HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS

master's thesis

Development of a motion control system for a driverless transport system

Submitted by:

Annika Gholam

Study program: Process Management Usability Engineering Industry 4.0
registration number: 1678060

Deadline: September 6, 2023

First examiner: Prof. Dr. Jens Hofschulte

second examiner Prof. Dr. Franz Kallage

task

University of Applied Sciences Hannover

master's thesis

Development of a motion control system for a driverless transport system

Annika Gholam
Matr. No. 1678060

Driverless transport systems are increasingly being used in production to transport the goods produced between the individual production sites. These transport systems must be highly maneuverable in order to be able to operate flexibly even in tight spaces.
Vehicles with Mecanum wheels are particularly well suited for this as they enable omnidirectional movements. However, they require more control effort.

The aim of this master's thesis is to develop a motion control system for a driverless transport system based on a Mecanum platform. To do this, first of all, a single-axis control of the Mecanum wheels and control of the kinematics must be implemented on the vehicle's microcontroller. In addition, a trajectory planning for a point-to-point movement is to be implemented on an external computer, which controls the microcontroller appropriately.

First examiner: Prof. Dr.-Ing. Jens Hofschulte

Second examiner: Prof. Dr.-Ing. Franz Kallage

statutory declaration

I hereby declare under oath that I have written this work on the subject of “Development of a motion control system for a driverless transport system” independently and have not used any sources or resources other than those specified and that all parts of the work that have been taken verbatim or in essence from other sources have been identified as such.

This work has not been submitted to any other examination authority in the same or a similar form.

Hameln, September 4, 2023 _____

ANNIKA GHOLAM

Table of contents

List of Figures

List of Tables Table 3-1:

Program code directory Program

abbreviations

abbreviation	Designation
API	Application programming interface
COM port	Communication port
DOF	degrees of freedom
FTF	driverless transport vehicle
FTS	driverless transport system
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
LAN	Local Area Network
LED	Light-emitting diode, light-emitting diode
PRM	Probabilistic Roadmaps
PoE	Power over Ethernet
PWM	pulse width modulation
REST	Representational State Transfer
RRT	Rapid Exploring Random Tree
USB	Universal Serial Bus
URI	Uniform Resource Identifier
WWW	World Wide Web

formulas

formula	Unit	Designation
	mm2	Area
	mm/s2	Maximum acceleration
,	mm/s2	Maximum acceleration of a component
0	-	Homogeneous transformation matrix from the coordinate system of the vehicle to the reference coordinate system
0	-	Homogeneous transformation matrix of the reference coordinate system to the vehicle's coordinate system
	1/s	proportionality factor translational
	1/s	proportionality factor rotational
	-	rotation matrix
	-	elementary rotation around the y-axis
	-	elementary rotation around the z-axis
	mm	Route
	mm	total distance
,	mm	total distance of a component
	mm	vehicle position
	mm	target position vehicle
'	mm/s	speed
'	mm/s2	acceleration
	s	Time
	s	acceleration time
	s	Time from which braking begins
	s	total time
	mm/s	speed

	mm/s	Maximum speed
,	mm/s	Maximum speed of a component
	mm	translational speed
	mm/s	velocity in x-direction
	mm/s	speed in y-direction
	mm	x-coordinate of the vehicle
	mm	x-target coordinate of the vehicle
	mm	y-coordinate of the vehicle
	mm	y-target coordinate of the vehicle
ÿ	mm	Distance difference between vehicle and target position
ÿ	mm	x-portion of the distance difference between vehicle and target position
ÿ	mm	y-part of the distance difference between vehicle and target position
ÿ	mm	position deviation in x-direction
ÿ	mm	position deviation in y-direction
ÿ	wheel	angular deviation
	wheel	target orientation
	wheel	orientation of the vehicle
	rad/s	angular velocity

1 Introduction

As industrialization progresses, the way products are manufactured is constantly changing. From mechanization in the 18th century, to electrification in the 19th century, to automation and digitalization from the 1970s onwards, the type of production has continued to evolve. The last stage of industrialization, "Industry 4.0", focuses on the networking of factories along the value chain. It describes a production chain in which people are almost completely replaced by machines and robots. People only have control and intervention functions.
During industrialization, motion control gained importance with electrification and, above all, automation. The term motion control refers to the control, coordination and monitoring of a movement. It is therefore a fundamental requirement of every machine or robot. The main task of motion control in robots is to receive and process robot programs and motion commands. With the help of interpolations, suitable temporal intermediate positions are created based on a desired target pose. These serve to describe the state of the system at different points in time (cf. Wenz, 2008: p.1).

With automation, especially with the networking of factories, transport tasks are largely taken over by driverless transport systems (FTS). FTS is characterized by floor-based transport through the use of driverless transport vehicles (FTF). These transport goods of different sizes, masses and shapes on the basis of a defined route network. They are automatically controlled and operated without human contact. They are therefore indispensable in an automated and networked factory (see Linde, 2023).

For this reason, universities and colleges are dealing with these topics. The University of Applied Sciences in Hanover also aims to deal with these topics. Therefore, in various laboratories, such as the robotics laboratory, students are familiarized with robots and the topics of "networking" and "automation" through various workshops and lectures.
The laboratories have, for example, programmable virtual factories, miniature production systems and various robots. Here, students have the opportunity to develop an understanding of the programming and functioning of such systems.

The result of this work will become part of the robotics laboratory and show students what options there are for controlling the motion of an AGV. The aim of the task is to develop a motion control system for an AGV for a point-to-point movement. In addition, this work will generate autonomous path planning. The determined points of the path will be traveled using a motion control system for a multi-point movement. The university will provide AGVs and a tracking system for this purpose. To implement this task, the procedure and the initial situation will first be considered.

When considering the initial situation, we analyze how the vehicle is constructed, what components it consists of, how the tracking system works, etc. Then we define the requirements for the motion control. In the next step, the interfaces are examined and, if necessary, the systems are expanded to include suitable interfaces for communication. This is followed by programming a trajectory and path plan. Trajectory planning is carried out for a point-to-point connection and is generated by interpolating intermediate poses. Various path-finding algorithms are examined for path planning. The aim is to enable autonomous path planning.

An initial theoretical consideration of the necessary content is not included in this work. The required theoretical information is included in the relevant chapters and is intended to supplement the practical content at appropriate points. The result of this work should be the program codes for the motion control. For better readability, the generic masculine form is used in this work. The personal terms used in this work refer to all genders, unless otherwise indicated.

2 Approach to the development of movement control

The procedure for implementing the motion control is based on the agile project management method using *Scrum.* The Scrum method originates from software development, but can be adapted to many other areas. It is a framework for agile project management and therefore does not specify any specific techniques, but rather general conditions such as project roles and a process flow. (cf. Preußig, 2020: pp. 135-142;

Wirdemann & Mainusch, 2017: pp. 28-32)

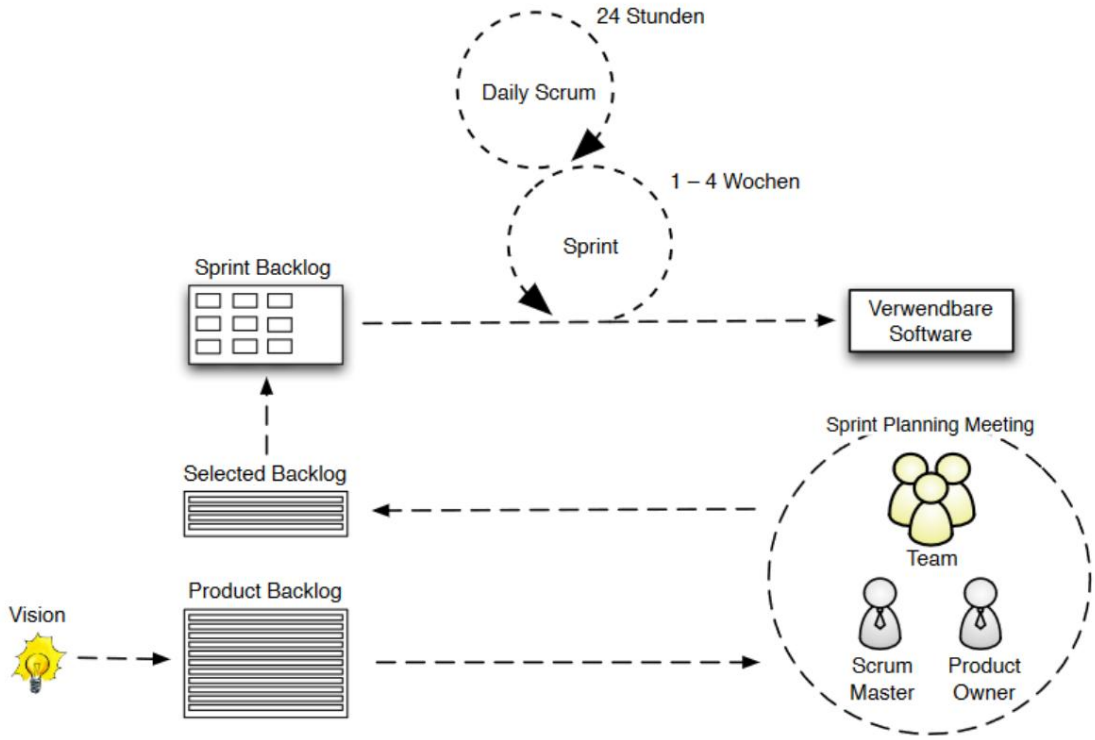


Figure 2-1: *Scrum* process flow
(Wirdemann & Mainusch, 2017: p.30)

Figure 2-1 shows the process flow of a *Scrum .* At the beginning of the process there is the vision or idea of a product. From this idea an objective or task is defined. On this basis the *product owner* creates a *product backlog.* The *product owner* is the visionary and the owner of the product to be developed. The *product backlog* contains the total tasks or requirements to be implemented and prioritizes them. It is therefore the first and elementary artifact of the process. The next step is the *Sprint Planning Meeting. The Scrum Master, Product Owner* and the team come together at this meeting . The *Scrum Master* is responsible for compliance with the Scrum process. The *product owner* is the subject matter expert for the requirements

responsible. During the meeting, the goal of the first *sprint* is determined and a *selected backlog* or a *sprint backlog* is compiled. A *sprint* corresponds to an iteration and can last from one to four weeks. The *sprint backlog ,* in turn, records the requirements and tasks to be implemented for this iteration. These tasks are carried out within the *sprint*
processed so that at the end of a *sprint* an increment, a partial product, is created. During a *sprint,* the project team meets daily at a fixed time to plan the respective daily tasks and to give feedback on the results and tasks already completed. After a *sprint* has been processed, the next *sprint* is planned in another *sprint planning meeting .* This is repeated until the project has ideally been successfully completed and a final product is available. (see Wirdemann & Mainusch, 2017: p. 25-31)

Based on this process, after considering the initial situation, a *product backlog* is also created and individual *sprints* and *sprint backlogs* are developed from this. However, this work is an individual work. Therefore, the author takes on the roles of both the *Scrum*

Masters, the project team and partly also the role of the *product owner.* Only the University of Hanover as the owner and task assigner also takes on the role of the *product owner* .

In order for the *Product Owner* to define a suitable goal, the initial situation must be known. Therefore, this is analyzed in the first step. On the basis of this analysis, the *Product Owner* can
then formulate his *product backlog* . The *product backlog* results in four sprints for this work. Three of these sprints are part of this work.

To complete this task, an AGV and a tracking system are provided. These systems are first introduced and the initial situation is described. Then the
Development of the motion control. This implementation is iterative. In this context, this means that the program code is expanded and improved with each sprint. The initial situation and *sprints* are briefly explained below .

The first step is to analyze the initial situation in order to create an understanding of the task.
The AGV consists of several components and is programmed using the Arduino environment *"Arduino IDE"* . Therefore, the individual components and the technical basics are examined, as well as the *Arduino IDE* and the structure of an Arduino program are described. The tracking system used is also explained in more detail.

The first *sprint* deals with the interfaces of the systems. It is examined which interfaces are available and which communication channels are suitable for data transfer between the systems. A suitable interface is then implemented. Before implementing the interface, it is first necessary to install and calibrate the tracking system. Part of this *sprint* is also the programming of the first simplified

Motion control in which only driving commands are converted into motion regardless of the location of the vehicle.

In the second *sprint,* the motion control is expanded to include path planning or trajectory planning. The start and destination are now recorded or specified and point-to-point control and movement are programmed. The control of the system and the transformation of coordinates are also considered.

The last *sprint* of this work deals with autonomous path search and path planning. In the second *sprint,* a linear movement between start and destination is generated. In this sprint, we look at which path leads to the destination if a direct connection is disrupted, e.g. by an obstacle. The generated path is followed using motion control for a multi-point movement.

Finally, there is a final conclusion. This summarizes the results of this work. In addition, the positive and negative aspects of the work are compared in a short discussion.
A brief outlook provides suggestions for further work.

3 initial situation

The University of Applied Sciences and Arts Hannover provides an FTS for the processing of this master's thesis. The main support for this work is the AGV used and a tracking system. The individual components and systems are described in more detail below.

3.1 Arduino

Different programs and programming languages are used to develop the motion control. On the one hand, the Arduino platform and on the other hand, the software
Visual Studio Code is required to create the program codes. In the following, the *Arduino* will be explained in detail be considered.

Arduino was developed in 2005 by the *Interaction Design Institute* Ivrea. The Italian institute recognized the problem that there were no easy-to-use and inexpensive microcontroller systems available on the market. So, in collaboration with electrical engineers, they created their own integrated system. The first *Arduino series was created.*
(cf. Bruhlmann, 2012)

Since then, *Arduino* has been continuously developed. *Arduino* adapts new technologies and components for the so-called *Arduino boards.* Since both the hardware and software are open source, *Arduino* spreads quickly on the market and finds its way into seminars and lectures at colleges and universities. Many code examples require little technical understanding, which makes *Arduino* very user-friendly.

very extensive projects can be developed. For this reason, *Arduino* is very popular. (cf. Margolis, 2012; Odendahl, Finn, & Wenger, 2010)

The term *"Arduino"* is usually associated only with the hardware components, but it also includes the software area of the same name. The physical world can only be controlled and perceived by working with both components. The development environment is called *"Arduino IDE".* This is used to program so-called *"sketches"* on the computer. These *sketches* are then converted by the integrated compiler into instructions that the *Arduino board* 's microcontroller can interpret. The *sketch* is transferred from the development environment on the computer to the *board* 's memory via a USB port using serial data transfer. (cf. Margolis, 2012;

Bruhlmann, 2012)

Figure 3-1 shows a screenshot of an empty *Arduino sketch* in the *Arduino IDE.* It can be seen that the program can be divided into two areas. Firstly, the *"void setup()"* area and secondly, the *"void loop()" area.* The *"void setup"* area contains those parts of the program that are called once when the program is started. The parts of the program in the *"void* loop()" area are executed permanently in a loop.

The keyword *“void”* indicates that a function is defined that does not return a value, so
a procedure with a specific activity is carried out. Furthermore, other functions can be defined outside of these areas. These can then be accessed in the *"void setup"* or in the *"void loop" .* (cf. Bräunl, 2022: p.54ff)

Figure 3-1: Empty Arduino sketch

The Arduino programming language is based on *"Wiring".* Similar to *Arduino,* this is also an open source programming environment for microcontrollers. The program code is written in a simplified form of the "C/C++" programming language. The microcontroller of the *Arduino board* does not understand the *C/C++* language , however, so it is necessary to translate the program code into machine language. This is made possible by the *compiler* (see Figure 3-2). However, this is a one-way connection. This means that the *compiler* can translate the C/C++ language into machine language, but conversely the program in machine language cannot be transformed back from the microcontroller into the Arduino language. (see Bartmann, 2012)

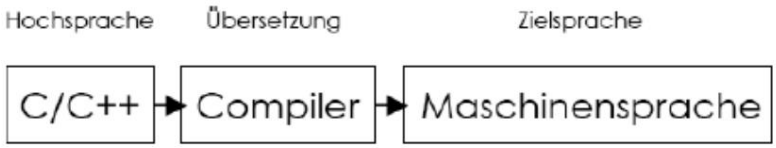


Figure 3-2: The compiler as interpreter
(Bartmann, 2012)

As soon as the *Arduino sketch* has been written and compiled, it can be transferred to the *board* via *upload* and the USB port . Once the *sketch* has been loaded onto the *board ,* the code written is immediately executed on it. There are inputs and outputs on the *board .* Sensors and actuators can be connected, controlled and monitored to these inputs and outputs (pins). The sensors are used to convert values and certain aspects from the real, physical world into electricity, which the Arduino hardware can then react to. The actuators, on the other hand, convert electricity that they receive from the *board* into mechanical movement. (see.
Margolis, 2012)

The term *“boards”* itself refers to circuit boards that are equipped with electrical components such as circuits, resistors, capacitors, etc. Furthermore, these circuit boards are equipped with various connections and interfaces. These can be used to establish a connection to the physical world. (see Bruhlmann, 2012)

Figure 3-3 shows the standard board, the *Arduino Uno.* This runs on the ATmega328 microcontroller and has 14 digital input/output pins and six analog pins. Pin 0 and pin 1 are also known as RX/TX pins. These are used for serial data transmission. In addition to the pins, the board has a memory of 32KB. (see Jänisch & Donges, 2017: p.8)

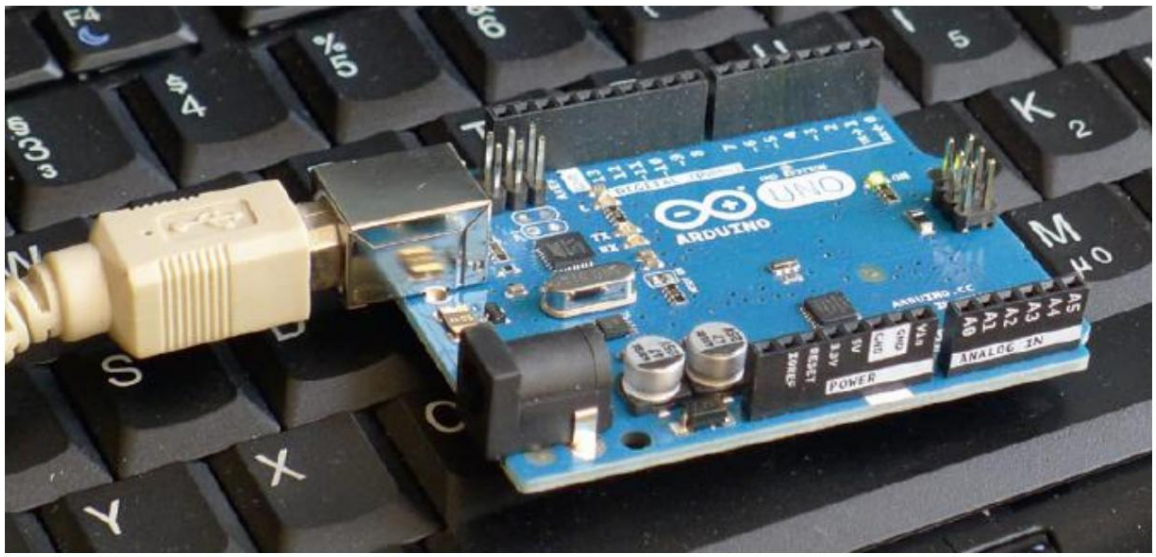


Figure 3-3: The *Arduino board* Uno (Jänisch & Donges, 2017: p.8)

3.2 Driverless transport vehicle with Mecanum wheels

The university provides an AGV with *Mecanum wheels* . This has four Mecanum wheels, four servo motors including encoders to operate the wheels and four ultrasonic sensors. These components are linked together via two Arduino boards (see.
Figure 3-4). The following subsections describe the respective components. In addition, scripts and libraries are provided by NEXUS, the company that distributes the FTF. Small sections of these libraries will also be explained in order to explain the functionality of the FTF.
to be able to understand.

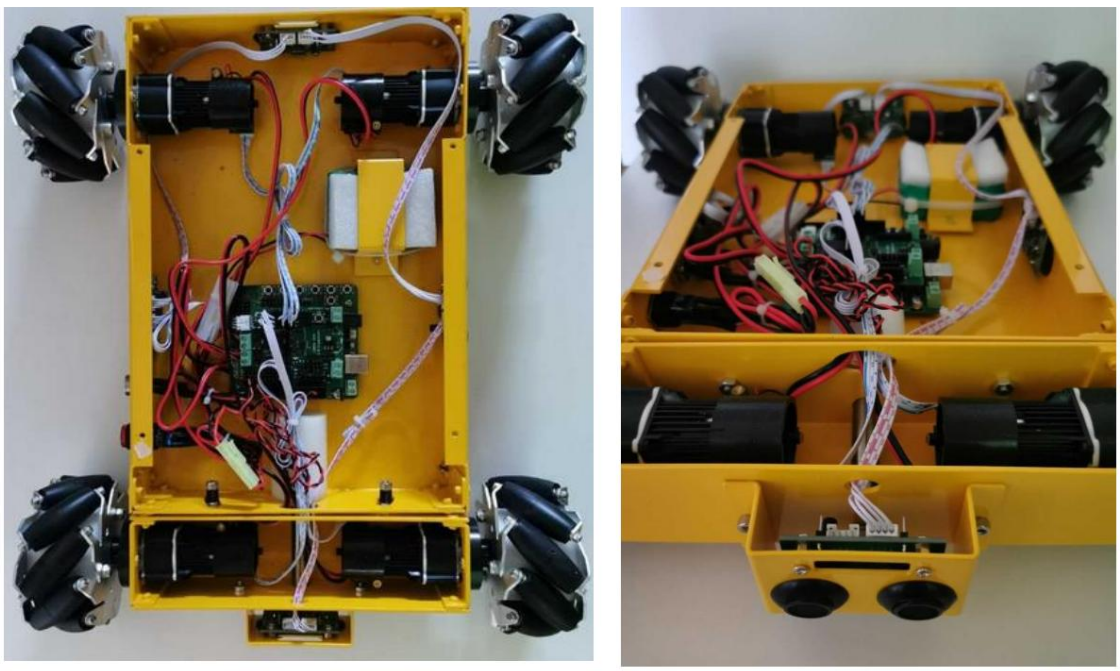


Figure 3-4: Images of the vehicle from two perspectives

3.2.1 Structure and kinematics of the Mecanum Wheels

The Mecanum wheel was invented in 1973 by Bengt Ilon. The idea of the Mecanum wheel is based on the concept of a central wheel with a large number of free rollers that are arranged at a certain angle on the circumference of the central wheel (see Figure 3-5). This creates a circular profile. The angled rollers create two force components when driving the wheel.
Part of the force is generated in the direction of travel, another part orthogonal to it. Opposing forces of individual wheels on a vehicle are compensated via the axles and frame. The angle between the roller axle and the central wheel can be selected arbitrarily, but is usually 45°. (see Kanjanawanishkul, 2015)

Figure 3-5: Mecanum wheel structure (Generation Robotics, 2023)

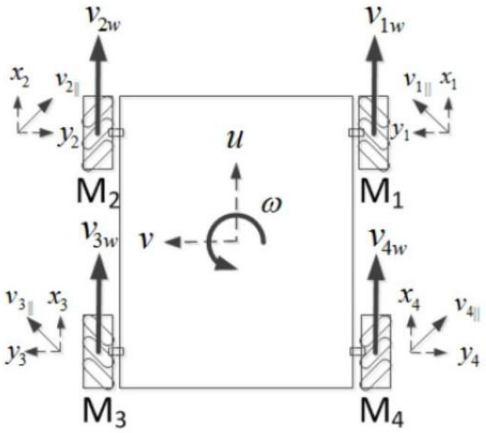


Figure 3-6: Velocity vectors of the Mecanum wheels of a vehicle (Kanjanawanishkul, 2015: p.292)

While only two degrees of freedom (DOF) can be realized with conventional wheels, the use of Mecanum wheels adds another degree of freedom: (cf. Baumgarten, 1990: p.6)

- 1. DOF: x-direction
- 2. DOF: y-direction
- 3. DOF: rotation around z-axis

Each wheel on a vehicle is driven by a separate motor. By controlling the direction of rotation and the speed of rotation of the wheels, any maneuver can be performed. The wheels are arranged rectangularly or squarely at each corner of the chassis (see Figure 3-6). Each wheel generates an individual force vector. The superposition of these vectors results in a force vector that moves the vehicle in the desired direction. The following equation can be used to describe how a vehicle with Mecanum wheels behaves and moves in space: (see Kanjanawanishkul, 2015: p.293)

$$\begin{bmatrix} \frac{1}{2} \\ \frac{1}{4} \end{bmatrix} = \begin{bmatrix} 1 & 1 & + \\ 1 \ddot{y}1 & \ddot{y} & \ddot{y} \\ 1 & 1 \ddot{y} & \ddot{y} \end{bmatrix} \begin{bmatrix} \\ \\ \end{bmatrix} \quad (3\text{-}1)$$

Here, $_{1,2,3,4}$ the translational speeds of the individual wheels and the speeds , , of the vehicle. These can be achieved by controlling the motors, whereas the constants and for the distance between the wheel axis and the center of the

Vehicle in the x and y directions and assume fixed values depending on the chassis (cf. Kanjanawanishkul, 2015: p.293)

Figure 3-7 shows different vehicle movements. The combination of the movements of the four wheels allows the vehicle to move in any direction at any angle.

