tau12],
                              [p1[0]-p3[0],p1[1]-p3[1],tau13],
                              [p1[0]-p4[0],p1[1]-p4[1],tau14],
                              [p1[0]-p5[0],p1[1]-p5[1],tau15]])

                C = np.array([[0.5*(p1[0]**2-p2[0]**2+p1[1]**2-p2[1]**2+tau12**2)],
                              [0.5*(p1[0]**2-p3[0]**2+p1[1]**2-p3[1]**2+tau13**2)],
                              [0.5*(p1[0]**2-p4[0]**2+p1[1]**2-p4[1]**2+tau14**2)],
                              [0.5*(p1[0]**2-p5[0]**2+p1[1]**2-p5[1]**2+tau15**2)]])

                B = np.linalg.lstsq(A, C, rcond=None)[0][:2].flatten()  #matrix calculation
                                                                        #to get coordinates
            elif cl_mic == 2:
                tau21 = ((abs(h1).argmax() - abs(h0).argmax())*v/Fs) #calculate the
                tau23 = ((abs(h1).argmax() - abs(h2).argmax())*v/Fs) #difference between
                tau24 = ((abs(h1).argmax() - abs(h3).argmax())*v/Fs) #peaks in the channel
                tau25 = ((abs(h1).argmax() - abs(h4).argmax())*v/Fs) #estimates

                A = np.array([[p2[0]-p1[0],p2[1]-p1[1],tau21],
                              [p2[0]-p3[0],p2[1]-p3[1],tau23],
                              [p2[0]-p4[0],p2[1]-p4[1],tau24],
                              [p2[0]-p5[0],p2[1]-p5[1],tau25]])

                C = np.array([[0.5*(p2[0]**2-p1[0]**2+p2[1]**2-p1[1]**2+tau21**2)],
                      [0.5*(p2[0]**2-p3[0]**2+p2[1]**2-p3[1]**2+tau23**2)],
                      [0.5*(p2[0]**2-p4[0]**2+p2[1]**2-p4[1]**2+tau24**2)],
                      [0.5*(p2[0]**2-p5[0]**2+p2[1]**2-p5[1]**2+tau25**2)]])

                B = np.linalg.lstsq(A, C, rcond=None)[0][:2].flatten()  #matrix calculation
```

```
                                                                    #to get coordinates
            elif cl_mic == 3:
                tau31 = ((abs(h2).argmax() - abs(h0).argmax())*v/Fs) #calculate the
                tau32 = ((abs(h2).argmax() - abs(h1).argmax())*v/Fs) #difference between
                tau34 = ((abs(h2).argmax() - abs(h3).argmax())*v/Fs) #peaks in the channel
                tau35 = ((abs(h2).argmax() - abs(h4).argmax())*v/Fs) #estimates

                A = np.array([[p3[0]-p1[0],p3[1]-p1[1],tau31],
                      [p3[0]-p2[0],p3[1]-p2[1],tau32],
                      [p3[0]-p4[0],p3[1]-p4[1],tau34],
                      [p3[0]-p5[0],p3[1]-p5[1],tau35]])

                C = np.array([[0.5*(p3[0]**2-p1[0]**2+p3[1]**2-p1[1]**2+tau31**2)],
                      [0.5*(p3[0]**2-p2[0]**2+p3[1]**2-p2[1]**2+tau32**2)],
                      [0.5*(p3[0]**2-p4[0]**2+p3[1]**2-p4[1]**2+tau34**2)],
                      [0.5*(p3[0]**2-p5[0]**2+p3[1]**2-p5[1]**2+tau35**2)]])

                B = np.linalg.lstsq(A, C, rcond=None)[0][:2].flatten()  #matrix calculation
                                                                    #to get coordinates
            else:
                tau41 = ((abs(h3).argmax() - abs(h0).argmax())*v/Fs) #calculate the
                tau42 = ((abs(h3).argmax() - abs(h1).argmax())*v/Fs) #difference between
                tau43 = ((abs(h3).argmax() - abs(h2).argmax())*v/Fs) #peaks in the channel
                tau45 = ((abs(h3).argmax() - abs(h4).argmax())*v/Fs) #estimates