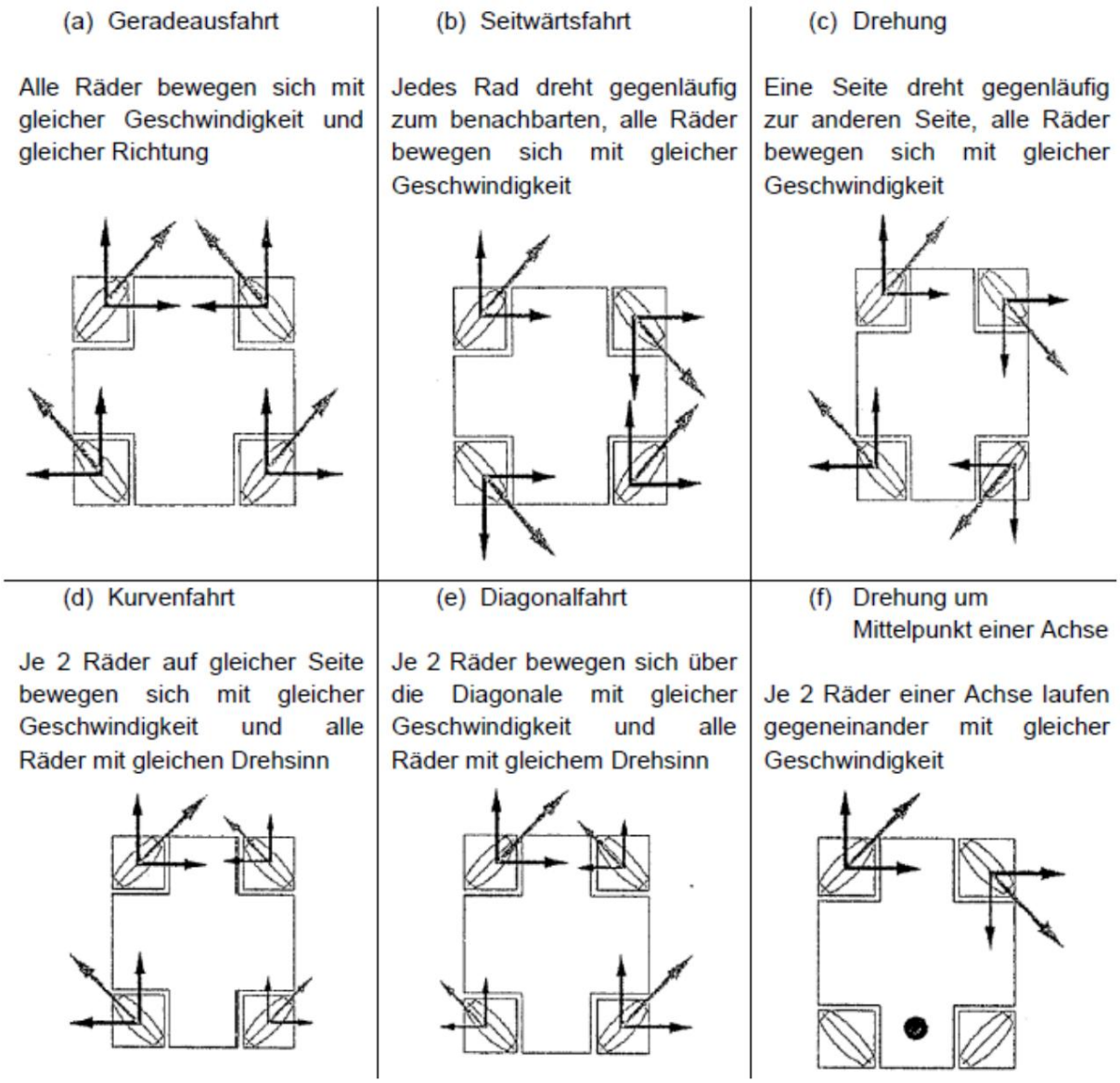


Figure 3-7: Direction of movement of a vehicle with Mecanum wheels according to Baumgarten, 1990: p.7f

3.2.2 Servo motors, PWM signal and timer for setting the PWM signal

The movement of the vehicle is generated by four servo motors. The term servo motor generally refers to a variable-speed electric motor drive that is operated in a controlled manner. A servo motor has a position sensor, which enables precise control of the angular position and the rotational speed. (cf. Kollmorgen, 2023; Probst, 2022: p.1f)

The servo motors are controlled via a PWM signal. The abbreviation PWM stands for pulse width modulation. PWM is a method in which an analog signal can be simulated at digital inputs and outputs. In contrast to analog signals, digital signals can only assume the value "on" and "off". With PWM, the duration of the
applied voltage is delivered to the outputs in accordance with a certain cycle. A series of repeating on and off pulses generates a square wave signal. The "on"
Time is the time during which a voltage is applied and the "off" time is the time during which the supply is switched off. Figure 3-8 shows three different PWM signals with a duty cycle of 10%, 50% and 90%. The duty cycle indicates how many percent the signal is switched on. If the supply is 5V and the duty cycle is 10%, for example, the result is an analog signal of 0.5V. (see Barr, 2001; Hirzel, 2023)

On an *Arduino,* the scale for the PWM signal goes from 0 to 255. This means that at 255 the duty cycle is 100% and the motors are controlled at maximum speed. At a value of 127, the duty cycle is 50% and the motors are operated at half speed. (see Hirzel, 2023)

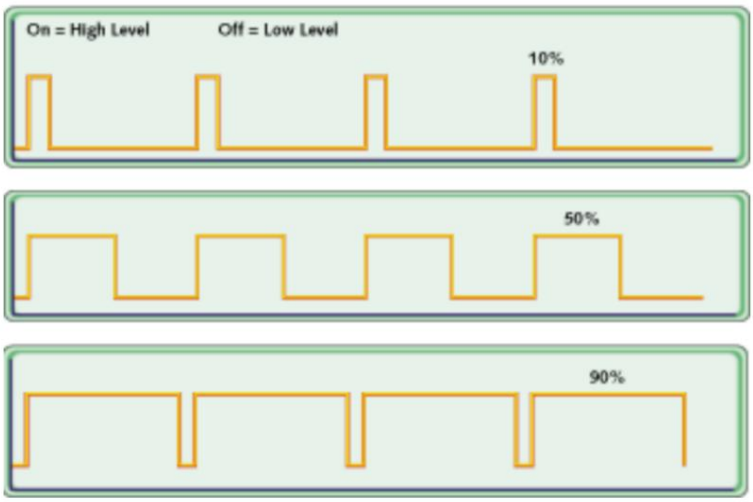


Figure 3-8: PWM signal with different duty cycles
(Barr, 2001)

The PWM signal can be set using *timers .* The ATmega328P microcontroller has three *timers.* These are called *Timer 0, Timer 1* and *Timer 2. Timers* are independent counters and integrated hardware in the *Arduino controller.* They work like a clock and are used to measure time events. Time functions such as *delay* or *millis,* but also the PWM functions require a *timer.* All *timers* are dependent on the *system clock,* which has a frequency of 16MHz in an ATmega328. Each timer has a *prescaler,* which generates the timer clock by dividing the *system clock* by a *prescaler factor* such as 1,8,64,256 or 1024. *Timer 0* is used for the timer functions such as *delay(), millis()* and *macros() ,* so *Timer 1* and *Timer 2* are used to set the PWM signal. Each *timer* has timer registers. If the bits of these registers are adjusted, the timer behavior is set individually. Figure 3-9 shows a part of the timer registers using *Timer 0* as an example.

(cf. Atmel, 2015: pp.74-88)

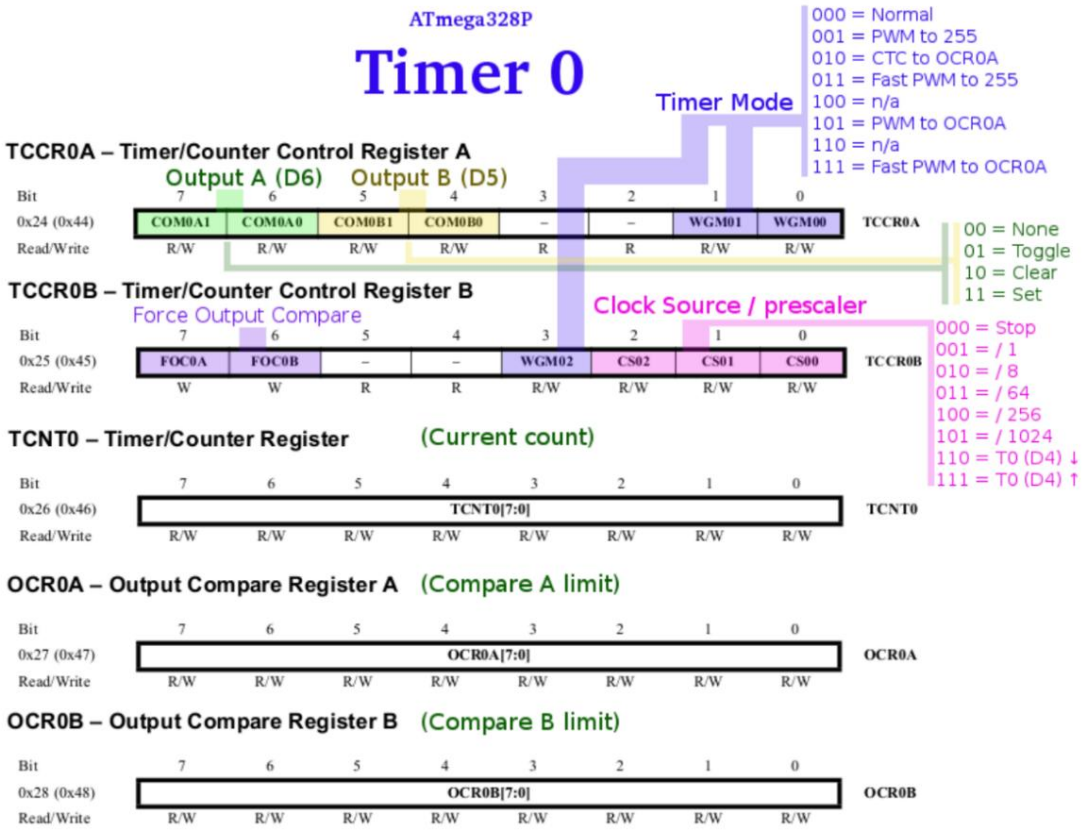


Figure 3-9: Timer register for timer 0
(Gammon, 2015)

The first timer register is the *Timer/ Counter Control Register TCCRx.* This register is used to configure the *timer .* The *prescaler* is set here. The *prescaler* affects how fast the *timer* counts. If a larger *prescaler* is selected, the *timer* will count slower. The register is again divided into *TCCRnA* and *TCCRnB.* These registers contain the main control bits for the *timers.* The main control bits include the WGM bits, the CS bits and the *COMnA/ COMnB bits. WGM* is the *Waveform Generation* Bit. This functions like a mode switch.

Different modes can be selected or set. They determine how the *timer* counts and how it reacts when it reaches certain values. *CS* are the *Clock Selected bits.* These are for setting the *prescaler.* The COMnA/COMnB are the Compare Match Output A/B Mode bits. These modes influence the behavior of the *timer* 's output pins (A or B) when a comparison is made between the *timer* and a comparison value. They can switch the output on, off or invert it. The comparison value is specified in the *Output Compare Register (OCRnA/ OCRnB) .*

Another register, the *Timer/ Counter Register (TCNTx),* stores the current value of the *timer.*
When the *timer* ticks or counts, this value is incremented. It is as if the *timer* is keeping track of time by counting the number of ticks. The *timer* continuously compares the current value in the *timer/ counter register TCNTx* with the value in the *output compare register OCRnA/ OCRnB*
stored value. If they match, a predefined event can be triggered. The x or n in the register names can take the value 0, 1 or 2, depending on the *timer* .

The PWM mode is a special mode of the *timer.* This is set via the WGM bits. The mode is often used to control analog signals such as brightness or speed. The *timer* counts repeatedly from 0 to a certain value (e.g. 255). Depending on when the *timer* is triggered, the output state changes. This allows the ratio of on to off states (duty *cycle)* to be controlled. The higher the value in the *Output Compare Register ,* the higher the *duty cycle.* The *timer* can be configured using the microcontroller's data sheet . It contains information about how the registers

must be set to achieve the desired functionality. Equation 3-2 and equation 3-3 calculate the desired frequency or *duty cycle* in PWM mode, where "N" represents the *prescaler factor* and "MAX" the maximum value in the *output compare register* . The variable is the clock frequency. For an ATmega328, this is 16MHz. (cf. Atmel, 2015: p.74-88)

$$= \frac{}{2ÿ\ ÿ256} \tag{3-2}$$

$$= \frac{(\quad +1)}{256} \tag{3-3}$$

3.2.3 *Dual UltraSonic* ultrasonic sensors **and RS485 bus system**

Since the vehicle has four ultrasonic sensors connected in series, communication or serial data transfer between the sensors is required. This serial data transfer takes place via the RS485 bus system. Before looking at the bus system, however, we will take a brief look at how an ultrasonic sensor works.

Ultrasonic sensors work using sound waves in the ultrasonic range. These sound waves have a frequency of over 20 kHz and are therefore not perceptible to the human ear. The principle of ultrasonic measurement is widespread in nature. For example, bats can fly in the dark and search for food using ultrasound. The special feature of sound waves is their high frequency with a short wavelength, which makes them similar to light waves. In technology, ultrasonic sensors are used for distance measurement, among other things. For example, the distance to an object or a person approaching the sensor can be measured.

The ultrasonic sensor (see Figure 3-10) can measure distances from 4cm to 3m. To do this, the sensor sends an ultrasonic pulse that is reflected off an obstacle such as a wall. After bouncing, the pulse returns to the sensor. The distance is calculated based on the time elapsed between the pulse being sent and its return. (see Ericson, 2022)

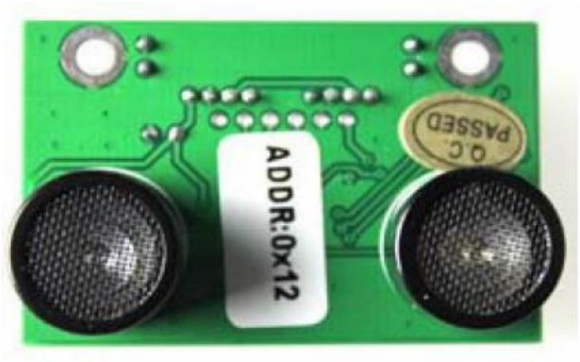


Figure 3-10: Dual Ultra Sonic ultrasonic sensor
(Nexus Robot, 2012: p.23)

The four *dual ultrasonic sensors* communicate via the RS485 bus system. RS485 is a method for serial communication between computers or other devices. The system is characterized by simple wiring, long transmission distance and high
Reliability. In addition, several devices can be connected to the same bus. This means that several nodes can be connected to each other. (see Bies, 2017)

The communication network consists of several transceivers. These are connected via two twisted wires. The basic idea of the interface is therefore a differential
Data transmission. This means that a signal is transported over two wires. The first wire transmits the original signal, while the second transmits an inverse copy. This type of data transmission ensures a high resistance to interference. The data transmission is

via the data transmission protocol. The protocol regulates how the data packets are sent and received. With an RS485 interface, not all devices can send and receive data at the same time. This would lead to a conflict between the senders and a collision of data packets could occur. Therefore, the commands are sent by a defined master (see Figure 3-11). The other devices are called slaves. These receive the sent data via RS485 ports and can respond on the master's line depending on the information sent. Weis, 2021

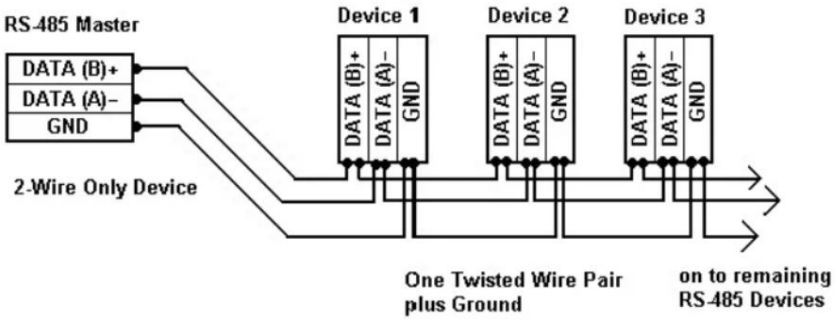


Figure 3-11: Data transfer from the master to the slaves via RS485
(Weis, 2021)

3.2.4 Arduino board

The central element of the vehicle is the *Arduino board,* which consists of the Arduino board V1.1 with ATmega328P microcontroller and an expansion board. It is responsible for processing signals and controlling the motors and sensors. The hardware components are explained in more detail below, followed by a look at how the individual components are connected.

Figure 3-12 shows the Arduino Board V1.1. This shows all input/output pins and the connections on the board. It includes: (see Nexus Robot, 2012: p.2f)

- A motor power input terminal (7V -12V)
- A servo power input terminal (4V – 7.2V)
- Two DC motor terminals
- One analog port with eight analog input pins
- A digital I/O port with 13 digital pins, of which pins 4,5,6,7 are motor control pins can be used
- A USB port
- A power connection
- An APC220 connector: Provides the possibility of a module for wireless communication to connect
- APC220 *Wireless Selection Jumper:* This is required to enable wireless communication via an APC220 module. If this is removed, no communication can be established via it.

The digital pins can be used as inputs and outputs. They operate at a voltage of 5V. Some of these pins have a special function. Pin 0 (RX) and pin 1 (TX) are used to receive (RX) or transmit (TX) serial data. The two pins of the *board* are connected to the USB port. There are also two external *interrupt pins,* pin 2 and pin 3. These can be used for *hardware interrupts* (see chapter 3.2.5 *PinChange).* Pins 3, 5, 6, 9, 10 and 11 offer an 8-bit PWM output. There is also an integrated LED which is connected to pin 13. If the pin has the value HIGH, the LED is on. If the pin has the value LOW, it is off.