                A = np.array([[p4[0]-p1[0],p4[1]-p1[1],tau41],
                      [p4[0]-p2[0],p4[1]-p2[1],tau42],
                      [p4[0]-p3[0],p4[1]-p3[1],tau43],
                      [p4[0]-p5[0],p4[1]-p5[1],tau45]])

                C = np.array([[0.5*(p4[0]**2-p1[0]**2+p4[1]**2-p1[1]**2+tau41**2)],
                      [0.5*(p4[0]**2-p2[0]**2+p4[1]**2-p2[1]**2+tau42**2)],
                      [0.5*(p4[0]**2-p3[0]**2+p4[1]**2-p3[1]**2+tau43**2)],
                      [0.5*(p4[0]**2-p5[0]**2+p4[1]**2-p5[1]**2+tau45**2)]])

                B = np.linalg.lstsq(A, C, rcond=None)[0][:2].flatten()  #matrix calculation
                                                                    #to get coordinates
            return B
        d = channel_estimate(x1,x2,x3,x4,x5,y,Fs)       #makes the channel estimates
                                                        #for each mic
        cl_mic = closest_mic(d[0],d[1],d[2],d[3])       #finds the reference mic
        coordinates = matrix_calc(d[0],d[1],d[2],d[3],d[4],cl_mic)  #does the matrix
                                                                    #calculation taking into
                                                                    #account the reference
                                                                    #mic
        return coordinates

    def tdoa_input(self,mic1,mic2,mic3,mic4,mic5):
        b = []
        b = np.stack((mic1, mic2, mic3, mic4, mic5), axis =-1)  #takes input arrays and
            stacks
                                                                #them in a 3d array
                                                                #with each column containing
                                                                #a different microphone
        return b
```

A.4. GoldCodeGenerator

Listing A.8: Gold code Generator

```
import numpy
def gen_gold(seq1, seq2):
    """Function to produce a gold sequence based on two input preferred pair Maximal Length
        Sequences."""
    gold = [seq1, numpy.roll(seq2, 1)]  # Initial gold code with seq1 and a single shift of
        seq2
    for shift in range(1, len(seq1)):
        gold.append(numpy.logical_xor(seq1, numpy.roll(seq2, shift)))
    return gold

```

```

seq1 = numpy.random.choice([0, 1], size=32)  # Random binary sequence
seq2 = numpy.random.choice([0, 1], size=32)

seq1 = numpy.where(seq1, 1.0, -1.0)
seq2 = numpy.where(seq2, 1.0, -1.0)

gold = gen_gold(seq1, seq2)[0]
print(gold)

gold = numpy.where(gold == 1, 1, 0)
print(numpy.count_nonzero(gold))
print(len(gold))
print(gold)
```

A.5. PID Controllers

Listing A.9: PID control class

```
import numpy as np

class PID:
    def __init__(self):
        self.angleIntegral = 0
        self.LastAngleError = 0

        self.distIntegral = 0
        self.LastdistError = 0

        self.minForce = 1.04
        self.maxForce = 8.91
        self.minAngle = 0.0872665
        self.maxAngle = 0.349066

        self.ForceList  = [8.91,4.992,1.04,1.03,0,-2.70,-2.71,-6.24,-9.984]
        self.PWMList    = [165,160,156,150,150,150,146,140,135]

        self.ForceKp = 2
        self.ForceKi = 0.0
        self.ForceKd = 0.0

        self.AngleKp = 0.8
        self.AngleKi = 0.0
        self.AngleKd = 0.0


        ## test values
        self.angle = 0


    def CalculateErrors(self, setpointx, setpointy, currx, curry, currentAngle):
        ## (x,y) is Vector pointing from current position to setpoint
        x = setpointx - currx
        y = setpointy - curry
        deltaP = np.sqrt((x)**2 + (y)**2)