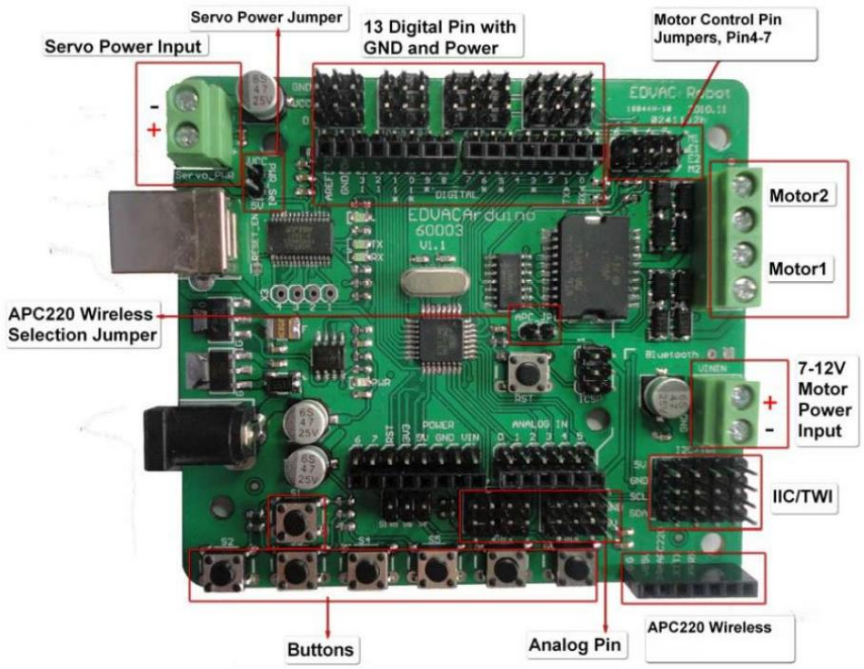


Figure 3-12: Arduino board V1.1
(Nexus Robot, 2012: p.2)

The *Arduino IO Expansion Board* can be used for serial communication via RS485 and for connecting two additional motors (see Figure 3-13). The expansion board can be easily plugged onto the *Arduino board V1.1 .* The *board* provides additional digital and analog pins, some of which also support PWM. The following connections are integrated on the board: Nexus Robot, 2012: p. 4f

- A connection for XBee/Bluetooth Bee
- An RS485 output to connect RS485 devices
- A connection for an APC220 module

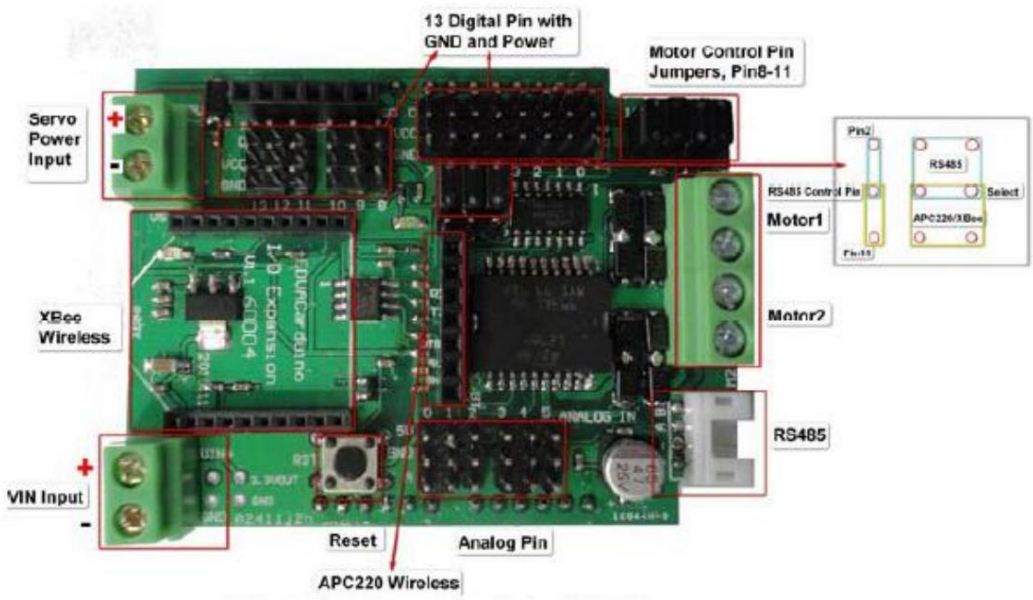


Figure 3-13: Arduino IO Expansion Board
(Nexus Robot, 2012: p.4)

The motors, sensors and battery are connected to the *Arduino boards* (see Figure 3-14).
The motors are each connected to a DC motor terminal. Motor 1 and Motor 2 are connected to the *Arduino Board V1.1* , while Motor 3 and Motor 4 are connected to the expansion board. The motors are each connected to the board via four additional pins, of which 2 pins are for the encoder, one for the PWM signal and another for the direction indication (DIR).
The encoder pins are divided into phase A and phase B. The pin connected to phase A of the encoder is used to enter commands to the motor. The pin connected to phase B of the encoder is used to output the motor signal. A 12V battery is connected to the board via the *Motor Power Input* connection . This allows the vehicle to drive without a power connection. The sensors are connected to the RS485 output. Only the first sensor is connected to this. The other sensors are then connected in series.

switched on.

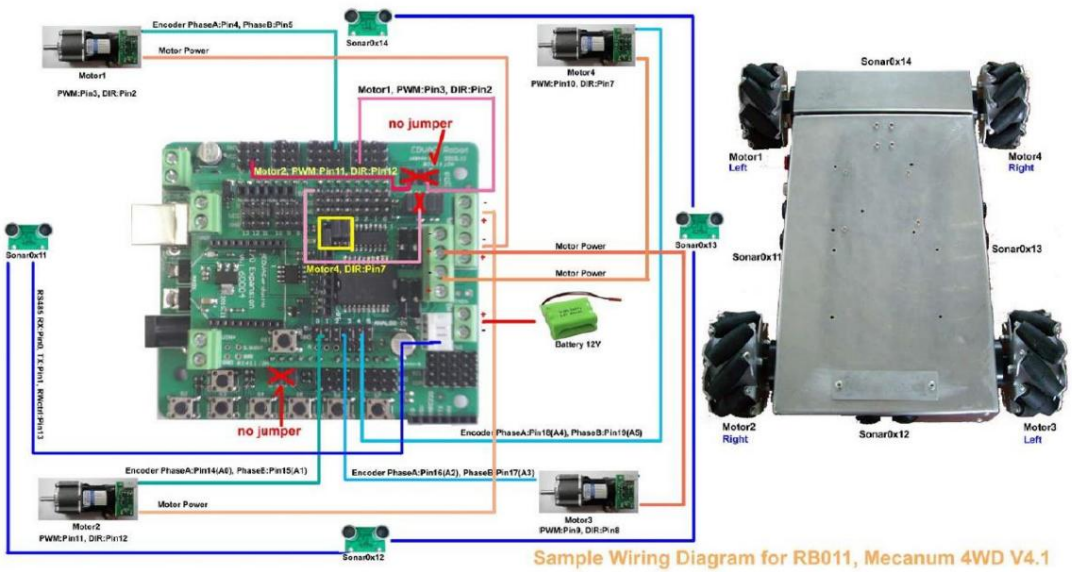


Figure 3-14: Connecting the components to the Arduino board
(Nexus Robot, 2012: p.114)

Table 3-1 lists the pin assignment of the individual pins for the sake of clarity. This contains the respective pin number and also indicates the function of the respective pin in relation to the connected component. Finally, the respective component is assigned to the pin.

Table 3-1: Pin assignment

Pin Nr.	Funktion	Komponente
Pin 0	RX	Rs485 Sonar
Pin 1	TX	Rs485 Sonar
Pin 2	Dir	Motor1
Pin 3	PWM	Motor1
Pin 4	Encoder Phase A	Motor1
Pin 5	Encoder Phase B	Motor1
Pin 6	-	-
Pin 7	Dir	Motor4
Pin 8	Dir	Motor3
Pin 9	PWM	Motor3
Pin 10	PWM	Motor4
Pin 11	PWM	Motor2
Pin 12	Dir	Motor2
Pin 13	RWctrl	Sonar
Pin 14	Encoder Phase A	Motor2
Pin 15	Encoder Phase B	Motor2
Pin 16	Encoder Phase A	Motor3
Pin 17	Encoder Phase B	Motor3
Pin 18	Encoder Phase A	Motor4
Pin 19	Encoder Phase B	Motor4

3.2.5 Libraries and demo script

The Arduino environment can be extended by using libraries. These libraries provide additional functions for use in *sketches,* such as working with specific hardware or manipulating data. A variety of standard libraries are already stored in the *Arduino IDE* and can be imported. However, you can also create and download your own libraries. (see Arduino, 2023)

The libraries are often divided into a header file (header.h) and a source file *(sourcefile.cpp) .* The header file contains declarations of data types, structures, functions or classes, for example. It indicates that these exist, but not how they work.
These declarations are used in the source file and then implemented. For example, this is where you define how a declared function works. Splitting it into two separate files has the advantage of significantly reducing compilation time, as the source file is already available in compiled form on the platform and the compiler only calls up the declaration of the header file. A library or a header file is called by declaring it using *#include* is added in the *sketch .* The use and structure of the libraries will become clearer when looking at the following libraries and the demo script.

In addition to the AGV hardware, Nexus also offers libraries and a demo script for moving the vehicle. The demo script integrates the *libraries MotorWheel.h, Omni4WD.h, PID_Beta6.h, sonar.h* and *PinChangeInt.h, among others.* The functionality and use of these libraries are covered in the respective sections.

Pin Change Interrupt

With this library, it is especially important to understand what a *pin change interrupt* is and what it is used for. Therefore, the script itself will not be examined in detail, but only explained how it works.

Microcontrollers often have a multitude of tasks to handle and are therefore quite busy. Nevertheless, external events and internal timing events must be executed in a controlled manner. The use of *interrupts is suitable for this. Interrupts* are a method of interrupting the current program execution. The program is stopped and its data put aside so that it can be continued at the same point later. Then the program code that refers to the *interrupt* is executed , the so-called *Interrupt Service Routine ISR* (see Figure 3-15). As soon as this is finished, the program continues where it left off. As a rule, two types of interrupt functions are distinguished. There are both *hardware interrupts* and *software interrupts.* Hardware *interrupts* are triggered by external signals. *Software interrupts* are controlled by internal signals such as *timers* or software-related events. (see DroneBot Workshop, 2022; mwwalk, 2014)

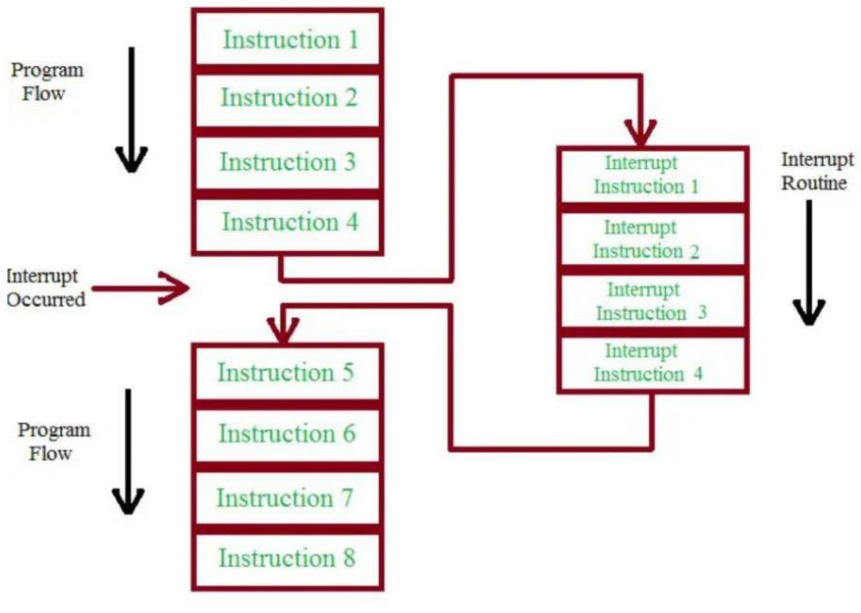


Figure 3-15: Interrupt Service Routine
(Carrasco, 2021)

Furthermore, interrupts are divided into three types:

1. *Hardware interrupts* - External *interrupt signals* on specific pins
2. *Pin Change Interrupts* - External *interrupts* on each pin, grouped into ports
3. *Timer interrupts* – Internal timer-generated *interrupts* that are manipulated in software

Pin change interrupts and *timer interrupts* belong to the higher-level category of *software interrupts.* As the name suggests, the *pin change interrupt library* describes the second type of *interrupt,* " *pin change interrupts".* Since *hardware interrupts* are limited to just a few pins and often more pins have to be used, the number of pins for using *interrupts* has to be expanded. The *pin change interrupt library* defines functions that specify how a *pin change interrupt* is triggered, on which pins it is triggered, how it is cancelled, etc., so that the *Arduino* program or other libraries can then use the pin change interrupt functions.

PID Beta6

The *PID Beta6* implements the functions for a PID controller. Similar to the *PinChange* interrupt library, however, the functions will not be examined in detail here, but rather a general explanation of what a PID controller is.

A PID controller is used to compensate for a difference between a measured value and a target value so that there is no permanent control deviation. Figure 3-16 shows a simple control loop. This consists of a controller and a controlled system. In this case, the controller is a PID controller and the controlled system is the vehicle or the individual motors of the vehicle. First, a target value, e.g. a target speed, is specified. This is compared with the actual value. A control deviation is determined from these two values, which is passed on to the controller. The controller sends a manipulated variable to the controlled system as an output so that the deviation between the target and actual is reduced. At the end, an actual value is determined and fed back and the control starts again from the beginning. (see RN-Wissen, 2023)

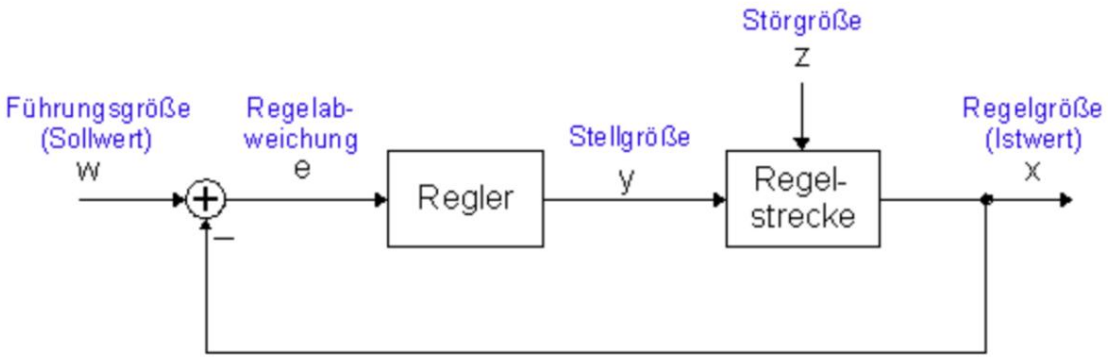


Figure 3-16: Simplified control loop
(Tobola, 2014)

The problem is best illustrated with an example. For example, a speed of 50 km/h is to be achieved. This means that the target value is 50 km/h. However, an actual value of 45 km/h is measured, resulting in a control deviation of 5 km/h. A control variable is then determined using the control parameters. This can, for example, take the value 7 km/h and represents the necessary change. The control variable is fed to the control system and processed in such a way that the actual value of the speed is obtained again. This can, for example, take a value of 51 km/h.

The controller should react to a deviation as quickly as possible and with little or no overshoot. The PID controller consists of three components: the proportional component (P component), the integral component (I component) and the differential component (D component). Each of these components has different functions. The P controller acts proportionally to the control deviation. It reacts immediately to any change in the control deviation, but the control deviation cannot be completely eliminated. The I controller, on the other hand, reacts very slowly, but can adjust the actual size exactly to the target size. The D controller reacts to the change in the control deviation over time. In the case of constant deviations, they therefore do not cause any control intervention. The sole use of a D controller is unsuitable. In combination as a PID controller, the D component enables a fast, overshoot-free approach to the setpoint. The control behavior can be set via the control parameters, with each controller component being adjusted by the respective control parameter. (see RN-Wissen, 2023)

sonar

The *sonar* library is used to describe the functionality of the communication protocol of the ultrasonic sensors. The functionality of the ultrasonic sensors and their communication path were already described in chapter 3.2.3. The sensors communicate with each other via an RS485 interface. A communication protocol is used to specify, for example, what the address of a sensor looks like or how a measurement is triggered. This is described within the library. Various commands are defined here. The following commands and functions are described, among others: (cf. Nexus Robot, 2012: p.27f)

- *Set Device Addres (SONAR::setAddr(addr)):* Here a sensor is assigned a specific address assigned
- *Trigger measurement (SONAR::trigger()):* Triggers a trigger by which a distance is measured. However, this function does not return any data.
- *Get distance (SONAR::getDist()):* This function returns the measured distance value.

MotorWheel and Omni4WD

The two central libraries are the *MotorWheel* library and the *Omni4WD* library . They

On the one hand, they use the functions of the other libraries and, on the other hand, they describe functions for controlling and regulating the motors. Like the other libraries, these two are also divided into *MotorWheel.h* and *Motorwheel.cpp* or *Omni4WD.h* and *Omni4WD.cpp.*
While the header file of most libraries mostly declares the functions, the header file *MotorWheel.h* also defines parameters for the chassis, wheels, control, motor, etc. The definitions of the parameters for the motor, PWM signal and maximum speed are given as examples in program code 3-1. Since the motors are Faulhaber motors, the parameters are selected accordingly. In addition, a maximum PWM of 255 and a maximum speed of 8000rpm are specified. These values can be found in the vehicle's data sheet.1

```
/*for arduino*/
#else
    #define  MAX_PWM 255

#endif

#ifdef _NAMIKI_MOTOR
    #define  TRIGGER CHANGE
    #define  CPR 4  // Namiki motor
    #define  DIR_INVERSE
    #define  REDUCTION_RATIO 80
#else
    #define  TRIGGER RISING
    #define  CPR 12 // Faulhaber motor
    #define  DIR_INVERSE !
    #define  REDUCTION_RATIO 64
#endif

#define  MAX_SPEEDRPM 8000
```

Program Code 3-1: Defining Vehicle Parameters

1 The data sheet can be found in Appendix A

The source file *MotorWheel.h* describes the functions for the individual motors as well as the *interrupt* and control functions. These functions are described in the source file *Omni4WD.cpp*
This library creates a class that links the four motors together so that functions can be introduced that apply equally to all motors. Two of these functions are briefly explained below.

The *Omni4WD::setCarMove(…)* function is described in program code 3-2 . This function specifies three values. Firstly, a speed *speedMMPS,* secondly an angle *rad* and finally an angular velocity *omega.* The speed specifies a value between 0 and 255, the angle is used to determine the direction of travel and the angular velocity enables rotation around the vehicle axis. The respective shares are then calculated for each individual wheel.

```
int Omni4WD::setCarMove(int speedMMPS,float rad,float omega) {
    wheelULSetSpeedMMPS(speedMMPS*sin(rad)+speedMMPS*cos(rad)-omega*WHEELSPAN);
    wheelLLSetSpeedMMPS(speedMMPS*sin(rad)-speedMMPS*cos(rad)-omega*WHEELSPAN);
    wheelLRSetSpeedMMPS(-(speedMMPS*sin(rad)+speedMMPS*cos(rad)+omega*WHEELSPAN));
    wheelURSetSpeedMMPS(-(speedMMPS*sin(rad)-speedMMPS*cos(rad)+omega*WHEELSPAN));

    return getCarSpeedMMPS();

}
```

Program code 3-2: Definition of the function "SetCarMove"

The *setCarMove()* function is then used to define the movement or driving directions. This defines functions for driving forward, driving backwards, driving sideways to the left and right, driving diagonally (top left, bottom left, top right, bottom right), rotation around the vehicle axis to the left and right, and stopping the driving movement. Program code 3-3 describes driving forward. For this, a status is set to advance . The driving direction and speed are then defined using *setCarMove* . An angle of Pi/2 is required for the vehicle to move forward. Figure 3-17 shows at which angle the vehicle moves in which direction.

```
int Omni4WD::setCarAdvance(int speedMMPS) {
    setCarStat(STAT_ADVANCE);
    //wheelULSetSpeedMMPS(speedMMPS,DIR_ADVANCE);
    //wheelLLSetSpeedMMPS(speedMMPS,DIR_ADVANCE);
    //wheelLRSetSpeedMMPS(speedMMPS,DIR_BACKOFF);
    //wheelURSetSpeedMMPS(speedMMPS,DIR_BACKOFF);
    //return wheelULGetSpeedMMPS();
    return setCarMove(speedMMPS,PI/2,0);
}
```

Program code 3-3: Definition of the function "setCarAdvance"

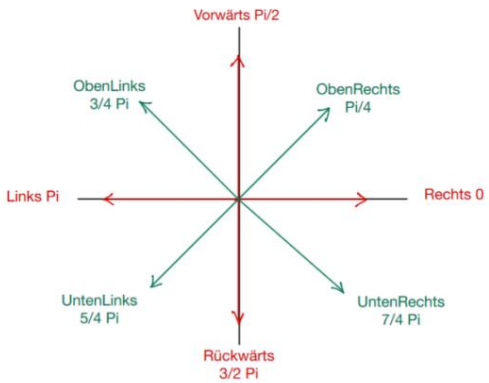


Figure 3-17: Direction of travel depending on the angles

demo script

The company Nexus Robot provides a demo script for the vehicle. Excerpts from this will be briefly explained here.2

The first excerpt can be found under program code 3-4. This shows how an object is created for motor 4. First, the interrupt function *irqISR(irq4, isr4)* is created for motor 4. This works on the basis of pulses generated by the encoder. Next, the object *MotorWheel wheel4(10,7,18,19&irq4)* is created for the motor itself. The information for the pins is in brackets. Pin 10 is for PWM, pin 7 is the direction pin, pin 18 is for encoder phase A and pin 19 is for encoder phase B. The other 3 motors are created in a similar way. Next, an object *Omni4WD Omni(&wheel1, &wheel2, &wheel3, &wheel4)* is created. This object integrates all motors, making it possible to create or call functions that affect all four motors at the same time. In the next step, the ultrasonic sensors are assigned their address and a *buffer distBuf[4]* is created in which the measured distances of the sensors can be stored. In the last part of the program code snippet, the function *sonarUpdate()* is defined. In this, a running variable *sonarCurr* is first defined and set to the value 1. As soon as the running variable takes on the value 4, it is always reset to the value one, otherwise it is increased by one. If the running variable *sonarcurr==1 ,* the distance of the ultrasonic sensor with the address 0x12 is written to the *buffer* at position 1. This can be output on a serial monitor using *showDat() .* A *trigger* is then set to initiate a new measurement. If the running variable *sonarcurr==2 ,* then

The same is done for the sensor *sonar13 ,* with its distance values transferred to position 2 of the *buffer*
The same principle applies to the remaining two sensors. Finally, the value of the control variable *sonarcurr* is returned.

```
43  irqISR(irq4,isr4);
44  MotorWheel wheel4(10,7,18,19,&irq4);
45
46  Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4); //Alle Motoren inbegriffen
47
48  SONAR sonar11(0x11),sonar12(0x12),sonar13(0x13),sonar14(0x14);
49  unsigned short distBuf[4];
50
51  unsigned char sonarsUpdate() {
52      static unsigned char sonarCurr = 1;
53      if(sonarCurr==4) sonarCurr=1;
54      else ++sonarCurr;
55      if(sonarCurr==1) {
56          distBuf[1]=sonar12.getDist();
57          sonar12.showDat();
58          sonar12.trigger();
59      } else if(sonarCurr==2) {
60          distBuf[2]=sonar13.getDist();
61          sonar13.showDat();
62          sonar13.trigger();
63      } else if(sonarCurr==3){
64          distBuf[3]=sonar14.getDist();
65          sonar14.showDat();
66          sonar14.trigger();
67      } else {
68          distBuf[0]=sonar11.getDist();
69          sonar11.showDat();
70          sonar11.trigger();
71      }
72
73      return sonarCurr;
74  }
```

Program code 3-4: Declaration of the motors and sensors as well as function *sonarsUpdate()*

2 The complete demo script can be found in Appendix B

The next code snippet describes how to create a function for driving ahead.
First, the status is checked. If the status does not correspond to the status “drive ahead”, but e.g. "Reverse" means that the motors are slowly stopped. Otherwise, the direction of the vehicle is set to "forward" using the *Omni.setCarAdvance* function and the speed and duration for how long the vehicle should travel in this direction is set using the *Omni.setCarSpeedMMPS(speedMMPS,300)* function, here 300ms. Functions for the other directions of travel or rotation are created in a similar way.

```
76  void goAhead(unsigned int speedMMPS){
77      if(Omni.getCarStat()!=Omni4WD::STAT_ADVANCE) Omni.setCarSlow2Stop(300);
78          Omni.setCarAdvance(0);
79          Omni.setCarSpeedMMPS(speedMMPS, 300);
80  }
```

Program code 3-5: Function “goAhead”

The next section of the demo script produces an array and a function. With the help of these, the direction of travel of the vehicle is determined depending on the sensors. The array *motion[16]* contains various functions for the directions of travel. For example, if position four of the array is called, the vehicle should move to the left (turnleft) . Which of these positions is called is determined using the function *demoWithSensors(speedMMPS,distance) .* There are two if loops within the function. The first ensures that the function *SonarUpdate is called* every 80ms.
is called. This function (see above) returns a value between 0 and 4. As soon as the value *sonarcurrent==4,* the second if function is called. Here, it is determined for each sensor whether the distances stored in *the buffer* are smaller than a specified distance. If this is the case for a sensor, a 1 is set for a byte. Depending on the sensor, this is positioned at a different location in the byte, so that the variable *bitmap* can ultimately take on a value between zero and 15. The *bitmap* then specifies which position in the array *motion[bitmap]* is selected, which in turn determines the direction of travel. Finally, the function *Omni.PIDRegulate()* is called.
which allows the control of the movement.

```
120 void(*motion[16])(unsigned int speedMMPS) = {goAhead, turnRight, goAhead, turnRight,
121     turnLeft, goAhead, turnLeft, goAhead,
122     rotateRight, rotateRight, turnRight, turnRight,
123     rotateLeft, backOff, turnLeft, allStop};
124
125 unsigned long currMillis=0;
126 void demoWithSensors(unsigned int speedMMPS,unsigned int distance) {
127     unsigned char sonarcurrent = 0;
128     if(millis()-currMillis>SONAR::duration + 20) {
129         currMillis=millis();
130         sonarcurrent = sonarsUpdate();
131     }
132
133     if(sonarcurrent == 4){
134         unsigned char bitmap = (distBuf[0] < distance);  //right
135         bitmap |= (distBuf[1] < distance) << 1;  // back
136         bitmap |= (distBuf[2] < distance) << 2;  // left
137         bitmap |= (distBuf[3] < distance) << 3;  // front
138
139         (*motion[bitmap])(speedMMPS);
140         Serial.println(bitmap);
141     }
142
143     Omni.PIDRegulate();
144 }
```

Program code 3-6: Array *motion* and function *demoWithSensors*

The last part of the source code describes the functions *void setup()* and *void loop().* In *void setup()* First, the frequency of the PWM signal is set (see chapter 3.2.2) and then the control pin *pinCtrl=13* is set via *SONAR::init ,* via which the sensors are called.
The PID controller is then activated and its parameters are set using the *Omni.PIDenable* function. The parameter for the P component is 0.35, for the I component 0.02 and for the D component 0. This means that the D component is omitted and the control is carried out using a PI controller. Finally, only the function *demoWithSensors* is called in the *void loop()* and the driving speed (160mm/s) and distance at which a sensor is considered to be "triggered" are specified.

```
146  void setup() {
147      delay(2000);
148      TCCR1B=TCCR1B&0xf8|0x01;    // Pin9,Pin10 PWM 31250Hz
149      TCCR2B=TCCR2B&0xf8|0x01;    // Pin3,Pin11 PWM 31250Hz
150
151      SONAR::init(13);
152      Omni.PIDEnable(0.35,0.02,0,10);
153  }
154
155  void loop() {
156      demoWithSensors(160,20);
157  }
```

Program code 3-7: Function *void setup()* and *void loop()*

3.3 OptiTrack tracking system

Tracking systems are used to locate and track physical objects.
For this master's thesis, an OptiTrack S250e tracking camera and an associated application program for reading the tracking data are provided (see Figure 3-18). This chapter first describes how the camera itself works. The application program is discussed in Chapter 5.3.1.

OptiTrack cameras use infrared LEDs to track passive round markers. The passive markers are coated with a reflective material. They reflect most of the light from the infrared LEDs back, allowing the camera to detect the position of the marker. A 2D position of a marker can be determined using one camera. However, if several cameras are used, a three-dimensional position can be determined by overlapping the images. The tracking system is very accurate, so that all types of movements can be recorded. If at least two markers are visible, the orientation of an object can be specified. However, as soon as a marker is covered, the tracking camera can no longer detect its position. The camera can track over 80 objects simultaneously. In addition, data processing in real time is possible for most applications. (see

NatrualPoint Inc., 2023)

Figure 3-18: OptiTrack S250e
(NatrualPoint Inc., 2012)

4 Product Backlog for Motion Control

After the initial situation and the functionality of the vehicle and tracking system
As explained above, the requirements for motion control should now be defined in a *product backlog*
The *product backlog* is contained in Table 4-1 and Table 4-2. It consists of the requirements for control and the prioritization of the requirements. This chapter does not go into detail about the individual requirements, but simply explains the prioritization and processing.

The *product backlog* is divided into four priority classes, with priority 4 being the highest priority and priority 1 being the lowest priority. The priorities are used to classify the requirements and assign them to the respective *sprints .* All requirements with a priority of four are implemented in the first *sprint .* All requirements with a priority of 3 in the second *sprint,* and so on.
This initially results in four *sprints,* of which only three are part of this work. The requirements also do not contain any technical, spatial or other specifications. They only represent the general requirements or tasks of the motion control.

Priority class 4 results in requirements regarding interfaces and communication. There are also requirements regarding the type of movement. Priority class 3 contains requirements for the targeted control and regulation of the driving movement along a point-to-point connection. The requirements within priority class 2 relate to the spatial consideration of the driving movement or the movement along a path. The last priority class describes all requirements for optimized motion control, for example driving along curves or in dynamic environments.

Table 4-1: *Product Backlog* Part 1

Product Backlog		
Nr.	**Anforderung**	**Priorität**
1	Datenübertragung und Kommunikation zwischen Trackingsystem und FTF	4
2	Implementieren der Hardware	4
3	Kalibrieren des Trackingsystems	4
4	Ansteuern von FTF über eine kabellose Schnittstelle	4
5	Überlagerung von Bewegungen	3
6	Fahren von geraden Strecken	4
7	Rotieren um die eigene Achse des FTF	4
8	Fahren von Kurven	1
9	Direktes Fahren zu einem vorgegebenen Punkten	3
10	Lageregelung des FTF	3
11	Stetiger Geschwindigkeitsverlauf	3
12	Interpolation einer Trajektorie	3
13	Synchrone Bahnplanung	3
14	Fahren entlang eines vorgegebenen Pfades über mehrere Punkte	2
15	Autonomes Suchen eines Pfades zu einem vorgegebenen Punkt	2
16	Fahren in einer statischen Umgebung	2

Table 4-2: *Product Backlog* Part 2

Product Backlog		
Nr.	**Anforderung**	**Priorität**
17	Kollisionsfreies Fahren in einer dynamischen Umgebung	1
18	Dynamische Hindernisse detektieren	1
19	Kompensation von Bewegungsungenauigkeiten	2
20	Erstellen von virtuellen Hindernissen	2
21	Kollisionsfreies Fahren im freien Konfigurationsraum	2
22	Definierbarer Arbeits- und Konfigurationsraum	2
23	User Interface	1
24	Visualisierung/Simulation des Fahrweges des FTS im Konfiguartionsraum	1
25	Verschleifen von Punkten entlang des Fahrweges	1
26	Stetiger und Ruckfreier Geschwindigkeitsverlauf	1

4	Höchste Priorität
3	Hohe Priorität
2	Mittlere Priorität
1	Niedirige Priorität

5 interfaces for data transmission

The communication and data transfer between the individual components of the vehicle has already been considered. This chapter now deals with a higher level: the communication between the vehicle itself and the tracking system. Only by linking these two individual systems can a driverless transport system be developed. For this purpose,
The requirements for the system are first described in the *SprintBacklog .* In order to ensure data transfer between the systems, it is then necessary to consider which interfaces are available for the respective systems and which options are available for connecting an interface. Finally, the appropriate hardware is implemented and the interfaces programmed.

5.1 Sprint Backlog #1

In the first *sprint ,* all requirements that have been given priority 4 in the *product backlog* (see Chapter 4) are implemented . *Sprint backlog #1* (see Table 5-1) summarizes the requirements again. The first requirement calls for data transmission or a communication path between the tracking system and the AGV. The AGV should be controlled via a wireless interface. For this, the hardware for the interface must be implemented and the tracking system installed and calibrated. It is also specified that the vehicle

should travel straight distances and rotation around the vehicle's own axis must be possible so that the vehicle can operate freely in any direction of movement.

Table 5-1: *Sprint Backlog #1*

Sprint Backlog #1	
Nr.	**Anforderung**
1	Datenübertragung und Kommunikation zwischen Trackingsystem und FTF
2	Implementieren der Hardware
3	Kalibrieren des Trackingsystems
4	Ansteuern von FTF über eine kabellose Schnittstelle
5	Fahren von geraden Strecken
6	Rotieren um eigene Achse des FTF

5.2 Interfaces of the existing systems

As already described, an AGV is to be set up. This consists of an AGV and a tracking system. In order to achieve an AGV, data transfer between the systems is necessary. In this chapter, the existing interfaces of the individual components are examined.

Figure 5-1 shows the interfaces of the individual systems. It is clear that no direct data transfer is possible between the tracking system and the vehicle. The tracking system detects the position of the AGV, but has no way of sending data to the AGV itself. Therefore, a detour via another interface must be taken. An interface can be generated using a script, which reads the vehicle coordinates from the tracking system and also passes the driving commands to the vehicle. The tracking system transmits

its data via a REST API. The vehicle's *Arduino board ,* on the other hand, offers several connection options to connect it to a Bluetooth module or other radio modules such as
ZigBee or APC220. For reasons of availability, space and minimal effort, a Bluetooth module is chosen for communication. The Bluetooth module is addressed via the script. The following sections therefore describe how REST API and Bluetooth work.

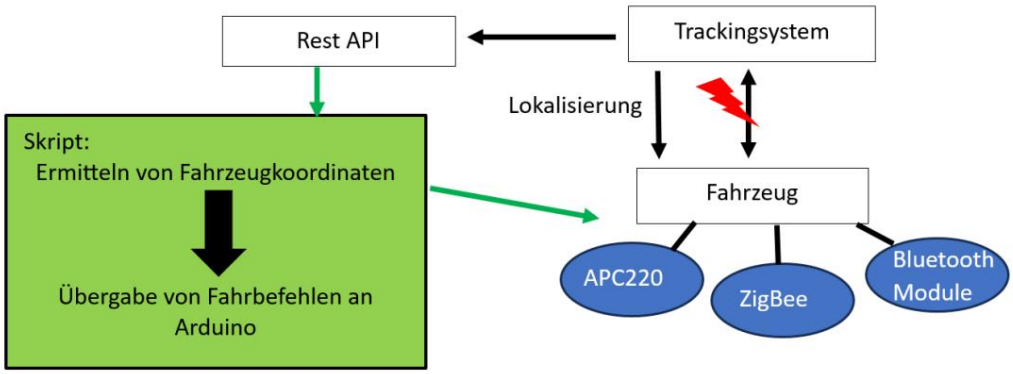


Figure 5-1: Interfaces between the systems

5.2.1 Rest API

An API *(Application Programming Interface)* is an interface for exchanging data, usually between a *client* and a *server.* REST is the abbreviation for *Representational State Transfer.* It describes a standardized way of exchanging this data. REST APIs are stateless . This means that *the client* and *server* must communicate with each other in a stateless manner. Every request from a *client* therefore contains all the information that a server needs.
The *server* itself cannot access any stored context. REST is used in particular for *web services* and is based on the *World Wide Web (WWW)* web technology . The *WWW* is an interactive information system for the worldwide exchange of data. It consists of hypertext systems, so-called websites. A website usually comprises several related web documents. These are located on special computers that are connected to the Internet. The computers provide them using web server services. Browsers are required to access the documents. A web document is addressed using the URI *(Uniform Resource Identifier).* A platform-independent language, *HTML (Hypertext Markup Language),* is used for hypertext documents on the *WWW.*

used. To access documents on the web via hyperlinks, the Hypertext Transfer Protocol (http) was developed for the transfer of data. Figure 5-2 shows this principle in a simplified form. A web browser displays a website using HTML. If a link on this website is clicked, a web document is opened on the

Web server is requested. The content is retrieved using HTTP. This content is then displayed on the browser using HTML. (cf. Lackes, 2018; FH Bielefeld, 2017)

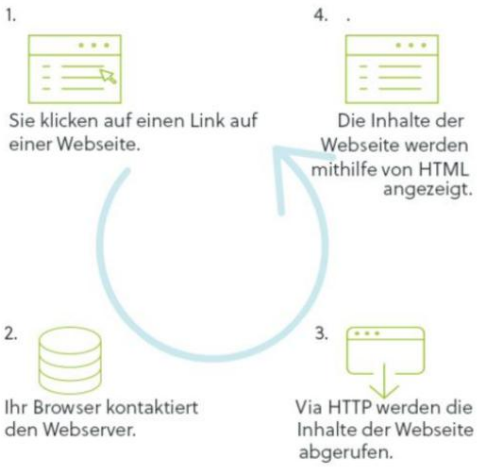


Figure 5-2: How to retrieve web documents on the WWW
(fonial GmbH, 2023)

The REST API works in a similar way. The *client* requests information from a resource via the REST API (see Figure 5-3). The REST API processes this request and returns all relevant information about the resource to the *client* . The information is translated into a format that the *client* can easily interpret. In addition, *clients* can use a REST API to change elements on a server and also add new elements. The requests are made via HTTP commands. These are similar to the typical database commands *create, read, update* and *delete.* There are also four HTTP commands: (see Parmar, 2022)

- POST to create a resource using *create*
- GET to retrieve or read a resource using *read*
- PUT to edit or update a resource using *update*
- DELETE to delete a resource using *delete*

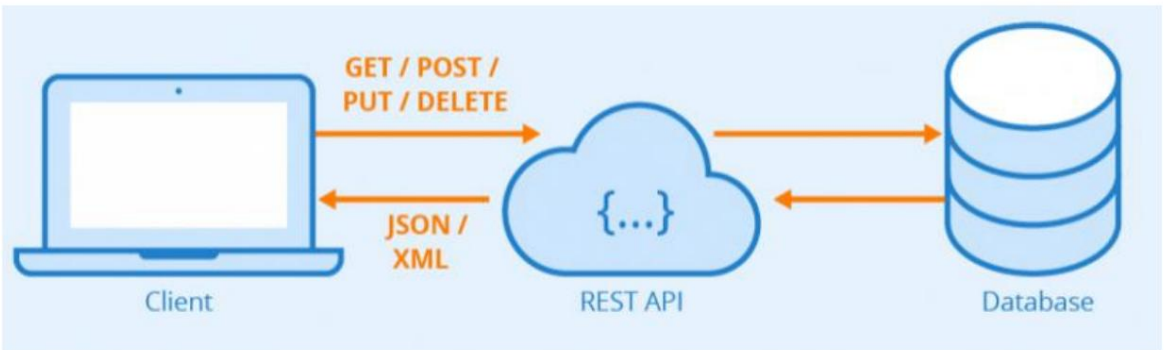


Figure 5-3: REST API
(Seobility, 2023)

In order to send a request to a resource, the resource must be known or identified. This is done in a similar way to the WWW via a URI. The URI defines the access location of an individual resource in a network. The identifier must be based on a standardized

schema that specifies five parts. (cf. FH Bielefeld, 2017)

- **Schema:** The schema is the protocol used to request the respective resource. HTTP is usually used for REST.

- **Authority:** The authority refers to the organization to which the offered resource is subject
- **Path:** The path describes where the resource is located within the organization and is typically hierarchical. The individual hierarchy levels are separated from each other by a *backslash* .

- **Query:** A query can optionally be extended to include a query. It is used when the location of the resource cannot be specified precisely using the path alone. They are also used to retrieve specific information from a source identified by the path, such as a record from a database.

- **Fragment:** A fragment is triggered exclusively by the *client* and is not transmitted to the *server* via a request . The use of fragments can also vary depending on the protocol used. For example, if a fragment is part of an HTTP URI that returns an HTML object, the browser automatically scrolls to the HTML tag described by the fragment after the entire web page has been displayed.

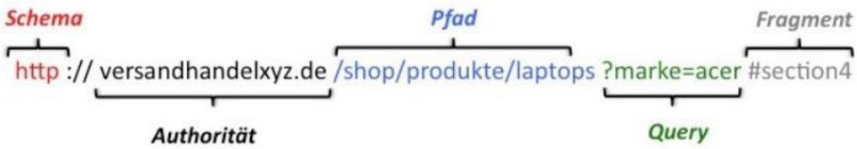


Figure 5-4: Components of a URI
(FH Bielefeld, 2017)

5.2.2 Data exchange via Bluetooth

Bluetooth is a radio technology that allows devices to communicate with each other wirelessly and without a direct line of sight. It is therefore used for a wide range of applications.
One of the most common uses for Bluetooth is a wireless connection between a smartphone and audio devices such as headphones or hands-free devices. Since a large number of different manufacturers now offer Bluetooth devices, perfect interoperability is a fundamental requirement for smooth communication. This is ensured using standards. Communication between the devices takes place via piconets. So that several piconets can be operated simultaneously in one place, each piconet uses its own *hopping sequence.* Bluetooth has 79 channels available for this. This number is sufficient to operate many Bluetooth networks simultaneously in one place without significant mutual interference. The *hopping sequence* of the piconet is calculated using the hardware address of the end device that first makes contact with another end device. This end device also sets up the piconet. There is always a master in a piconet.

The master can be connected to up to seven slave devices. Since Bluetooth is often only used for point-to-point connections, this number of devices within a network is completely sufficient. Each device can be *the master* or *slave* of a device. The *master* is the device that originally sets up the piconet. The *master* is always in control and decides who can transmit data and at what time. (see Sauter, 2022: pp. 339-346)

5.3 Implementation of hardware and software interfaces

The implementation of the hardware and the software interfaces are fundamental for setting up a motion control system for an AGV. The hardware components to be implemented include the tracking system and the integration of a Bluetooth module. An *HC05 Bluetooth wireless RF transceiver module RS232* is used as the Bluetooth module.

5.3.1 Installation and calibration of the tracking camera

This chapter deals with the installation and calibration of the tracking system. Since at the beginning only the tracking camera and a calibration template of size A0 are available, the tracking camera must first be set up and the necessary components for calibration must be made available. The tracking camera is screwed to a wooden crossbeam. This is supported by two tripods. The height of the tracking camera can be adjusted using the tripods. It is important to ensure that the tripods are the same height (see Figure 5-5).

Figure 5-5: Structure of the tracking system

As soon as the tracking camera is set up, it is wired. The wiring is done according to the installation instructions (see Figure 5-6). A *PoE switch (Power over Ethernet)* is required for the installation. A *PoE switch* is a network switch that supplies network-capable devices with electrical power via an Ethernet cable. At the same time, network signals can also be transmitted (see Intellinet, 2023). The tracking camera is connected to the *PoE switch* via an Ethernet cable.

connected. The *PoE switch* is supplied with power via a power supply unit. The *PoE switch* is also connected to the laptop or computer. The computer or laptop is in turn connected to the local network via a LAN cable.

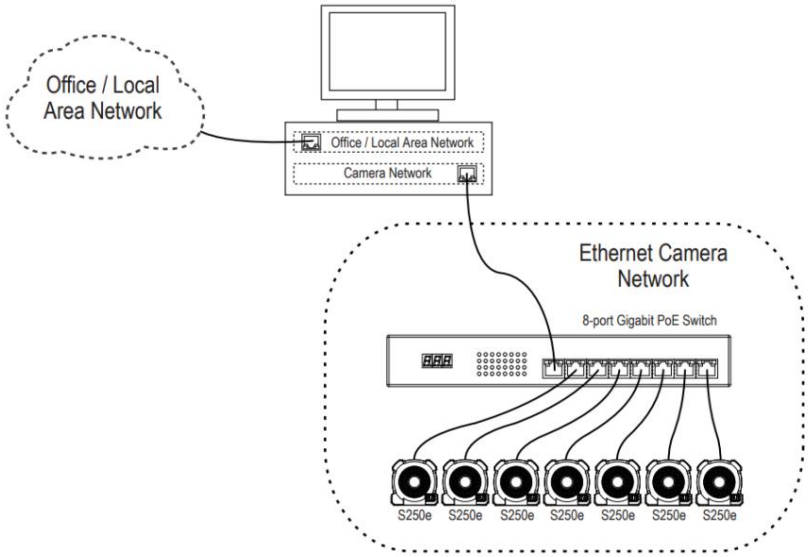


Figure 5-6: Installation of the tracking camera
(Natural Point, 2010)

After the tracking camera has been successfully set up, the camera now needs to be calibrated. The university provides a 2D tracker application for calibrating and operating the tracking camera. This has an HTML page that documents and enables use (see Figure 5-7). The tracker's REST interface can be accessed via the URI *http:// laptop-n1q2j4ee:1201* . The tracked objects can be queried via the URI *http:// laptop-n1q2j4ee:1201/ data?homography=-2* . If the value -2 in the *query* is changed to -1, the calibration can take place. After the one-time initial calibration, the value of the *query* can be reset to -2. This then automatically retrieves the calibrated configuration.

2dTracker

2dTracker is using Naturalpoint Optitrack cameras to recognize reflective markers and provide their 2d-coordinates as a REST-server.

All provided positions and lengths are in mm; orientations in rad

/index.html	This list of GET-query commands.
/cameras	List of found OptiTrack cameras.
/image.png	Current camera image with tracking data.
/data?homography=-2	Sets the homography calculation interval and shows current homography-matrix.
/data?markers	Lists all recognized marker objects.
/data?bodies	Lists all recognized body pairs of marker with its distances.
/data?body=*dist*	Replies with the body whose marker-distance is closest to the requested *dist*.
/data?body=*dist*$*tol*	Replies with the body whose marker-distance is closer than *tol* to the requested *dist*.

Figure 5-7: HTML page of the tracker

To perform the calibration, the URI *query* must be set to the value -1. In addition, the calibration template is placed under the tracking camera. There are four circles on the calibration template
marked. The first circle 0 has a diameter of 70mm. In relation to the coordinate system drawn in the template, this is located at the coordinates x=-525 and y=-350. The remaining circles have a diameter of 40mm. They are located at the points:

a. X=525/Y=-350
b. X=525/Y=350
c. X=-525/Y=350

Reflective markers of the appropriate size must be positioned at these coordinates.
The reflective markers are recorded by the tracking camera and used for calibration. However, since the vehicle to be tracked has a certain height, it is necessary that the markers are read at the same height. Otherwise, parallax errors will occur. In order for the markers to assume the same height, so-called calibration towers are additively printed using a 3D printer (see Figure 1).
Figure 5-8). The calibration towers have a height of 74mm and therefore correspond to the height of the vehicle. The calibration towers are then placed on the positions of the calibration template.
The calibration process can begin (see Figure 5-9). To do this, the application program for the tracking camera is started. As soon as the tracking camera detects the markers, a coordinate cross is placed in the calculated center. The URI query is set back to -2 and the process is complete. The calibration towers can be put aside.