        ## Prevent rounding errors
        if deltaP < 0.1:
            deltaP = 0.0

        ## Angle of pointing vector
        angle = np.arctan2(y,x)
        ## DeltaTheta is the angle between the direction vector and the pointing vector
        deltaTheta = angle - currentAngle

        ## Limit deltaTheta between -pi and pi
        while not ( -np.pi <= deltaTheta <= np.pi):
            deltaTheta = -(np.sign(deltaTheta) * 2 * np.pi) + deltaTheta

        if abs(deltaTheta) > np.pi/2:
```

```
52              deltaP = -deltaP
53              deltaTheta = (np.sign(deltaTheta) * np.pi) + deltaTheta
54
55          ## Limit deltaTheta between -pi and pi
56          while not ( -np.pi <= deltaTheta <= np.pi):
57              deltaTheta = -(np.sign(deltaTheta) * 2 * np.pi) + deltaTheta
58
59          return deltaP, deltaTheta
60
61      def calculateForce(self, deltaP, deltaT):
62          self.distIntegral += deltaP * deltaT
63          distDiff = (deltaP - self.LastdistError ) / deltaT
64          self.lastdistError = deltaP
65
66          Force = (self.ForceKp * deltaP) + (self.ForceKi * self.distIntegral) + (self.ForceKd
                * distDiff)
67
68          if abs(Force) > self.maxForce:
69              Force = np.sign(Force) * self.maxForce
70
71          if abs(Force) != 0.0:
72              if abs(Force) > self.maxForce:
73                  Force = np.sign(Force) * self.maxForce
74              elif abs(Force) < self.minForce:
75                  Force = 0.0
76
77          return Force
78
79      def calculateAngle(self, deltaTheta, deltaT):
80          self.angleIntegral += deltaTheta * deltaT
81          angleDiff = (deltaTheta - self.LastAngleError)
82          self.LastAngleError = deltaTheta
83
84
85          Angle = (self.AngleKp * deltaTheta) + (self.AngleKi * self.angleIntegral) + (self.
                AngleKd * angleDiff)
86
87          if abs(Angle) < self.minAngle:
88              Angle = 0
89          elif abs(Angle) > self.maxAngle:
90              Angle = np.sign(Angle) * self.maxAngle
91
92          return Angle
93
94      def Update(self, setpointx, setpointy, currx, curry, currentAngle, deltaT):
95          deltaP, deltaTheta = self.CalculateErrors(setpointx, setpointy, currx, curry,
                currentAngle)
96
97          Force = self.calculateForce(deltaP, deltaT)
98          Angle = self.calculateAngle(deltaTheta, deltaT)
99
100         return Force, Angle
101
102
103
104     def ForcetoPWM(self, Force):
105         for i in range(1,len(self.ForceList)):
106             if self.ForceList[i-1] >= Force >= self.ForceList[i]:
107                 PWM1 = self.PWMList[i-1]
108                 PWM2 = self.PWMList[i]
109                 Force1 = self.ForceList[i-1]
110                 Force2 = self.ForceList[i]
111                 break
112
113
114         return int(np.round(((PWM2 - PWM1)/(Force2 - Force1)) * (Force - Force1) + PWM1, 0))
115
116     def AngletoPWM(self, Angle):
117         return int(np.round(10/6 * Angle + 150, 0))
118
119     def RadiansToPWM(self, Angle):
```

```
        PWM1   = 100
        PWM2   = 200
        Phi1 = -self.maxAngle
        Phi2 = self.maxAngle
        return int(np.round(((PWM2 - PWM1)/(Phi2 - Phi1)) * (Angle - Phi1) + PWM1, 0))
```

A.6. Pure Pursuit

```
import math
import numpy as np
from shapely.geometry import LineString
from shapely.geometry import Point
from module4 import KITTmodel
from PID import PID

class purePursuit:
    start_point = []
    end_point = []

    def __init__(self): #initialises all variables
        self.intersec_1 = 0
        self.intersec_2 = 0
        self.targetPoint = 0
        self.wheelbase = 0.335
        self.x_location = 0
        self.y_location = 0

    def intersections(self, _location_x, _location_y, _x1, _y1, _x2, _y2):
        """_summary_
            calculates the target point:
                - draws a line using the start and end points
                - creates a circle of radius lookaheaddistance around kitt
                - takes the intersection of the line and the circle
                - returns array of intersections
                uses the shapely library for these calculations
            Args:
                _location_x (_type_): x_position of kitt
                _location_y (_type_): y_position of kitt
                _x1 (_type_): start position x
                _y1 (_type_): start position y
                _x2 (_type_): end position x
                _y2 (_type_): end position y
            Returns:
                _type_: array of intersection coordinates
        """
        _point = Point(_location_x, _location_y)
        _circle = _point.buffer(self.lookAheadDistance)
        _path = LineString([(_x1, _y1), (_x2, _y2)])
        _intersection = _circle.intersection(_path)

        if len(_intersection.coords) == 2:
            return np.array([(_intersection.coords[0]),
                (_intersection.coords[1])])
        elif len(_intersection.coords) == 1:
            return np.array([(_intersection.coords[0])])
        else:
            print("nothing␣found")
            return  np.array([(0,0), (0,0)])

    def steeringAngle(self, _x_tp, _y_tp, orientation):
        """deterimines the steering angle for kitt based on the target point
        Args:
            x_tp (_type_): x coordinate of the target point
            y_tp (_type_): y coordinate of the target point
            orientation  : current orientation of the car
        Returns:
        _   steering_angle: steering angle
        """

        _alpha = np.arctan2((_y_tp - self.y_location), (_x_tp - self.x_location)) -
```

```
            orientation

        _angle = np.arctan((2 * self.wheelbase * np.sin(_alpha))/self.lookAheadDistance)

        return _angle

    def distance_calc(self, x2, y2, x1, y1):
        _distance = np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
        return _distance

    def point_selection(self, point_1, point_2, x_destination, y_destination):
        _distance_1 = self.distance_calc(point_1[1], point_1[0], x_destination,
        y_destination)
        _distance_2 = self.distance_calc(point_2[1], point_2[0], y_destination,
        x_destination)
        if _distance_1 <= _distance_2:
            return point_1
        else:
            return point_2

    def purepursuit(self, _x_position, _y_position, _x_target, _y_target, orientation):
        """
        Args:
            _x_position: x coordinate of the car
            _y_position: y coordinate of the car
            _x_target:   x coordinate of the target point
            _y_target:   y coordinate of the target point
            orientation: current orientation of the car
        Returns:
            angle      : steering angle in radians
        """
        for multiplier in np.linspace(0.1, 2, 20):
            self.lookAheadDistance = multiplier * self.distance_calc(_x_target, _y_target,
                _x_position, _y_position) # the radius of the circle
            self.intersection = self.intersections(_x_position, _y_position, self.x_location,
                 self.y_location, _x_target, _y_target)
            if self.intersection[0,0] == 0 and self.intersection[1,1] == 0 and self.
                intersection[1,0] == 0 and self.intersection[0,1] == 0:
                continue
            else:
                break
        self.Target = self.point_selection(self.intersection[0], self.intersection[1],
            _x_target, _y_target)

        self.angle = self.steeringAngle(self.Target[0], self.Target[1], orientation)

        return self.angle
```

A.7. Integrated Code

A.7.1. Challenge A - Driving to a point on the field

Listing A.10: Challenge A integrated Code