Figure 5-8: Calibration template (left) and calibration tower (right)

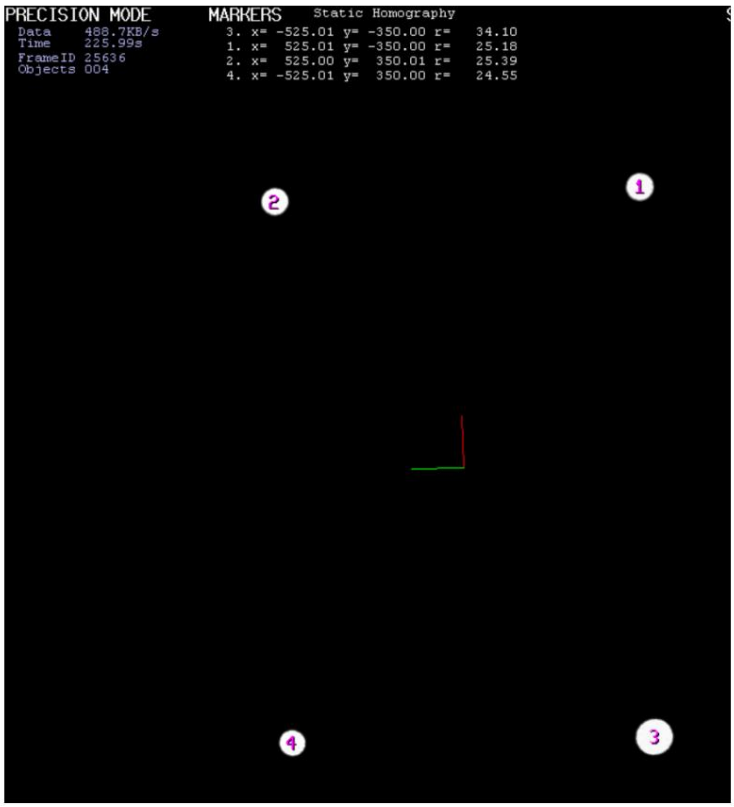


Figure 5-9: Coordinate system of the tracker after calibration

The next step is to test the tracking of the vehicle. To do this, the vehicle is equipped with two reflective markers of different sizes, whose positions in the plane can be recorded by the optical tracker. The system recognizes the vehicle or a component based on the distance between the markers and defines a coordinate system for the component.
This coordinate system has its origin in the position of the larger marker and the x-axis points in the direction of the smaller marker. This means that not only a vehicle position but also a vehicle orientation can be output. It is important that the application program always records four markers at the same time from the camera, otherwise the application program may crash and no positions will be output. This means that even if only two markers are used to track the vehicle, it is necessary that two more markers are in the room. Figure 5-10 shows an example of the HTML page that

describes the existing markers. The markers are divided into *bodies* over the distances summarized.

Bodies

1: dist=53.5
marker1 : {nr : 4, x : -348.8, y : 135.6, r : 14.6}
marker2 : {nr : 3, x : -307.7, y : 169.9, r : 9.8}

2: dist=111.9
marker1 : {nr : 2, x : 545.1, y : -353.4, r : 25.0}
marker2 : {nr : 1, x : 574.2, y : -245.3, r : 24.9}

3: dist=974.8
marker1 : {nr : 1, x : 574.2, y : -245.3, r : 24.9}
marker2 : {nr : 3, x : -307.7, y : 169.9, r : 9.8}

Figure 5-10: Tracking system "Bodies"

5.3.2 Implementing the Bluetooth module HC05

This chapter looks at the implementation of a Bluetooth module. In this work, the *HC05 Bluetooth Wireless RF Transceiver Module RS232* will be used (see Figure 5-11). This is a powerful Bluetooth device that can be set up as either *a master* or *a slave .* The integrated Bluetooth transceiver device can convert serial interfaces into a Bluetooth interface. Therefore, it is often used to establish communication between an *Arduino board* and another device such as a computer. The prerequisite for this is that the computer or other device is equipped with a Bluetooth connection. (see Figure 5-12).

AZ-Delivery, 2023)

Figure 5-11: *HC05 Bluetooth Wireless RF Transceiver Module RS232*
(AZ-Delivery, 2023)

Before the HC05 module can be connected to the vehicle's *Arduino board ,* it is necessary to configure the module. This is easier with the help of a second *board* such as an *Arduino Uno* or an *Arduino Mega.* Since an *Arduino Mega board* is available, this will be used below. The HC05 module can be operated in two different modes. There is the AT command mode and the data mode. The AT command mode is used to configure the module, while the data mode is used to exchange data.

The HC05 module has five pins, a push button and an LED. The pins *State, VCC, Ground, RX, TX* and *Enable/Key* are distinguished: (see Components101, 2021)

- *Enable/Key:* This pin is used to switch between data mode and AT command mode used. By default, the HC05 module is in data mode.
- *VCC:* Supplies the module with power and must be connected to a 5V supply voltage
- *Ground:* This is the ground pin of the module. This must be connected to the *"Ground"* of the *Arduino boards* can be connected.
- TX Transmitter: Transmits serial data. Anything received via Bluetooth is output as serial data via this pin.
- RX Receiver: Receives serial data. All serial data sent to this pin are transmitted via Bluetooth.
- *State:* The state pin is connected to the integrated LED and can be used as feedback to check whether Bluetooth is working properly.
- LED: The LED indicates the status of the module:
  - o Flashing once every 2s means the module is in AT command mode passed over.
  - o Repeated flashing means the module is waiting for a connection in data mode
  - o A blink twice per second indicates that the connection is in data mode was successful
- Push button: Used to control the *Key/Enable pin* to switch between data and command mode is used.

Figure 5-12 shows how the HC05 module is connected to the *Arduino Mega board .* The *ground pin* is connected to the *ground pin* via a cable (orange), RX goes to RX (blue), TX is wired to TX (red), the VCC pin receives its supply voltage from the 5V pin of the *board* (green) and the *enable/ key pin* is coupled to the 3.3V pin (yellow).

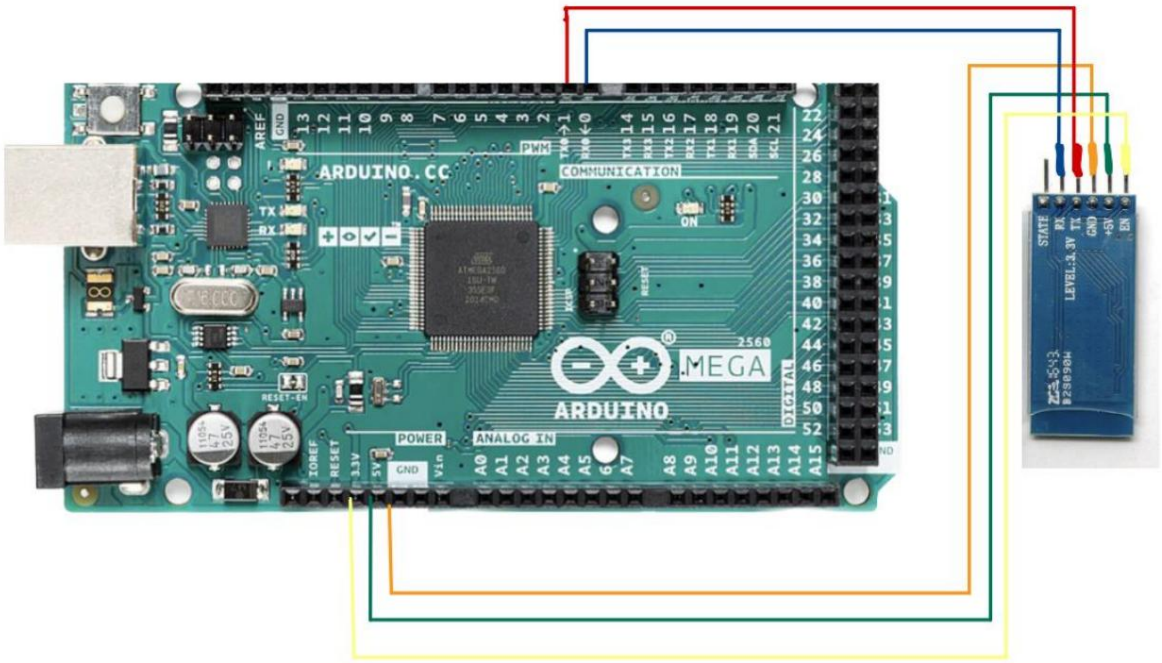


Figure 5-12: Connecting the HC05 module to the *Arduino board* "Arduino Mega"

In the next step, the *Arduino IDE* is opened on the computer and an empty script is loaded. The *Arduino board* is connected to the computer using a USB cable and the empty script is loaded onto the *board.* loaded. The connection between *VCC* and 5V is then separated and reconnected, but the push button must be pressed continuously during the connection. This puts the module into AT command mode. The LED flashes every two seconds, confirming the mode. Since the module is now in the correct mode, it can be set. To do this, open the serial monitor of the *Arduino IDE* . The commands for setting up the module are entered here. Figure 5-13 shows the command chain that is used at this point to query information from the module and to set the baud rate. The baud rate indicates how many symbols are transmitted per second. One baud corresponds to one symbol per second (cf. Kunbus, 2023).

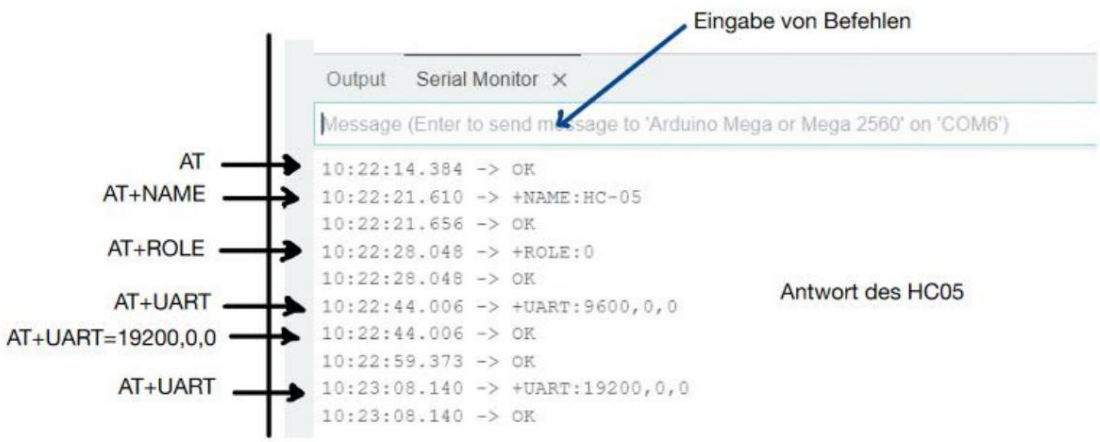


Figure 5-13: Command sequence for setting the HC05 module

First, the module is queried using the "AT" command, which confirms the connection with the return "OK". Then, information about the module is queried using "AT+NAME", "AT+ROLE" and "AT+UART". The "AT+ROLE" command is used to determine whether the module is set as a master or a slave. By returning the value 0, this indicates that it is currently a slave. This means that it is in the correct state for the application of this work. The "AT+UART" command is used to query the baud rate. This is 9600 baud by default. To enable faster data transfer, the baud rate should be increased to 19200 baud. This is done using the command "AT+UART=19200,0,0". The baud rate is confirmed by querying it again.

After the module has been successfully configured, it is connected to the vehicle's *Arduino board* . The connection is similar to the previous one, but a few changes must be made. Unlike the connection to the *Arduino Mega* , the RX and TX pins are swapped and the connection between the *Enable/ Key pin* and the 3.3V pin is omitted (see Figure 5-14). The module's LED flashes and waits for a connection to be established.

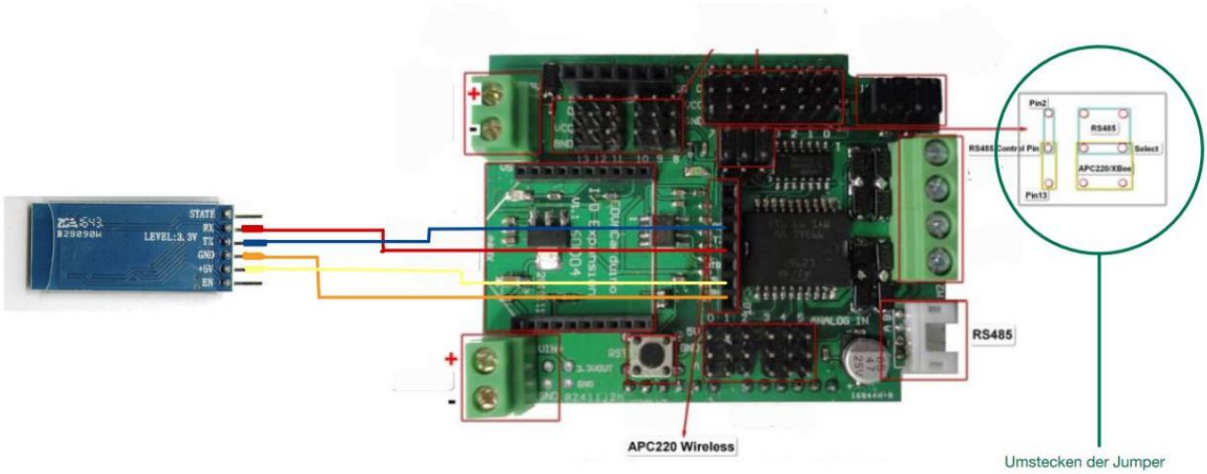


Figure 5-14: Connecting the HC05 module to the vehicle's *Arduino board*

To establish a connection, the HC05 module is added to the computer's Bluetooth settings.
added and using a password (default 1234) the devices can be paired with each other. However, pairing the devices does not automatically mean that they are connected to each other for data exchange. For data exchange the COM port of the

module. This is also done via the Bluetooth settings of the computer. In this case, the Bluetooth module is addressed via COM5. With the help of a Python script that tests the connection, an exchange of data should now be possible. However, the connection fails on every attempt. After several tests, it is determined that this is related to the jumper assignment on the Arduino expansion board (see Figure 5-14).

marked green). The *pinout* of the board indicates that, depending on the jumper assignment, either the RS485 interface or the APC220 interface can be used. Since the Bluetooth module is connected to the APC220

interface, the jumpers must be changed accordingly. However, this has the disadvantage that the RS485 interface can no longer be used. However, the ultrasonic sensors communicate via this. Since the ultrasonic sensors are not initially used for this work, changing the jumpers is legitimate. If the RS485 and the APC220

However, if the interface is to be operated simultaneously, another solution must be found.
Once the jumpers are in the correct position, a connection can be made.
The module's LED confirms this by flashing twice within one second. The implementation of the module has been completed successfully.

5.3.3 Program code for data exchange

After successfully implementing the hardware components, the software interfaces for data exchange should now be considered. First, a script should be written that outputs the position and orientation of the vehicle in the tracking system.

The REST API provides the data as JSON. When the coordinates are requested, a status value and a JSON structure with the requested data are returned. JSON or *JavaScript Object Notation* is a text-based file format. JSON is completely independent of programming languages, so this format is particularly suitable for data exchange. In JSON, the data is expressed in name/value or key/value pairs (see 4D SAS, 2023).

In order for Python to process the JSON data, the JSON library must be imported (see Program code 5-1). The libraries *Requests* and *Time* are also required. The *Requests* library allows you to easily send HTTP requests. The *Time* library offers time-related functions that can be used after importing the library.

The command *r=requests.get(URI)* reads the data from a resource with the corresponding URI. The URI contains the address of the tracker. The *query* "body=56" at the end of the URI selects the component that has a marker distance of 56mm. This allows the vehicle to be clearly identified and the vehicle coordinates and the status 200 are returned. Finally, the positions x and y and the orientation *phi* are output to the console using the *print()* command , so that the user receives graphical feedback of the information.
This HTTP request is wrapped in a *while* True loop with a time delay function *time.sleep(2)* integrated. Therefore, the data is requested every two seconds. This means that changes in the position and orientation of the vehicle can be directly detected. Ultimately, it can be determined that the data is successfully read from the REST API via the Python script.

```
1   import json
2   import requests
3   import time
4
5   if __name__ == '__main__':
6       while True:
7           r = requests.get("http://laptop-n1q2j4ee:1201/data?body=56")
8           r=json.loads(r.text)
9           print("x="+str(r["x"]))
10          print("y="+str(r["y"]))
11          print("phi="+str(r["phi"]))
12          print("")
13
14          time.sleep(2)
```

Program code 5-1: Reading the tracking data

The next step is to establish a connection between the computer and the vehicle's HC05 module. This requires a Python code on the one hand and a corresponding *Arduino sketch* on the other , which can be loaded onto the vehicle.
First, the Python code is considered.

To establish serial communication, the library *Serial* or *pyserial* needed. This is therefore imported at the beginning. A connection is then established using the function *arduinoData=serial.Serial('COM5', 19200) ,* where *arduinoData* represents any variable. The COM port COM5 specifies the output of the HC05 module. The number 19200 is the baud rate used. The specified baud rate must correspond to that of the HC05 module. In the *Arduino*
This baud rate must also be specified *in the sketch .* Now a while loop is started in which the

User can make an input. The entered command is then extended by the string '\n'. This indicates the end of the command during data transfer.
Finally, the command *arduino.write(cmd.encode())* is used to transfer the command or input to the serial interface. Before the transfer takes place, the function *cmd.encode()*
the command is converted into another encoding form. UTF8 is used by default. UTF8 is the most widely used Unicode character. Each character is converted into a binary code of zeros and ones. This is necessary because the serial interface can only transmit bits and not strings. After an input has been made, the user can start the next input directly.

```
1  import serial
2  arduinoData=serial.Serial('COM5',19200)
3
4  while True:
5      cmd=input('Please Enter Your Commamd: ')
6      cmd=cmd+'\n'
7      arduinoData.write(cmd.encode())
```

Program code 5-2: Sending data via serial interface from Python

An *Arduino sketch* is set up to receive the command sent . The program code of the demo script is used for this (see chapter 3.2.5 Demo script). However, all functions related to the ultrasonic sensors are removed, as these are not currently relevant. Since SprintBacklog #1 specifies that the vehicle should drive in a straight line and rotate around the vehicle axis, the functions for the directions of travel are taken from the demo script. The *void motion[]* array is, however, adjusted. It now contains six movement functions and a stop function (see program code 5-3). A new *driveCar* function with a *direction* variable is also introduced. This function is briefly described below.

```
85  void(*motion[7])(unsigned int speedMMPS) = {goAhead,turnLeft,turnRight,backOff,rotateLeft,rotateRight,allStop};
86
87  void driveCar(unsigned speedMMPS){
88    unsigned int direction;
89    while(Serial.available()==0){
90    }
91
92    direction=Serial.readStringUntil('\n').toInt();
93    (*motion[direction])(speedMMPS);
94
95    Omni.PIDRegulate();
96
97  }
98
99  void setup() {
100 Serial.begin(19200);
101
102 delay(2000);
103 TCCR1B=TCCR1B&0xf8|0x01;    // Pin9,Pin10 PWM 31250Hz
104 TCCR2B=TCCR2B&0xf8|0x01;    // Pin3,Pin11 PWM 31250Hz
105
106 Omni.PIDEnable(0.35,0.02,0,10);
107 }
108
109 void loop() {
110
111 driveCar(100);
112 }
```

Program code 5-3: Arduino sketch for accepting and processing commands

Using a while loop, no action is taken until data arrives via the serial interface. As soon as data arrives, it is read using the *Serial.readStringUntil('\n').toInt()* function . This means that all bits that are sent up to the end marker '\n' are read. These are then transformed back into a string. This string should be assigned to the *direction* variable. However, since this is defined as *an integer value ,* the string must be converted again into an *integer.* This is done by the .toInt function . The *direction* variable is then used to select the appropriate driving command from the *motion[]* array. If the user enters the value 0 in the Visual Studio Code3 console , for example, the *goAhead* driving command is executed. If the value is 5, the *rotateRight* function is activated and the vehicle should turn clockwise. Finally, the *Omni.PIDRegulate()* function should be used to regulate the driving behavior.

Now we look at the *void setup() .* This also contains similar code to the demo script. Only the line *Serial.begin(19200)* is added. This activates the serial interface. The data exchange takes place at 19200 baud and thus corresponds to the same value as in the Python script. In the *void loop(),* only the function *driveCar(100)* is called, where the value 100 indicates the driving speed in mm/s.

After the interface has been tested using various user inputs, it can be determined that a stable interface between the computer and the vehicle has been implemented. In addition, the tracker data is queried using another script, so that an interface between the computer and the tracking system is also available. However, the two scripts are not yet connected to each other. This means that although it is possible to control the vehicle, this is currently completely independent of the tracking system. This dependency will be created in the next chapter or in the next *sprint.4*

3 Visual Studio Code is used as the environment to write the Python script
4 The complete Arduino sketch for Sprint 1 can be found in Appendix C.

6 Motion control along a point-to-point connection

This chapter deals with motion control in terms of position control and path or trajectory planning for a point-to-point connection. The requirements for the sprint are briefly summarized in a *sprint backlog* . The basics of cascade and position control, trajectory planning and the transformation of coordinate systems are then described. These sections also contain derivations of formulas. These are required for the individual program codes in Python. In the last part, these program codes are described and their relationships explained.

6.1 Sprint Backlog #2

Sprint Backlog #2 summarizes the requirements that are to be implemented in the second sprint. The first requirement states that an overlap of movements should be possible.
Until now, movements were defined using statuses. This means that if the status was set to "drive ahead", a forward movement was carried out. Now, however, it should be possible to overlay the directions of movement and the rotation. The vehicle should therefore be able to move at an angle in any direction *and* at the same time rotate on its own axis. The second requirement calls for direct travel to a given point. If there is a starting point and a given destination point, the vehicle should travel directly to the destination. There should be a constant speed curve. For this, a trajectory must be interpolated. Movement along this trajectory or path should be synchronous. This means that both the target position and the target orientation should be reached at the same time.

Table 6-1: Sprint Backlog #2

Sprint Backlog #2	
Nr.	**Anforderung**
1	Überlagerung von Bewegungen
2	Direktes Fahren zu einem vorgegebenen Punkten
3	Lageregelung des FTF
4	Stetiger Geschwindigkeitsverlauf
5	Interpolation einer Trajektorie
6	Synchrone Bahnplanung

6.2 Basics of cascade control, trajectory planning and transformation of coordinate systems

In order to implement attitude control and trajectory planning, knowledge of these topics is required. Therefore, cascade control, trajectory planning and coordinate transformation are described and the required formulas are derived.