```
    #!/bin/python3.10

import numpy as np



def Challenge_A(cX, cY, Theta, tX, tY, target_offset, calibration_delay, sleeptime):
    import sys
    import os
    import time
    from pathlib import Path

    sys.path.append(os.path.join(os.path.dirname(sys.path[0]), 'inc'))
    from wavaudioread import wavaudioread
    from KITT import KITT
    from Audio import Audio
    from module4 import KITTmodel
```

```
    from PID import PID
    from var_ref_tdoa import TDOA
    from GUI import GUI


    ## Static Init Variables
    run_time  = 19e-127          # 0 but not 0 so the divide by 0 does not break
    size_of_the_field = 4.60     # It's a square

    #Check if the target is valid
    if ((tX > size_of_the_field) or (tY > size_of_the_field)):
        print("Invalid target, BYE! ")
        exit()

    # Checiking the os and connecting to the port
    if os.name == 'nt':
        comPort = 'COM5'
    else:
        comPort = '/dev/rfcomm0'


    # Initializing the mics and reference files
    mics = Audio()
    Fs = 44100
    N = Fs * 1
    data = mics.sample(N)
    samples = mics.split_data(data)



    # Initialize all of the objects
    kittmodel = KITTmodel()
    kitt = KITT(comPort)
    pid = PID()
    T = TDOA()


    #Using reference recordings from all microphones
    y = T.tdoa_input(samples[0], samples[1], samples[2], samples[3], samples[4])
    root_folder = Path("IntegrationTest.py").resolve().parent
    x1 = wavaudioread(root_folder / "utilities/data/Reference1.wav", Fs)
    x2 = wavaudioread(root_folder / "utilities/data/Reference2.wav", Fs)
    x3 = wavaudioread(root_folder / "utilities/data/Reference3.wav", Fs)
    x4 = wavaudioread(root_folder / "utilities/data/Reference4.wav", Fs)
    x5 = wavaudioread(root_folder / "utilities/data/Reference5.wav", Fs)


    kittmodel.position_state_vector = np.array([[cX],[cY]])
    kittmodel.theta = Theta

    # Checking the initial position (c for current)
    kitt.startBeacon()
    time.sleep(0.5)
    data = mics.sample(N)
    samples = mics.split_data(data)
    y = T.tdoa_input(samples[0], samples[1], samples[2], samples[3], samples[4])
    pos = T.localization(x1,x2,x3,x4,x5, y, Fs)
    print(pos)
    kitt.stopBeacon()

    print(cX, cY)

    G = GUI(cX, cY,Theta, tX, tY, sleeptime = sleeptime)

    # The loop function:
    i = 0
    while(1):

        if(i < calibration_delay - 5):
            # Getting the PWMs to move the car in the correct direction
            F, phi = pid.Update(tX, tY, cX, cY, Theta, run_time)
```

```
89              pwmMotor    = pid.ForcetoPWM(F)
90              pwmSteering = pid.RadiansToPWM(phi)
91
92              # Get the time when the car started to move
93              time_old = time.monotonic()
94
95              # Transmitting the PWMs to the car and making it move
96              kitt.set_speed(pwmMotor)
97              kitt.set_angle(pwmSteering)
98
99              #GUI update, also includes the sleep for movement
100             G.update(cX, cY, Theta, tX, tY)
101
102             # Calcualting the current position of the car with the model
103             run_time = time.monotonic() - time_old
104             position_Vector, Theta = kittmodel.update(phi, 4.06, run_time)
105             cX = position_Vector[0][0]
106             cY = position_Vector[1][0]
107
108             print("PMWs:␣")
109             print(pwmSteering, pwmMotor)
110             print()
111
112
113         # Calibrate the current position any other time
114         if (i == calibration_delay):
115             # Stop the car
116             #pwmSteering = 150
117             pwmMotor    = 150
118             F = 0
119             kitt.set_speed(pwmMotor)
120             #kitt.set_angle(pwmSteering)
121
122             # Wait utill it actually stops
123             time.sleep(1)
124
125             # Use the TDOA
126             kitt.startBeacon()
127             time.sleep(0.5)
128             data = mics.sample(N)
129             samples = mics.split_data(data)
130             y = T.tdoa_input(samples[0], samples[1], samples[2], samples[3], samples[4])
131             pos = T.localization(x1,x2,x3,x4,x5, y, Fs)
132
133             '''
134             if not (0 <= pos[0] <= size_of_the_field) or not (0 <= pos[1] <=
                    size_of_the_field):
135                 cX = cX
136                 cY = cY
137             else:
138                 cX = pos[0]
139                 cY = pos[1]
140                 print(f'TDOA POS{cX, cY}')
141             '''
142
143             cX = pos[0]
144             cY = pos[1]
145
146             kitt.stopBeacon()
147
148             # Update the position in the simulation and restart the models since the car
                    waited to stop
149             kittmodel = KITTmodel()
150             kittmodel.position_state_vector = np.array([[cX],[cY]])
151             kittmodel.theta = Theta
152
153             i -= calibration_delay
154
155
156         # Check if the target was reached with some margin
157         if ((abs(tX - cX) <= target_offset) and (abs(tY - cY) <= target_offset)):
```

```
158             # Stop the car
159             pwmSteering = 150
160             pwmMotor    = 150
161             F = 0
162             kitt.set_speed(pwmMotor)
163             kitt.set_angle(pwmSteering)
164 
165             # Wait until it actually stops
166             time.sleep(1)
167 
168             # Check if you actually reached the target
169             kitt.startBeacon()
170             time.sleep(0.5)
171             data = mics.sample(N)
172             samples = mics.split_data(data)
173             y = T.tdoa_input(samples[0], samples[1], samples[2], samples[3], samples[4])
174             pos = T.localization(x1,x2,x3,x4,x5, y, Fs)
175 
176             '''
177             if not (0 <= pos[0] <= size_of_the_field) or not (0 <= pos[1] <=
                    size_of_the_field):
178                 cX = cX
179                 cY = cY
180             else:
181                 cX = pos[0]
182                 cY = pos[1]
183                 print(f'TDOA POS{cX, cY}')
184             '''
185 
186             cX = pos[0]
187             cY = pos[1]
188             print(f'TDOA␣POS{cX,␣cY}')
189             kitt.stopBeacon()
190 
191             # Update the position in the simulation
192             kittmodel.position_state_vector = np.array([[cX], [cY]])
193 
194             # If it still equals then stop everything, if not then continue the loop
195             if ((abs(tX - cX) <= target_offset) and (abs(tY - cY) <= target_offset)):
196                 # Print the values that can be used to run the code again for challange B
197                 print(cX, cY, Theta)
198 
199                 kitt.serial.close()
200                 break
201 
202         # Log the report from the car
203         # kitt.log_status()
204 
205         # Incremeant the loop counter
206 
207         i = i + 1
208 
209     return cX, cY, Theta
210 
211 
212 if __name__ == '__main__':
213     # Define static variables
214     target_offset = 0.2      # The distance from the tarrget to still count success
215     calibration_delay = 15     # The amount of loop iterations before the TDOA calibration
216     size_of_the_field = 4.60    # It's a square
217     sleeptime = 0.1
218 
219     # Setup function:
220 
221     # Ask for the target input
222     print("Give␣target␣coordinates␣in␣meters␣separated␣by␣a␣new␣line")
223     print("in␣following␣notation:␣\"X\\nY\",␣ex.␣\"0.69\\n4.20\"")
224     tX = float(input())
225     tY = float(input())
226 
227     # For testing the initail position can be typed in
```