6.2.1 Cascade control

Position control in robots or vehicles with electric drives is usually implemented as a cascade control. Each motor is controlled individually. The control takes place on different levels. This is why it is also referred to as a subordinate control loop. The current controller, which is usually a PI controller type, operates on the lowest level. This current control loop has the highest dynamics. The speed controller is superordinate to it. This provides the setpoint of the motor current isoll. This is usually implemented as a PI controller or a PID controller.

The position controller is on the top level. This outputs the speed setpoint vsoll . In the simplest case, a proportional controller is used for the control. The position setpoint is referred to as ssoll . Figure 6-1 shows such a cascade control. The current controller is usually included in the power electronics of the motor. The cascade control has the advantage that disturbances inside the cascade are regulated with a high bandwidth. (see Winkler, 2016: p.30f)

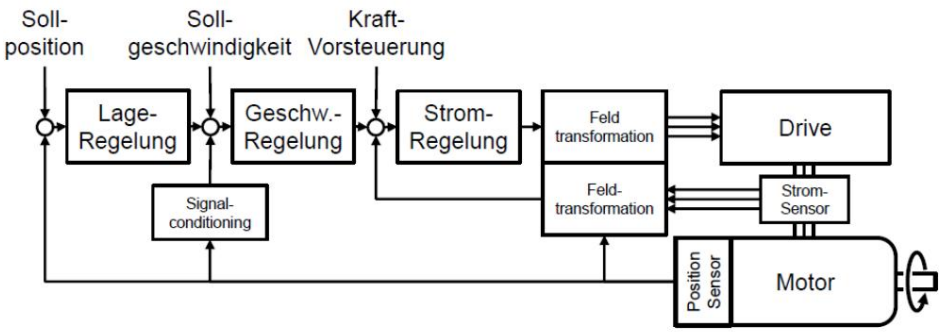


Figure 6-1: Cascade control (Hofschulte, 2022: p.230)

The existing system also requires the implementation of a cascade control, with the current control already integrated in the power electronics of the motors. This means that a speed control and a position control must be introduced. The speed control is implemented by the *Arduino ,* while the position control is implemented using a Python script. However, the design of the speed controller is not necessary, as this is implemented using the integrated Arduino libraries. Therefore, only the position control is designed.

From the control loop it is clear that the position deviation is converted into a speed during position control. Three parameters are considered for the target position. Firstly, the position coordinates x and y, and secondly, the orientation via the angle . These target values should be achieved. Therefore, the control deviation is determined from the target position and the current position. The current position is designated with and because this is where the

,

vehicle is located.

ÿ = (6-1)

ÿ = (6-2)

ÿ = (6-3)

The control deviation is converted into a speed via the position control. The easiest way to do this is with a P controller. This means that a control parameter is introduced for the conversion. Two speeds are determined during the control. The first is the translational speed. This results from the speed components in the x-direction and y-direction.

$$= \ddot{y} (\ddot{y}) \quad (6\text{-}4)$$

$$= \ddot{y} (\ddot{y}) \quad (6\text{-}5)$$

$$= \ddot{y} \overline{2 \quad + \quad 2} \quad (6\text{-}6)$$

$$= \ddot{y} \ddot{y}(\ddot{y}) 2 + (\ddot{y}) 2 \quad (6\text{-}7)$$

The second speed is the angular speed. This can be set independently of the translational speed. Therefore, another proportionality factor is introduced. The angular speed results from the angle difference and the proportionality factor.

$$= \ddot{y} (\ddot{y}) \quad (6\text{-}8)$$

In the equations introduced, the target position and the current position are known, so only the proportionality factors need to be determined. The two values should be determined experimentally. Two conditions must be observed. Firstly, the controller should not react too slowly to control deviations. Secondly, the vehicle should not oscillate and, if possible, overshoot. Consequently, the values must not be too small or too large. They must be set accordingly.

6.2.2 Path and trajectory planning of a point-to-point movement

The task of a driverless transport vehicle is often to deliver a product or something similar to a specific location. The vehicle is therefore supposed to carry out a movement from its starting point to a destination point. The set of all points that are passed through between a given starting point and a given end point are referred to in the path planning as a path or trajectory.
A path does not provide any information about speed and acceleration. If each point on the path is assigned speed and acceleration information at a specific point in time, this is a trajectory. The trajectory represents the position and orientation of a body as a function of time. An important part of trajectory planning is setting up time functions or speed functions for the path.

The time course can be linear or a higher-order polynomial function. When planning the path, so-called via points can be defined. Further positions are interpolated between these via points. Figure 6-2 illustrates this. The points shown in red are the interpolated points. The trajectory at a specific point in time is shown in orange. Both the trajectory and the interpolated points are not drawn for the entire path, but only as examples at some positions. Trajectory planning therefore provides a parameter representation with time or another parameter for a sequence of points to be traveled one after the other. (see Mareczek, 2020, p.17f])

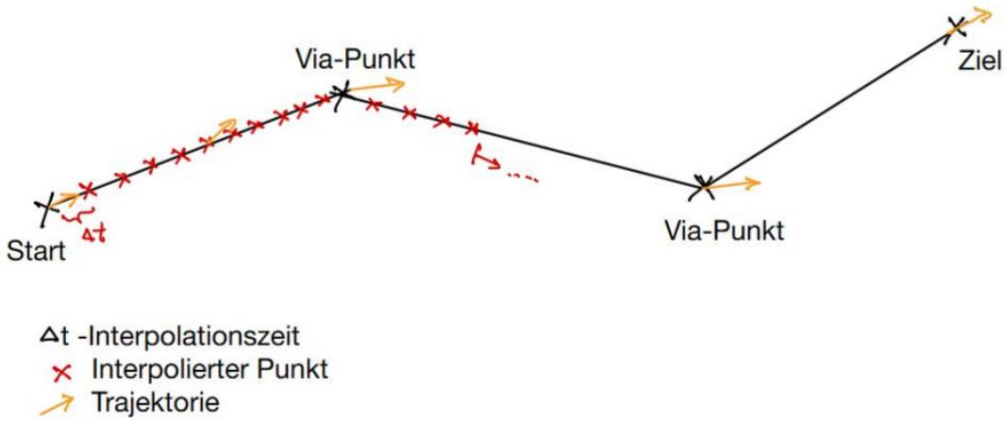


Figure 6-2: Trajectory according to Hofschulte, 2022: p.191

There are various methods for carrying out trajectory planning. At this point, a
Trajectory with trapezoidal velocity profile can be considered. In this method, no via points are defined.
The path should therefore run directly from the start to the end point. The simplest form is the interpolation
of the point-to-point movement with a ramp profile. This
trapezoidal velocity curve consists of three phases: (cf. Mareczek, 2020: p.24f)

Phase 1: Uniform acceleration
Phase 2: Driving at constant speed
Phase 3: Uniform deceleration

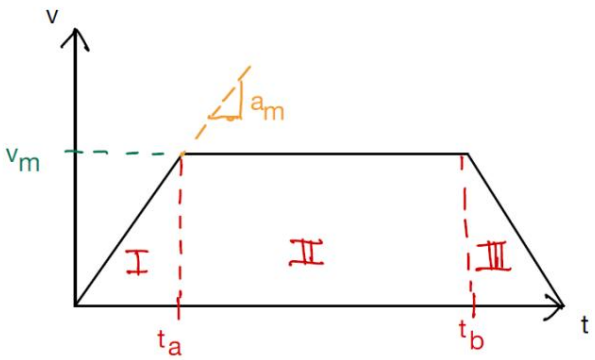


Figure 6-3: Speed curve with ramp profile according to Mareczek, 2020: p. 27

The advantage of this method is the simple calculation of the trajectory parameters. The disadvantage is the sudden acceleration, so that the movement is not free of jerks. The following quantities are introduced to describe this speed curve:

- Start and end time t0 and te
- Acceleration time ta
- Time at which the braking process begins tb
- Amount of acceleration and deceleration at
- Maximum value of speed vm
- Time point t
- Length of the total route

Using these parameters, the speed curve of a trajectory and the position of a body in relation to time can be described (see Figure 6-3). However, boundary conditions must first be defined in order to describe the curve. The following boundary conditions are initially

to comply with:

- Start and end states are known
- Speed at the beginning is zero
- Speed at the end is zero
- Speeds and accelerations are limited

In order to implement point-to-point control for a vehicle or to implement the points of the path, the distance to be traveled se is calculated from the known or specified parameters of speed, acceleration, start point and end point. The information for the speed and acceleration is then modified and the total time te, ta and tb is calculated. Interpolation, i.e. the calculation of the intermediate and target values, takes place on this basis.

To calculate the total distance, the distances of the individual phases are determined separately. In the first phase, the following equations result for acceleration, speed and distance:

$$0 \ddot{y} \ddot{y}$$

$$\ddot{y}(\ ) = \qquad (6\text{-}9)$$

$$\ddot{y}(\ ) = \ddot{y} + \ddot{y}(0) \qquad \ddot{y}(0) = 0$$

results for the speed

$$\ddot{y}(\ ) = \ddot{y} \qquad (6\text{-}10)$$

$$(\ ) = \frac{1}{2}\ddot{y} \quad \ddot{y}^{2} + (0) \qquad (0) = 0$$

results for the way

$$(\ ) = \frac{1}{2}\ddot{y} \quad \ddot{y}^{2} \qquad (6\text{-}11)$$

The second phase is the steady-state drive. The steady-state drive begins after the acceleration time and ends as soon as the braking time is reached. For this phase, the acceleration is zero. In addition, the speed is known.

$$\ddot{y} \ddot{y}$$

$$\ddot{y}(\ ) = \qquad (6\text{-}12)$$

$$\ddot{y}(\ ) = \qquad (6\text{-}13)$$

This leaves the description of the path. The distance covered is made up of the portion of the distance travelled during acceleration and the distance travelled with uniform movement.

$$(\ ) = \ddot{y}\ (\ddot{y}\ ) + (\ )$$

The acceleration time can be determined from equation 6-10 of the first phase:

$$\ddot{y}(\ ) = = \ddot{y}$$

$$= \quad \text{—} \tag{6-14}$$

This gives us the following route:

$$(\ ) = \ddot{y}\ (\ \ddot{y}\ ) + \quad \text{—} \quad \frac{1}{2}\ \ddot{y} \quad \ddot{y} \quad 2$$

$$= \ddot{y}\ \ddot{y} \quad \frac{1}{2}\ \ddot{y}\ \text{—} \quad ^{2} \tag{6-15}$$

Finally, the braking process remains in phase 3. The braking process starts at time tb and lasts the same time as the acceleration process. Therefore, the time at which the braking process starts can be calculated as the difference between the total time te and the acceleration time ta

During this phase there is a constant negative acceleration.

$$\ddot{y}\ \ddot{y} \quad = \ddot{y}$$

$$\ddot{y}(\ ) = \ddot{y}$$

The equation for the speed of the braking process is the speed at the beginning of the braking process minus the product of the acceleration and the time difference

between the point in time under consideration and the start time of the braking process.

$$\ddot{y}(\ ) = \ddot{y}\ \ddot{y}\ (\ \ddot{y}\ ) + \ddot{y}(\ ) = \ddot{y}\ \ddot{y}\ (\ \ddot{y}\ ) + \tag{6-16}$$

For the equation of the distance, the area under the ramp profile is considered. The total area under the curve is calculated from

$$= \ddot{y}\ (\ \ddot{y}\ ) \tag{6-17}$$

Only the last part from time t to the end is subtracted from this area. This gives the following equation for the distance:

$$(\ ) = \ddot{y}\ (\ \ddot{y}\ )\ddot{y} \quad \text{—}_{2} \ \ddot{y}\ (\ \ddot{y}\ ) \quad ^{2} \tag{6-18}$$

The entire distance can also be calculated using the formula for the area. The travel time for the route can be determined.

$$(\ ) = = \ddot{y}\ (\ \ddot{y}\ )$$

$$= + = + \quad \text{—} \quad \text{—} \tag{6-19}$$

In path planning, a distinction is also made between asynchronous and synchronous path planning. This means that if there are several axes or the movement is divided into several parts, it can happen that the individual axes do not reach the destination at the same time. In the case of a vehicle, this is, for example, the rotation of the vehicle and the translational movement. In an asynchronous movement, the orientation or the target angle is reached earlier than the target position. They therefore end independently of each other. In a synchronous movement, the orientation and the vehicle would reach the destination at the same time. They begin and end their movement at the same time.

Time. The travel time of the individual axes or direction and position must therefore match. The maximum value is assumed as the value for the entire time.

The speeds of the other axes etc. are adjusted accordingly. By changing the formula for the entire travel time 6-19, the new speeds of the individual components are obtained.

$$= \frac{}{2} \ddot{y}\ddot{y}, \frac{\sqrt{2\ddot{y}^2}}{4} \quad (6\text{-}20)$$

The interpolation of the individual points is ultimately carried out using the individual equations:

$$( ) = \begin{cases} \frac{1}{2}\ddot{y} \\ \ddot{y}(\ddot{y} \quad \frac{1}{2}\ddot{y} \quad) \\ \frac{1}{2}\ddot{y} \quad + \ddot{y}(\ddot{y})+ \ddot{y}(\ddot{y})\ddot{y} \\ \ddot{y}(\ddot{y}) \quad 2\ddot{y}\frac{1}{2} \end{cases} \quad \begin{matrix} 0\ddot{y}\ddot{y} \\ \ddot{y}\ddot{y} \\ \ddot{y}\ddot{y} \end{matrix} \quad (6\text{-}21)$$

$$( ) = \{ \quad \ddot{y}(\ddot{y} \quad) \quad \begin{matrix} 0\ddot{y}\ddot{y} \\ \ddot{y}\ddot{y} \\ \ddot{y}\ddot{y} \end{matrix} \quad (6\text{-}22)$$

6.2.3 Transformation between the coordinate systems

Coordinate systems are required to describe the position and movement of the vehicle. The system has two different coordinate systems, one a body-fixed coordinate system and the other a reference coordinate system. The body-fixed coordinate system is in the middle of the vehicle. The reference coordinate system is specified by the tracking system. This is determined when the tracking camera is calibrated. Figure 6-4 shows the two coordinate systems in the plane. It can be seen that the z-axis of the body-fixed coordinate system points orthogonally to the ground. The reference coordinate system, on the other hand, is the other way round in space. In order to carry out a vehicle movement between the actual position and the target position, the vehicle needs information about the direction of travel, direction of rotation and speed. These relate to the body-fixed coordinate system of the vehicle. The actual position and the target position of the vehicle are output by the tracking system in relation to the reference coordinate system. A coordinate transformation is therefore required to determine the information for the vehicle movement.

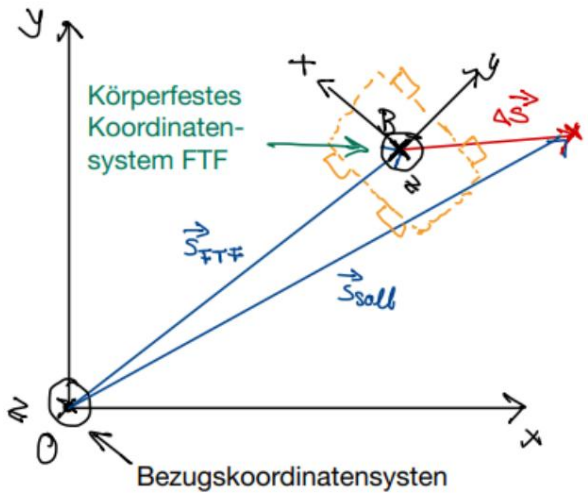


Figure 6-4: Coordinate systems

The coordinate systems are transformed using a displacement vector and elementary rotations. The aim is to bring the reference coordinate system into line with the body-fixed coordinate system of the AGV. To achieve this, two elementary rotations are required, first a rotation around the z-axis, then a rotation around the z-axis. The following formulas apply to the elementary rotations (cf. Hofschulte, 2022: p.69):

$$= \begin{bmatrix} \cos(\ ) & \ddot{y}\sin(\ ) & 0 \\ \sin(\ ) & \cos(\ ) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6-23}$$

$$= \begin{bmatrix} \cos(\ ) & 0 & \sin(\ ) \\ 0 & 1 & 0 \\ \ddot{y}\sin(\ ) & 0 & \cos(\ ) \end{bmatrix} \tag{6-24}$$

The value for the elementary rotation around the z-axis corresponds to the orientation of the vehicle. The elementary rotation around the y-axis is used to reverse the direction of the z-axis so that it also points orthogonally to the ground. To do this, the coordinate system is rotated 180° around the y-axis. Inserting the 180° into the matrix for the elementary rotation around the y-axis results in the following matrix:

$$= \begin{bmatrix} \ddot{y}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \ddot{y}1 \end{bmatrix}$$

The elementary rotations can be combined into a single rotation matrix. The order of multiplication is crucial here. Since rotation occurs first around the z-axis and then around the y-axis, this order must be maintained during multiplication.

$$= \ddot{y} = \begin{bmatrix} \cos(\ ) & \ddot{y}\sin(\ ) & 0 \\ \sin(\ ) & \cos(\ ) & 0 \\ 0 & 0 & 1 \end{bmatrix} \ddot{y} \begin{bmatrix} \ddot{y}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \ddot{y}1 \end{bmatrix} = \begin{bmatrix} \ddot{y}\cos(\ ) & \ddot{y}\sin(\ ) & 0 \\ \ddot{y}\sin(\ ) & \cos(\ ) & 0 \\ 0 & 0 & \ddot{y}1 \end{bmatrix} \tag{6-25}$$

After first considering the rotation, we now take a look at the entire transformation. The following formula results for the transformation

$$= + \quad \ddot{y}\ddot{y}\ddot{y}\ddot{y}\ddot{y} \tag{6-26}$$

Here the unknown quantity is ÿÿÿÿÿ . This formula can also be written in coordinate notation be formulated.

$$\begin{pmatrix} \ \ 0 \ \end{pmatrix} = \begin{pmatrix} \ \ 0 \ \end{pmatrix} + \begin{bmatrix} \ddot{y}\cos(\ ) & \ddot{y}\sin(\ ) & 0 \\ \ddot{y}\sin(\ ) & \cos(\ ) & 0 \\ 0 & 0 & \ddot{y}1 \end{bmatrix} \ddot{y} \begin{pmatrix} \ddot{y} \\ \ddot{y} \\ 0 \end{pmatrix}$$

Since a vehicle movement is considered in the plane, the z-coordinate takes the value zero.
Another way to represent this equation is using the homogeneous transformation matrix.
The homogeneous transformation matrix consists of the rotation matrix and the position vector of the vehicle. (cf. Hofschulte, 2022: p.104)

$$0 \qquad = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \ddot{y}\cos(\ ) & \ddot{y}\sin(\ ) & 0 & 0 \\ \ddot{y}\sin(\ ) & \cos(\ ) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6-27}$$

This gives the equation

$$(\ 0_1\ ) = [\ \begin{matrix} \ddot{y}\cos(\ )\ \ddot{y}\sin(\ )\ 0\ 0\ \ddot{y}1\ 0 \\ \ddot{y}\sin(\ )\ \cos(\ )\ 0\ 0\ 0\ 0 \end{matrix} \quad \begin{matrix} \ddot{y} \\ 0 \end{matrix} \quad 0\ 1 \quad]\ \ddot{y}\ (\ \ddot{y}\ 1\ ) \tag{6-28}$$

As mentioned before, the unknown quantity here is ÿÿÿÿÿ . Therefore, the equation must be rearranged according to this quantity. This requires the calculation of the inverse homogeneous transformation matrix. This consists of the transposed rotation matrix and the product of the negatively transposed rotation matrix and the position vector of the vehicle. (cf. Hofschulte, 2022: p.104)

$$^{0}\quad \ddot{y}1 = \quad {}^{0} = [0\ 0 \quad 0 \quad 1 \quad] \tag{6-29}$$

With

$$= [\ \begin{matrix} \ddot{y}\cos(\ )\ \ddot{y}\sin(\ )\ 0 \\ \ddot{y}\sin(\ )\ \cos(\ )\ 0\ 0 \end{matrix} \quad \begin{matrix} 0 \\ \ddot{y}1\ ] \end{matrix}$$

and

$$= [\ \begin{matrix} \cos(\ )\ \sin(\ ) \\ \sin(\ )\ \ddot{y}\cos(\ )\ 0 \end{matrix} \quad \begin{matrix} 0 \\ 0 \end{matrix} \quad 0 \quad 1]\ \ddot{y}\ (\ \ 0\ ) = (\ \begin{matrix} \ddot{y}\ \cos(\ )+ \ddot{y}\ \sin(\ ) \\ \ddot{y}\ \sin(\ddot{y})\ddot{y}\ 0 \quad \ddot{y}\ \cos(\ ) \end{matrix}\ )$$

the inverse homogeneous transformation matrix is

$$^{0}\quad \ddot{y}1 = \quad \begin{matrix} \ddot{y}\cos(\ )\ \ddot{y}\sin(\ )\ 0\ 0\ \ddot{y}1\ 0 \\ \ddot{y}\sin(\ )\ \cos(\ )\ 0\ 0\ 0\ 0 \end{matrix} \quad \begin{matrix} \ddot{y}\ \cos(\ )+ \ddot{y}\ \sin(\ ) \\ \ddot{y}\ \sin(\ddot{y})\ddot{y}\ 0 \quad \ddot{y}\ \cos(\ ) \end{matrix}$$

^{0}

$$= [\ \begin{matrix} \\ \\ \end{matrix} \quad \begin{matrix} \\ \\ 1 \end{matrix} \quad]$$

The required path difference related to the vehicle is now determined by the following equation:

$$\begin{matrix} \ddot{y} \\ 0 \end{matrix}$$

$$(\ \ddot{y}1\ ) = [\ \begin{matrix} \ddot{y}\cos(\ )\ \ddot{y}\sin(\ )\ 0\ 0\ \ddot{y}1\ 0 \\ \ddot{y}\sin(\ )\ \cos(\ )\ 0\ 0\ 0\ 0 \end{matrix} \quad \begin{matrix} \ddot{y}\ \cos(\ ) + \ddot{y} \\ \sin(\ddot{y})\ \ddot{y}\ 0 \end{matrix} \quad \begin{matrix} \ddot{y}\ \sin(\ ) \\ \ddot{y}\ \cos(\ ) \end{matrix} \quad 1 \quad]\ \ddot{y}\ (\ 01\ ) \tag{6-30}$$

By multiplying out this formula, we obtain the vector for the path difference.

$$\begin{matrix} \ddot{y} \\ \ddot{y} \end{matrix}$$

$$(\ \ 0\ ) = (\ \begin{matrix} \ddot{y}\cos(\ )\ \ddot{y} \\ \ddot{y}\sin(\ )\ \ddot{y} \end{matrix} \quad \begin{matrix} \ddot{y}\ \sin(\ )\ \ddot{y} \\ +\cos(\ )\ \ddot{y} \end{matrix} \quad \begin{matrix} + \\ + \\ 0 \end{matrix} \quad \begin{matrix} \ddot{y}\ \cos(\ )+ \ddot{y}\ \sin(\ ) \\ \ddot{y}\ \sin(\ )\ \ddot{y} \quad \ddot{y}\ \cos(\ ) \end{matrix}\ )$$

The goal of the coordinate transformation has been achieved. Now the vector of the driving movement in relation to the vehicle can be specified.