```
    print("Type␣in␣the␣current␣position␣in␣meters␣and␣angle␣in␣Radians␣/␣(np.pi/2)")
    cX = float(input())
    cY = float(input())
    Theta = np.pi/2*float(input())

    cX, cY, Theta = Challenge_A(cX, cY, Theta, tX, tY, target_offset, calibration_delay,
        sleeptime)

    print(cX, cY, Theta)
```

A.7.2. Challenge B - Driving from to a point, then to a second point

Listing A.11: Challenge B code

```
from A_TDOA import Challenge_A
import numpy as np
import time
import tkinter as tk


def inputGUI():
    global tX1, tY1, tX2, tY2, cX, cY, Theta
    root = tk.Tk()
    root.geometry("600x400")
    root.title('KITT␣Input␣GUI')

    def Submit():
        global tX1, tY1, tX2, tY2, cX, cY, Theta
        cX =        float(carX.get())
        cY =        float(carY.get())
        Theta =     np.pi/2*float(carT.get())
        tX1 =       float(X1.get())
        tY1 =       float(Y1.get())
        tX2 =       float(X2.get())
        tY2 =       float(Y2.get())

        root.destroy()
        root.quit()

    X1 = tk.StringVar()
    Y1 = tk.StringVar()
    X2 = tk.StringVar()
    Y2 = tk.StringVar()
    carX = tk.StringVar()
    carY = tk.StringVar()
    carT = tk.StringVar()

    NameLabel = tk.Label(root, text = 'Please␣enter␣all␣relevant␣Values')
    CarLabel = tk.Label(root, text = 'Car␣Position␣[x,␣y,␣T]:␣')
    Car1X = tk.Entry(root, textvariable= carX)
    Car1Y = tk.Entry(root, textvariable= carY)
    Car1T = tk.Entry(root, textvariable= carT)

    Target1Label = tk.Label(root, text = 'Target1␣[x,␣y]:␣')
    Target1X = tk.Entry(root, textvariable= X1)
    Target1Y = tk.Entry(root, textvariable= Y1)

    Target2Label = tk.Label(root, text = 'Target2␣[x,␣y]:␣')
    Target2X = tk.Entry(root, textvariable= X2)
    Target2Y = tk.Entry(root, textvariable= Y2)

    sub_btn=tk.Button(root,text = 'Submit', command = Submit)

    NameLabel       .grid(row = 0, column = 0)
    CarLabel        .grid(row = 1, column = 0)
    Car1X           .grid(row = 1, column = 1)
    Car1Y           .grid(row = 1, column = 2)
    Car1T           .grid(row = 1, column = 3)
    Target1Label    .grid(row = 2, column = 0)
    Target1X        .grid(row = 2, column = 1)
    Target1Y        .grid(row = 2, column = 2)
```

```
58      Target2Label    .grid(row = 3, column = 0)
59      Target2X        .grid(row = 3, column = 1)
60      Target2Y        .grid(row = 3, column = 2)
61      sub_btn         .grid(row = 4, column = 0)
62
63      root.mainloop()
64
65
66  if __name__ == '__main__':
67      # Define static variables
68      target_offset = 0.3      # The distance from the target to still count success
69      calibration_delay = 15     # The amount of loop iterations before the TDOA calibration
70      sleeptime = 0.1
71
72      # Setup function:
73
74      tX1 = 0.0
75      tY1 = 0.0
76      tX2 = 0.0
77      tY2 = 0.0
78      cX = 0.0
79      cY = 0.0
80      Theta = 0.0
81
82      inputGUI()
83
84      print(tX1, tY1, tX2, tY2, cX, cY, Theta)
85      ## Run challenge A for the First time
86      cX, cY, Theta = Challenge_A(cX, cY, Theta, tX1, tY1, target_offset, calibration_delay,
            sleeptime)
87
88      print(cX, cY, Theta)
89
90      ## Wait for the measuring delay
91      time.sleep(10)
92
93      ## Run challenge A the second time
94      cX, cY, Theta = Challenge_A(cX, cY, Theta, tX2, tY2, target_offset, calibration_delay,
            sleeptime)
```