6.3 Program Codes Sprint #2

Arduino and Python program codes are used to control the AGV's movement . The Python script is used to process the vehicle's tracking data and convert it into driving commands. This data is transferred to the vehicle via a serial interface, which executes the driving movement. The Python script is designed to integrate a function for position control and trajectory planning of a point-to-point movement. The following sections therefore go into more detail about the individual program codes.

6.3.1 Program code for driving movement in *Arduino*

The *Arduino board* is responsible for controlling the motors and thus for executing the driving movement. In order for the board to be able to pass the correct instructions on to the motors, it is necessary to load a corresponding program code onto the board. This program code specifies the direction of travel of the vehicle and regulates the speed. The *sprint backlog* shows that an overlay of the driving movement is required. This means that the vehicle should be able to drive in a certain direction and at the same time a rotational movement of the vehicle should be realized.

The program codes examined so far are formulated in such a way that no overlapping is possible. They work by setting a certain status. For example, if the status is set to "Forward", a forward movement is carried out. The command *Omni.setCarMove* is used for the movement itself.
This command uses information about the speed, direction of travel and rotation speed and processes it accordingly to control the motors. The information is passed to the *Arduino* via the serial interface of the Bluetooth module.

The program code 6-1 shows how this is implemented in the Arduino sketch.

```
79  void loop() {
80  while(Serial.available()==0){
81    }
82
83  cmd=Serial.readStringUntil('\n');
84  i=cmd.indexOf(':');
85  speed=cmd.substring(0,i).toInt();
86  j=cmd.indexOf(':',i+1);
87  direction=cmd.substring(i+1,j).toFloat();
88  k=cmd.indexOf('\n');
89  rotation=cmd.substring(j+1,k).toFloat();
90
91  Omni.setCarMove(speed,direction,rotation);
92  }
```

Program code 6-1: Reading and processing serial data for motion execution

Initially, no action is taken until data is transferred via the serial interface
The *Arduino* then receives data via the serial interface and reads it up to a

character “\n” (see chapter 5.3.3). The resulting data set is assigned to the variable command *“cmd” .* The command can, for example, look like this:

cmd=200:1.5783:0.0000\n

The first number is the speed. After the colon comes the direction in radians, then after the next colon comes the information about the rotational speed. Finally, the command is terminated with "\n". The character "\n" always indicates the end of a transmitted data set. It is necessary so that it is clear up to where the data should be read.

In order to be able to further process the information about speed, direction of travel and rotational speed, the command is split. The separated command parts are each assigned to a variable. The data set is separated using the *substring command.* As the command indicates, the data is still in the form of *strings .* However, the *setCarMove* command works with *integer* or float numbers. This means that a conversion to a number is required. This is done using the *"toInt"* or *"toFloat"* functions . The numerical values can now be passed to the *serCarMove* command and the movement can be carried out.

In order to receive data via the serial interface, a serial connection must first be established. This is done in the *void setup()* function (see program code 6-2).
The baud rate for data transmission is increased again to 57600 baud.5 The increase is necessary because the Python script needs to send a significant number of bits per second.
To ensure that no data is lost, the data transfer must be fast enough. The Bluetooth module is set to 56700 baud so that this is the case.

```
65  void setup() {
66  pinMode(13,OUTPUT);
67  Serial.begin(57600);
68
69  delay(2000);
70  TCCR1B=TCCR1B&0xf8|0x01;    // Pin9,Pin10 PWM 31250Hz
71  TCCR2B=TCCR2B&0xf8|0x01;    // Pin3,Pin11 PWM 31250Hz
72
73  OCR0A=100;
74  TIMSK0|=2;
75
76  Omni.PIDEnable(0.35,0.02,0,10);
```

Program code 6-2: Function void setup() for motion control

In addition to starting the serial interface, an LED is also declared in *void setup()* via *pinMode(13, OUTPUT) .* The LED is an LED integrated in the *board ,* which is called via pin 13. The LED is intended to serve as a graphical output and is related to the speed control.

In the previous chapters, the speed was controlled within the *void loop().* From the program code 6-1 it is clear that the command for the control is no longer contained in the loop. Previously, the control was always activated when a new loop run was carried out. However, this means that the control is called too rarely and the vehicle tends to oscillate. Accordingly, the control must be carried out more frequently. It is therefore integrated into a *timer . Timer* 1 and *Timer* 2 are already used for the motors (see chapter 3.2.2).

Therefore, the control should be integrated into an *interrupt service routine ISR* of *timer* 0. The *ISR* is executed whenever the value of *Output Compare Register OCR0A* is equal to the count value of *timer* 0 (TCNT0). In *void setup()* the value in register *OCR0A* is set to 100.
This means that whenever the timer reaches the compare value 100, the control is executed.
The trigger frequency depends on how the timer is set (see chapter 3.2.2).

So that the user can see the LED flashing as a graphic output, it should not be triggered every time the ISR is called. The LED's light frequency is reduced using a count variable. The count variable is incremented by one every time the ISR is called. As soon as it reaches a value of 100, the LED is activated. This means that the LED flashes 100 times as slowly as the control is called (see program code 6-3).

---

5 Chapter 5.3.2 explains how to increase the baud rate for the Bluetooth module

```
52  int count=0;
53  ISR(TIMER0_COMPA_vect)
54  {
55
56    count++;
57    if (count==100) {
58      digitalWrite(13,!digitalRead(13));
59      count=0;
60    }
61
62    Omni.PIDRegulate();
63  }
```

Program code 6-3: Interrupt service routine for timer 0

The main tasks of the program code have now been examined. In addition to these tasks, the motors are declared, the corresponding pins are assigned, the PWM frequency is set, etc. Since these parts of the program code are similar to the demo script or identical to the *"data exchange"* script from the first sprint, they will not be explained again here. The complete program code can be found in Appendix D.

6.3.2 Program code for attitude control

One requirement of the *sprint backlog* is the implementation of attitude control for the vehicle. Chapter 6.2.1 deals with the theoretical principles of attitude control and the derivation of the formulas. This chapter now deals with the implementation of the attitude control in the form of a Python script.

A new function "Position control" is defined for position control. The position control is carried out by calling this function. The position control function requires the position values xsoll and ysoll as well as the target orientation phisoll. These values are passed to the function when it is called.
Within the function, the values for the proportional controllers are first defined. Two separate values are distinguished for the translatory and rotary movement. The parameters are determined experimentally in this work. Once the script has been fully formulated, the parameters are determined through testing. They should be set so that they do not react too slowly to control deviations. At the same time, the vehicle should not overshoot too much.

In the next step, the data is requested from the tracking system via an HTTP request and the values of the returned JSON are assigned to the variables xFTS, yFTS and phiFTS . These values provide information about the position and orientation of the vehicle in the reference coordinate system (see program code 6-4).

```
8   def Lageregelung(x_soll,y_soll,phi_soll):
9       kp=1.2
10      k_phi=0.75
11
12      r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
13      obj = json.loads(r.text)
14      x_FTS = float(obj["x"])
15      y_FTS = float(obj["y"])
16      phi_FTS = float(obj["phi"])
17
18      alpha=phi_FTS-math.pi/2
```

Program code 6-4: Position control “Reading tracking data and correcting the angle”

After the values have been read from the tracking camera, it is necessary to convert the angle again. This is because the tracking system creates a coordinate system for a body when it detects two markers. The coordinate system points with the x-axis in the direction of the smaller marker (see Figure 6-5). However, in the vehicle, the markers are positioned so that the smaller marker is on the y-axis (see Figure 6-6). There is therefore a difference of ÿ/2. This is corrected by conversion within the script.

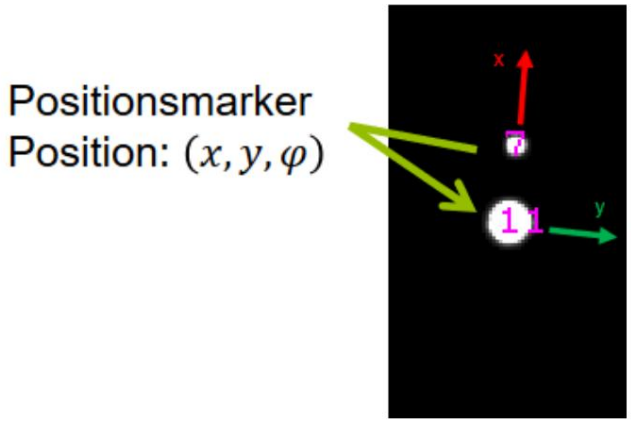


Figure 6-5: Marker position in tracking system (Hofschulte & Waldt, 2021: S10)

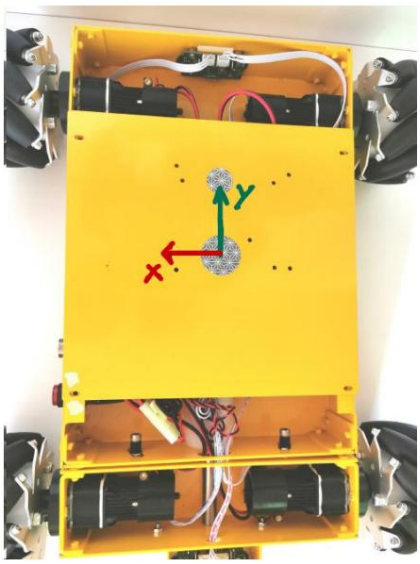


Figure 6-6: Marker position on vehicle

The system knows the position and orientation values for the target state. For the actual state, however, these refer to the reference coordinate system of the tracking system. However, in order to calculate the direction of travel, the target position must be converted to the body-fixed coordinate system of the vehicle (see Chapter 6.2.3). The transformation of the coordinate systems results in the vector for the target positions related to the vehicle:

$$\begin{pmatrix} \ddot{y} \\ \ddot{y} \\ 0 \end{pmatrix} = \begin{pmatrix} \ddot{y}\cos(\ddot{y}) & \ddot{y}\sin(\ddot{y}) & 0 \\ \ddot{y}\sin(\ddot{y}) & +\cos(\ddot{y}) & 0 \\ \ddot{y}\cos(\ddot{y}) + \ddot{y}\sin(\ddot{y}) & \ddot{y}\sin(\ddot{y}) & \ddot{y}\cos(\ddot{y}) \end{pmatrix}$$

This vector is used in the script, but the form is changed for clarity. The vector is rearranged so that the direct connection between the target position and the actual position
becomes apparent (see program code 6-5).

```
31      #Ziel bezogen auf FTS
32      dx=-math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS)
33      dy=-math.sin(alpha)*(x_soll-x_FTS)+math.cos(alpha)*(y_soll-y_FTS)
```

Program code 6-5: Position control “Conversion of the target position in relation to the vehicle”

In the next step, the speed for the control is calculated. First, the respective speeds in the x-direction and in the y-direction are required. These result from the path differences and the proportionality factor kp (see program code 6-6). The total speed is then calculated from the square root of the sum of the individual squared speeds. Once this has been determined, it can take on various values, including a value of 300mm/s. Since the maximum speed of the vehicle is 255mm/s, the limit would be exceeded. A control variable limit is therefore required. This is implemented with an if query. If the value is above 255mm/s, it is limited to 255mm/s, otherwise the value remains as before. Finally, the direction of travel of the translational movement must be determined. This is done using the *arctan2 function.*

```
26  #Berechnung der Geschwindigkeit
27  v_x=kp*dx
28  v_y=kp*dy
29
30  v_ges=math.sqrt(v_x**2+v_y**2)
31
32  if (v_ges>255):
33      v_ges=255
34  else:
35      v_ges
36
37  #Berechnung der Fahrrichtung
38  rad = math.atan2(dy,dx)
```

Program code 6-6: Position control “Calculation of driving speed and direction”

The next focus is on the rotational movement. First, the angular deviation between the target orientation and the actual orientation is determined. To reduce the angular deviation, it is necessary for the vehicle to always rotate in the direction that leads to a reduction in the control error.

When calculating the angular deviation, it can happen that this assumes a value greater than 180° or less than -180°. As a result, the vehicle would turn in a direction that would require a longer time to compensate for the control deviation. For example, if a control deviation of -200° is set, the vehicle turns 200° to the right. However, the path is much shorter if the vehicle were to turn anti-clockwise. When turning anti-clockwise, the deviation is only 160°.
It is therefore necessary to correct the angle in such cases. This is done using an *if* query. This fulfills exactly the function just described (see program code 6-7). After the value for the angular deviation has been identified from the if query, this is finally converted into an angular velocity using the proportionality factor kphi .

```
40  #Berechnung der Winkeländerung und Winkelgeschwindigkeit omega
41  d_phi=(phi_soll-phi_FTS)
42
43  if (abs(d_phi)>math.pi):
44      if (d_phi>0):
45          d_phi=-2*math.pi+d_phi
46      else:
47          d_phi=2*math.pi+d_phi
48  else:
49      d_phi=d_phi
50
51  omega=d_phi*k_phi #[1/s]
```

Program code 6-7: Position control “Calculation of angular velocity”

In the last part of the “position control” function, the information obtained about driving speed, driving direction and rotation speed is transmitted to the *Arduino* using the HC05 module.
This can convert the information into driving commands so that the vehicle executes its movement. To transfer the data, it is embedded in a command. The decimal places are limited here, as otherwise an unnecessary number of bits are used in the data transfer and the data set sent is larger than necessary. Therefore, the *float* numbers are limited to four decimal places. The direction of travel is transmitted by the *Arduino sketch* as *an integer number.*
processed, which is why no decimal places are required. How the data transfer works exactly is discussed in chapter 5.3.3 and therefore will not be explained in more detail here.

```
53      #Übergabe der Daten über serielle Schnittstelle
54      cmd="{:.0f}".format(v_ges)+":{:.5f}".format(rad)+":{:.5f}".format(omega)
55      cmd=cmd+'\n'
56      print(str(cmd))
57      arduinoData.write(cmd.encode())
```

Program code 6-8: Position control “Transfer of travel commands to *Arduino”*

The function of the positioning control is now completely defined. To execute this function, it must be called. This is done in an endless while loop. The parameters for the position xsoll and ysoll as well as the target orientation phisoll are also specified here. With the function *time.sleep()*
a time interval is defined which indicates after which period the position control should be carried out again. At this point a value of 0.5s is initially assumed.6

```
61  while True:
62      Lageregelung(-200,0,0)
63      time.sleep(0.5)
```

Program code 6-9: Attitude control “Calling the attitude control”

6 The complete program code for the attitude control can be found in Appendix E.

6.3.3 Program code for trajectory planning in Python

After the *Arduino sketch* for executing the movement and the position control for correcting position deviations have been implemented, the program code for trajectory planning between two points is now examined. For this, the derived formulas from chapter 6.2.2
In the following, the term “path planning” is used instead of “trajectory planning”.
used. This is due to an initial equation of the two terms. In the literature, path and trajectory are often incorrectly equated. Therefore, the term "path planning" was introduced for the program code. However, not only position and orientation are interpolated, but also the associated speeds. Therefore, trajectory planning is carried out in this chapter. Only the program code is referred to as path planning.

In the first step of the path planning algorithm, a target position and a target orientation are specified. From these, the path differences or the angle difference to the position in which the vehicle is located are calculated. The total distance to be traveled is also determined from the path differences in the x-direction and y-direction.

```
84  #======================Bahnplanung=============================================
85  #==============================================================================
86  def Bahnplanung(x_ziel, y_ziel, phi_ziel):
87      r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53")
88      obj = json.loads(r.text)
89      x_start = float(obj["x"])
90      y_start= float(obj["y"])
91      phi_start= float(obj["phi"])
92
93      print("Koordinaten FTS: " + str(x_start), str(y_start), str(phi_start))
94
95      s_x_gesamt=x_ziel-x_start
96      s_y_gesamt=y_ziel-y_start
97      s_gesamt=math.sqrt(s_x_gesamt**2+s_y_gesamt**2)
98      phi_gesamt=phi_ziel-phi_start
```

Program code 6-10: Path planning "Route calculation"

Next, the speed profile of the trajectory planning is defined. A speed profile with a ramp profile is to be programmed (see program code 6-10). To do this, the values for acceleration and speed are first defined. It is important that these do not exceed maximum values, i.e. for the translational speed, for example, the limit for a maximum speed of 255 mm/s must be observed and this must not be exceeded. The time of the individual movements is to be determined from the previous values, with the time in the x-direction and y-direction being combined into a total time for the translational movement. For synchronous path control, it is important that the vehicle reaches the position and orientation at the same time. Therefore, a comparison is made to determine which of the calculated times is the maximum time.

This is defined as the total time for the track (see program code 6-11).

```
'''
v
|
|   _________________
|  /|                |\
| / |                | \
|/__|________________|__\___t
    t_a              t_b
'''

v_trans=200 #mm/s
a_trans=a_x=a_y=150 #mm/s^2
teta=2.5   #rad/s^2
omega=3   #rad/s

t_ges_trans=s_gesamt/v_trans+v_trans/a_trans
t_ges_phi=abs(phi_gesamt)/omega +omega/teta

#für synchrone Bewegung müssen alle Positionen gleichzeitig erreicht werden
#bestimmen, welche Komponente am längsten braucht und diese als neue Zeit festlegen
t_ges=max(t_ges_phi,t_ges_trans)
```

Program code 6-11: Path planning “Calculation of the maximum time”

Once the new time has been determined, the new speeds are determined from it. This time, a distinction is made between the speeds in the x-direction and the y-direction. This is necessary in order to later determine the respective x-coordinates and y-coordinates of the interpolated points. In addition, the acceleration times for each component and the time at which the movement begins to decelerate in order to achieve a speed of zero at the target are determined (see.
program code 6-12).

```
#Berechnen der neue Geschwindingkeiten bezogen auf die neue Zeit
v_x_neu=((a_x*t_ges)/2-math.sqrt((a_x**2*t_ges**2)/4-abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt
v_y_neu=((a_y*t_ges)/2-math.sqrt((a_y**2*t_ges**2/4)-abs(s_y_gesamt)*a_y))*abs(s_y_gesamt)/s_y_gesamt
omega_neu=((teta*t_ges)/2-math.sqrt((teta**2*t_ges**2/4)-abs(phi_gesamt)*teta))*abs(phi_gesamt)/phi_gesamt

#Beschleunigungszeiten
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta

""" ...

#Zeit ab der abgebremst werden soll
t_bx=t_ges-t_ax
t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi
```

Program code 6-12: Path planning “Calculation of speeds and times”

The final preparations for the interpolation are then made. Empty lists are set up for this. On the one hand, lists are provided for the respective values of the coordinates and the angle, and on the other hand, lists for the speed components. The two empty lists for the x-component of the target positions and the speeds in the x-direction at the individual target positions are described in program code 6-13. In addition to the lists, a time *"delta_t"* is specified. This time specifies after how many seconds the next interpolation point should be determined. In addition, a time value of 0s is assigned to the actual position. The number of interpolated points that should be passed through is calculated from the interval time *"delta_t"* . This usually does not result in an even number. Therefore, it must be rounded. The problem that arises is that there is a shorter or longer time span between the last interpolated point and the target point than between the other points. This problem is solved by correcting the time interval *"delta_t"* . The time *“delta_t”* is now the total travel time divided by the

Number of points to be interpolated. The advantage of this double value assignment is that the time interval always has a similar value, here for example around 0.1s. This is useful for the position control, which is to be integrated into the path planning, because the proportionality parameters are best set to a defined time interval. If a number of interpolated points were to be specified from the outset without taking the length of the entire route into account, a time interval would result depending on the route length. This can cause the interpolation times to vary greatly.

```
176     x_soll=[]
177     vx_soll=[]
178 >   """     ...
185     t_=[]
186
187     delta_t=0.1 #Intervallzeit
188     t=0 #Startzeitpunkt
189
190     n=int(round((t_ges/delta_t),0))
191     delta_t=t_ges/n
```

Program code 6-13: Path planning "Lists for interpolation"

Since all preparations for interpolation have already been made, the interpolation of the individual points follows. A new function "Interpolation" is introduced for interpolation (see.
Program code 6-14). This is passed the time which indicates the interpolation time.
In addition, the acceleration time, braking time, acceleration and speed are assigned to the function. The function first checks whether the speed is negative. In this case, the acceleration also has a negative sign. The equations for interpolating the position and speed are then defined. These differ depending on the phase (see Chapter 6.2.2). Finally, the calculated position and speed are returned.

```
71  def Interpolation(t,t_a,t_b,a,v):
72      if v<0:
73          a=-a
74      else:
75          a
76
77      if(t <=t_a):
78          x=0.5*a*(t)**2
79          v_neu=a*t
80
81      if(t_a<t<=t_b):
82          x=v*t-0.5*v**2/a
83          v_neu=v
84
85      if(t>t_b):
86          x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2
87          v_neu=v-a*(t-t_b)
88
89      return [x,v_neu]
```

Program code 6-14: Path planning "Function for interpolation"

The "interpolation" function is now integrated into a for loop (see program code 6-15). The x-coordinates, y-coordinates and orientation are interpolated within the for loop. The "interpolation" function is called for each of these and the function's return values are assigned to a distance traveled and a speed. In the example for the x-coordinate, these are the variables x_t and vx_soll_neu. The x-coordinate at the time in question x_soll_neu is then again derived from the distance traveled x_t plus the start coordinate. Finally, the determined values are added to the lists for the interpolated coordinates or orientation and speeds. In addition, the time in question is increased by "delta_t". The for loop is run through until the variable z is reached. This variable is derived from the number of interpolated points n plus 1. This not only adds the interpolated points themselves to the list, but also the start and destination points.

```
195     z=n+1
196
197     for w in range(z):
198         [x_t, vx_soll_neu]=Interpolation(t,t_ax,t_bx,a_x,v_x_neu)
199         x_soll_neu=x_t+x_start
200         x_soll.append(x_soll_neu)
201         vx_soll.append(vx_soll_neu)
202
203         [y_t, vy_soll_neu]=Interpolation(t,t_ay,t_by,a_y,v_y_neu)
204         y_soll_neu=y_t+y_start
205         y_soll.append(y_soll_neu)
206         vy_soll.append(vy_soll_neu)
207
208         [phi_t, omega_soll_neu]=Interpolation(t,t_a_phi,t_b_phi,teta,omega_neu)
209         phi_soll_neu=phi_t+phi_start
210         phi_soll.append(phi_soll_neu)
211         omega_soll.append(omega_soll_neu)
212
213         #t_.append(t)
214         t+=delta_t
```

Program code 6-15: Path planning "Determining the interpolated points and speeds"

To clarify, the execution of a run through the for loop is described using the x coordinate. Initially, the time *t* has the value zero. This means that within the function

“Interpolation” in the first if-loop speed and distance at time *t=0*
determined. These values are returned and the determined value for the distance in the x-direction is added to the starting value of the vehicle. The starting value is always at the initially tracked position of the vehicle. This results in the starting value of the vehicle for the first value to be added to the list. The time *t* is then increased by the interval time *"delta_t"* and the next run through the for loop begins.

Once the interpolated positions and orientations are known, they should now be traversed. This is done using a while loop including an if query. At the beginning, a running variable k is defined. This should run through the lists with the interpolations. The if query ensures that new interpolations are only read from the lists as long as the list is long. Within the if query, the position control is called first. The position control is given the values for the target position and target orientation. It should use this to determine a control deviation and return values for the additional speeds.

```
251     k=0
252     while True:
253         if (k<(z-1)):
254
255             d_v=Lageregelung(x_soll[k],y_soll[k],phi_soll[k])
256             print("Werte aus Lageregelung: " +str(d_v))
257             d_v_x=d_v[0]
258             d_v_y=d_v[1]
259             d_omega=d_v[2]
```

Program code 6-16: Path planning "Running through the position control"

For this purpose, it is necessary to slightly modify the attitude control designed in chapter 6.3.2 to redesign. In chapter 6.3.2, the position control is designed so that the path and angle differences are calculated. From this, the speeds and direction of travel are then calculated and transmitted to the vehicle via a serial interface. Now, instead of forwarding the information via an interface, it should be returned as the function's *"return"* . The speed in the x direction, the speed in the y direction and the angular speed are returned as values. This return is output as a list and can be used within the path planning. In program code 6-16, the return is assigned to the variable *"d_v"* and then the individual elements of the list, i.e. the speed in the x direction, the speed in the y direction and the angular speed, are assigned to new variables.

The values for the speeds from the position control are added to the interpolated speeds, so that if a control deviation is detected, it can be corrected immediately in the best case. For the translatory speed, a control variable limit of 255 mm/s is necessary (see program code 6-17).

```
270             v_x_korr=vx_soll[k]+d_v_x
271             v_y_korr=vy_soll[k]+d_v_y
272             omega_korr=omega_soll[k]+d_omega
273
274             v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)
275
276             if (v_ges_korr>255):
277                 v_ges_korr=255
278             else:
279                 v_ges_korr
```

Program code 6-17: Path planning "Correction of speed values"

In the last step of the if query, the direction of travel to the interpolation point is calculated from the corrected speeds based on the vehicle coordinate system (see program code 6-18).
Therefore, all information is available to perform a movement up to the interpolation point.
In addition, the variable k is increased by one so that the next interpolation point can be reached in the next run. After the if query, the data is transferred to the vehicle's *Arduino board* via the serial interface. This is followed by a *time.sleep(delta_t) function,* which causes the system to pause for the interval time *"delta_t"* before the next run is started. This gives the vehicle the appropriate time to carry out the driving movement.

```
282             rad = math.atan2(v_y_korr,v_x_korr)
283             k=k+1
```

Program code 6-18: Path planning "Calculation of the direction of travel"

After all the interpolated points in the list have been run through, the vehicle should be at the target position with the target orientation. To ensure that the vehicle continues to hold its position, a while loop was introduced at the beginning. This consists of the if loop already considered and an *"else statement"*. Since the if query has been completely run through, the *"else statement"* is now considered with each subsequent run. The position control is integrated into this again (see program code 6-19) so that the vehicle can hold its position. The position control is only switched off when the process is aborted.7

```
287         else:
288             d_v=Lageregelung(x_ziel,y_ziel,phi_ziel)
289             print("Werte aus Lageregelung: " +str(d_v))
290             d_v_x=d_v[0]
291             d_v_y=d_v[1]
292             omega_korr=d_v[2]
293             rad = math.atan2(d_v_y,d_v_x)
294             v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)
```

Program code 6-19: Path planning "Holding the target position and target orientation"

7 The complete program code for path planning or trajectory planning can be found in Appendix F.

7 Autonomous Path Planning

Path planning or trajectory planning involves determining a path from a starting point to a destination. Previously, the path was designed as a direct connection between the start and destination; now the path is to be traversed via so-called via points. These specify the path and must be driven through. The term autonomy is not clearly defined in the literature. For this work, it is defined as independence. In the context of path planning, this means that a path between the start and destination is not simply specified by a user. Instead, an algorithm is used to determine a suitable path so that the vehicle can "independently" find its way to the destination. This chapter deals with autonomous path planning. To do this, the requirements for the sprint are first listed and described in a sprint backlog.

Then, various path planning algorithms are considered and a suitable algorithm is selected. This is then implemented. In addition, the path planning or trajectory planning script from Chapter 6 is extended so that a travel movement can be carried out for a multi-point movement. Finally, all of the subprograms of this work are combined so that autonomous motion control is achieved.

7.1 Sprint Backlog #3

Table 7-1 contains the requirements for the third *sprint.* The first two requirements deal with driving along a given path and autonomously finding a path to a given point. Therefore, a path planning algorithm must be implemented that

independently searches for a path in an environment to a defined destination. The vehicle must then be able to travel this path. Because a constant speed curve was previously required in the *sprint ,* this must also be introduced for the driving movement here. Another requirement is driving in a static environment. There are three types of obstacles. Firstly, there are static obstacles, secondly, semi-static obstacles and finally dynamic obstacles. Static obstacles are stationary and immobile, semi-static obstacles remain in the same place for a certain period of time but are then moved to another location and dynamic obstacles move freely in the environment (cf. Künemund, 2017: p.24). In this work, only stationary, i.e. static, obstacles will be considered.

Table 7-1: Sprint Backlog #3

Sprint Backlog #3	
Nr.	**Anforderung**
1	Fahren entlang eines vorgegebenen Pfades über mehrere Punkte
2	Autonomes Suchen eines Pfades zu einem vorgegebenen Punkt
3	Fahren in einer statischen Umgebung
4	Kollisionsfreies Fahren im freien Konfigurationsraum
5	Definierbarer Arbeits- und Konfigurationsraum
6	Kompensation von Bewegungsungenauigkeiten
7	Erstellen von virtuellen Hindernissen

The next two requirements address the working and configuration space. Collision-free driving in a free configuration space should be possible and the working and configuration space should be clearly and easily defined. Therefore, the first question is what exactly a working and configuration space is. From a physical point of view, the working space describes the space in which the AGV moves (cf. Künemund, 2017: p. 25). In this work, this is the entire space that can be captured by the tracking system. In robotics, the configuration space is defined as the set of all configurations that a robot can assume. Since the vehicle can move on the plane without restrictions, in this case the configuration space corresponds to the working space. However, the configuration space is further divided into free and occupied configuration space. The free configuration space represents the set of all configurations that the vehicle can assume. The occupied configuration space is the set of all configurations that are occupied by an obstacle (cf. Hinzmann, 2011: p.39f). The requirement of collision-free driving requires driving in the free configuration space. In addition, a query is required when searching for a path to ensure that the planned path does not run through the occupied configuration space. This requires a clear definition of the free configuration space. In this work, however, only virtual obstacles will be considered. This means that these must be created by the user themselves.

The last requirement deals with the movement inaccuracies. The movement inaccuracies must be compensated. When executing planned movements, there are often deviations from the planned trajectory, e.g. due to wheel slippage. These must be corrected in order to reach the goal (cf. Künemund, 2017: p.24). In the second *sprint*

Attitude control has already been introduced for this purpose. This will also be integrated into path planning.

7.2 Path planning algorithms

There are various algorithms for path planning. They differ in the way they search for a path and select a suitable path. The large number of different path planning algorithms can be divided into various classes. Three of these classes will therefore be considered. The first class is the cell-based methods, the second class is the potential field methods and the third class is the sampling-based methods. The individual methods will be briefly discussed below and examples will be explained. A suitable method will then be selected.

cell-based methods

In cell-based methods, the working space or the free configuration space is divided into cells. The size and geometry of the cells vary depending on the respective algorithm.
Neighboring cells are entered into a graph, with the nodes of the graph corresponding to the cells. Individual nodes are connected within the graph. Only neighboring nodes can be connected with an edge. After the graph has been constructed, the cell in which the starting and target points are located is determined. A graph is then sought between these. (Hinzmann, 2011: p.42)

One of the best known and most common cell-based methods is the A* algorithm (see Figure 7-1). Here, the workspace is divided into cells as described. The accuracy of the algorithm depends on the cell size, although different cell sizes can also be defined.
First, the start and destination nodes are determined and the obstacles are defined. One of the most important aspects is then the cost function. This means that the costs are determined for each cell. The costs consist of the distance traveled and the expected costs for the route to the destination. For example, if the fourth node or cell is considered as in Figure 7-1, it has covered a distance of 4 cells from the starting point. The previous costs therefore amount to 4. The future costs to the destination are then estimated. From cell 4 to destination 19, the diagonal across the obstacle is approximately 7. The total costs amount to 11. In this way, the costs of neighboring cells are always weighed against each other and the path with the shortest costs is selected. (see Swift, 2017)

The main advantage of this algorithm is that it is easy to implement with the same cell division. In addition, the optimal or shortest path to the target is always found. The disadvantage, however, is that the computational effort increases, especially with larger workspaces and the most precise possible mapping of the configuration space, and the method becomes unsuitable for systems that require fast path planning.

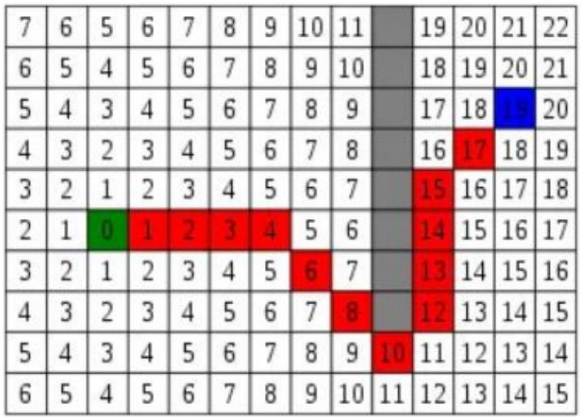


Figure 7-1: Cell division using an A-star algorithm
(Swift, 2017)

potential field method

The idea of potential fields is one of the first approaches to path planning. The idea of the potential method is based on the fact that particles move in space under the influence of a potential field
move. Therefore, in this process, every object is considered an obstacle with an artificial force field. The vehicle is a point mass that moves in space under the influence of the potential fields. The start represents a repulsive potential, whereas the target lies in a potential sink and therefore has an attractive effect. Obstacles are a potential mountain and thus generate a repulsive force. In every configuration, a total force acts on the vehicle, from which the torque of the motors is determined at any time. Figure 7-2 illustrates the process again. The blue area here represents a potential sink. The target is in this area. The start and the obstacles are potential mountains. The total potential of the space is obtained by superimposing the individual potential fields. (cf. Koren & Borenstein, 1991: p.1f)

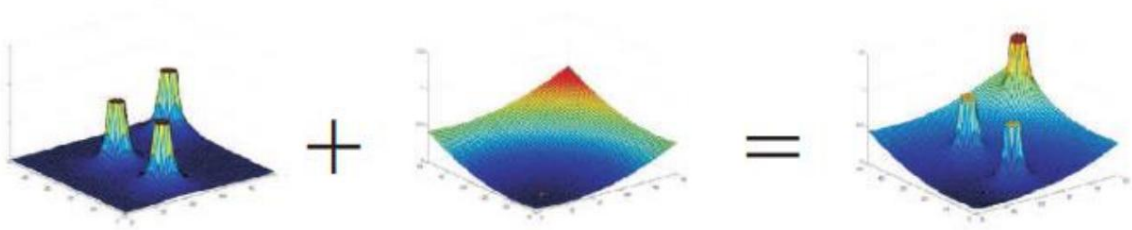


Figure 7-2: Total potential
(Kavraki & LaValle, 2016: p.145)

In environments with few or no obstacles, quick solutions can be found. However, there is a risk that local minima cannot be overcome. In addition, the mapping of the potentials in a Cartesian space is very complicated and as the number of obstacles and thus narrow passages increases, the vehicle can oscillate.

To avoid this, a great deal of effort must be put into creating the potential fields. (cf. Koren & Borenstein, 1991: p.7f)

sampling-based methods

The sampling-based methods are another subclass of path planning algorithms. Here, configurations are randomly searched from the configuration space. The focus is on the rapid expansion and exploration of the entire configuration space. There are various sampling strategies for this. These are designed to cover the configuration space according to certain criteria. The optimum criterion here is to find a path as quickly as possible. This path search can be accelerated using certain heuristics. The pure planning for a simple problem can therefore be done within seconds. However, such a path often requires optimization steps to smooth out the path or to remove detours. (cf. Leidner, 2011: p.8)

In the following, the two methods RRT and PRM will be presented in more detail in order to better understand the sampling-based methods.

Probabilistic Roadmaps PRM

Probabilistic *roadmaps* are a variant of the sampling-based methods. In PRM, the path in the configuration space is determined in two steps. In the first step, a so-called *roadmap* is created. This represents a kind of road map. This step is also known as the construction phase. Collision-free configurations are randomly created and then linked together by a planner. The nearest neighbors are linked together, creating a dense network. This network represents a map in which the configuration space can be traversed in different ways without collisions (see Figure 7-3). In the second step, an attempt is made to connect a starting configuration with a target configuration using graph search. (cf. Choset, et al., 2005: pp. 203-208)

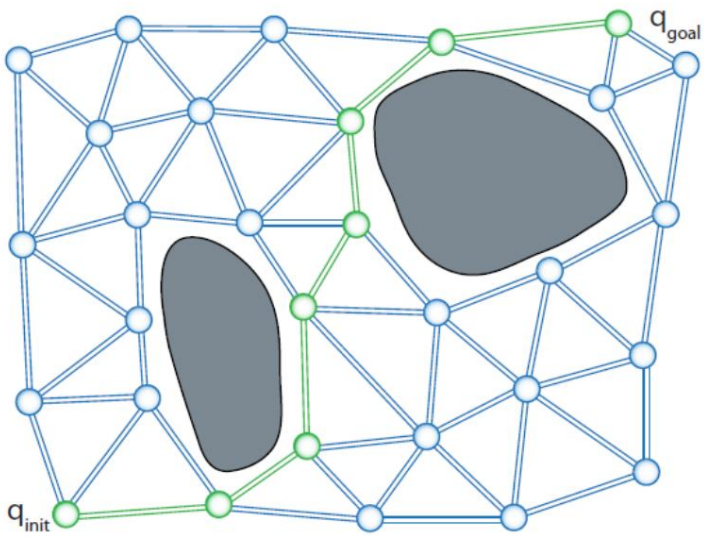


Figure 7-3: Creating a *Probabilistic Roadmap*
(Leidner, 2011)

As long as the scene does not change, *roadmaps* that have been created once can be used again and again for path planning. However, if no result is found between the start and the destination several times during the exploration phase, it is possible to extend the roadmaps, for example to find a suitable path by further exploring the configuration space. If the environment changes, the algorithm can also be extended to include a new collision query at runtime. *Probabilistic roadmaps* are particularly suitable for scenarios in which the same task is carried out again and again. The detailed construction of the map takes a lot of time, but it allows feasible paths to be found more quickly afterwards. However, the method is unsuitable for constantly changing environments, as the entire map would have to be constantly checked. (cf. Bohlin & Kavraki, 2000: p. 527f)

Rapid Exploring Random Tree

The class of *Rapid-Exploring Random Trees (RRT)* is another family of algorithms for random exploration of the configuration space. Unlike PRMs, RRTs establish a direct connection between individual nodes, meaning that each node has a predecessor, except for the start node (see Figure 7-4). The idea behind RRTs is to explore the configuration space
quickly with a search tree. Branches are created on the search tree until a connection is found between the start and target configuration. The start and target points as well as the obstacles on the map are known. Nodes are randomly selected in the workspace and checked to see if they are in the free configuration space. If the node is in the free space, it is connected to the densest node or edge of the tree. If the edge is

accepted because it does not collide with any obstacle, a new branch is created on the tree. The algorithm is repeated until the tree contains start and destination nodes. It is then checked which of the tree branches provides a suitable path between start and destination.
(cf. Kuffner James & LaValle, 2000: p.2f)

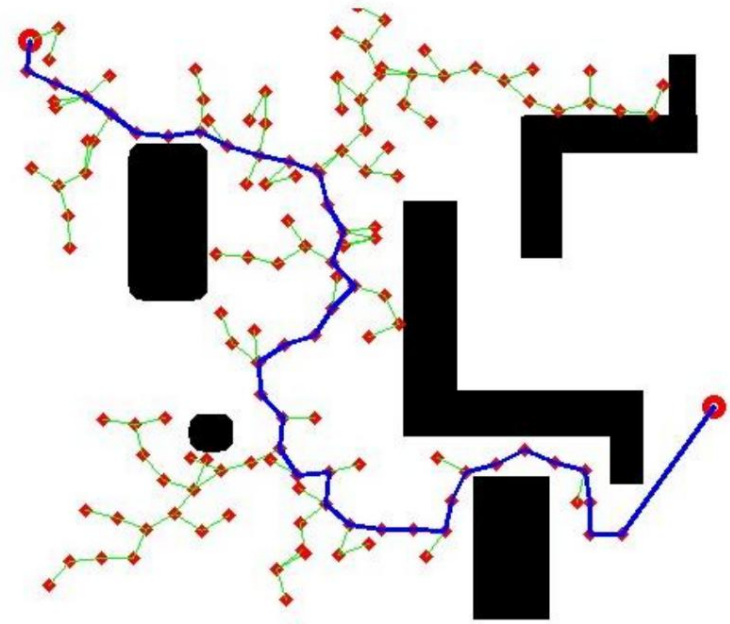


Figure 7-4: Rapid Exploring Random Trees
(Yadav, 22)

The RRTs are particularly well suited for queries where the environment subsequently changes. An unexplored space is quickly explored and a feasible path is found directly. In contrast to PRMs, there is no invalid network after a change in the configuration space, as it is constantly regenerated. However, this also makes the method a little slower than the PRMs. (cf. Leidner, 2011: p.11)

selection of a suitable procedure

In the previous sections, various algorithms for path planning were considered. Now a suitable algorithm will be selected for this work. Some advantages and disadvantages of the individual algorithms have already been briefly discussed. Table 7-2 compares these again and adds further advantages and disadvantages.

Table 7-2: Comparison of path planning methods

Vergleich der Verfahren	Vorteile	Nachteile
A-Stern	Einfach zu implementieren	hoher Rechenaufwand
	Reduktion auf einfache Graphensuche	hohe Kartenauflösung führt zu hohem Speicherbedarf
	Kürzester Weg zwischen Start und Ziel	Fahrzeug muss kleiner als Zellgröße sein
Potentialfeld	Anschauliche Idee	hoher Implemtierungsaufwand
	Geeignet für keine bis wenig Hindernisse	schwierig geeignete Potentialfunktion zu bilden
	für statische und dynamische Umgebungen	ggfs. Ziel unerreichbar
	glatte Pfade werden erzeugt	Oszillationsgefahr
PRM	sehr effizient und schnell	Optimierungsschritte notwendig
	für komplexer Probleme geeignet	Abstand zu Hindernissen nicht maximal
	schnelle Suche in Echtzeit	ungeeignet für veränderliche Umgebung
	relativ leicht zu implementieren	-
RRT	sehr effizient und schnell	Optimierungsschritte notwendig
	auf dynmische Umgebungen erweiterbar	Abstand zu Hindernissen nicht maximal
	für komplexe Probleme geeignet	-
	relativ leicht zu implemitieren	-

With the cell-based A-Star method, the problem of path planning is reduced to a simple graph search. This is easy to implement. The path determined is always the shortest path between the start and destination configuration. However, the map material must first be discretized. To do this, the vehicle size must be smaller than the cell size, otherwise the position is not precisely defined. The better the map resolution, the more memory is required. In addition, the computational effort increases with higher map resolution and higher number of obstacles and is therefore no longer as efficient.

The potential field method is impressive due to its clear idea and is also suitable for environments with few or no obstacles. In addition, smooth paths are generated directly, so that no further optimization steps are necessary. In contrast, the implementation effort is high, especially since it is difficult to create a suitable potential function. Furthermore, local minima can occur and the target may not be reachable. If there are many obstacles, there is also a risk of the vehicle oscillating.

The PRM is particularly suitable for complex problems. After a map has been created in the construction phase, the method works very quickly and efficiently, so that even a graph search can be carried out in real time. In addition, the method is relatively easy to implement, whereby the word relatively refers to the comparison with the A-star algorithm, which is even easier to implement. The disadvantage of this method is that further optimization steps for smoothing or
It is also not suitable for changing and dynamic environments. At the same time, it does not search for the path with the maximum distance to the obstacles.

The advantages and disadvantages of the RRT are similar to those of the PRM, as both methods are sampling-based approaches. The RRT also works very efficiently and quickly. It is also suitable for complex problems and relatively easy to implement. However, compared to the PRM, the graph search does not run in real time. Further optimization steps are also necessary after path finding. The distances to the obstacles are not maximum. The duration of the path search depends on the complexity of the environment and can therefore vary. It is advantageous that the method can be extended to dynamic environments.

The advantage-disadvantage comparison shows that the suitability of a procedure depends on the task and the environment. Sampling-based approaches are particularly suitable for this work. PRM and RRT are equally applicable for the workspace considered here. However, the later purpose of the AGV is not known. It can therefore be used in a dynamic or static environment, although the system is likely to be expanded for dynamic environments. Therefore, in this case, the RRT method is preferable to the PRM method and should be implemented accordingly.

7.3 Implementing the path planning algorithm

After selecting an RRT algorithm for path planning in Chapter 7.2, this should be implemented. However, even with the RRT algorithms there are different types of path search depending on the heuristics chosen. Two heuristics are defined for the RRT algorithm in this work. Firstly, there is no directed path search in the direction of the goal, i.e. the nodes in the space are randomly selected. No direction of expansion of the tree is specified. This minimizes the complexity of the algorithm. In order to achieve the most efficient path search possible, a path check towards the goal should be carried out. After a new node has been added to the tree, it should be checked whether a collision-free path from the last node, i.e. to the goal, is possible. If this is the case, the path search is already complete, otherwise it is continued. The exact functionality and procedure for the RRT algorithm is explained using the program code. However, only excerpts of the code are examined.

First, two new functions are defined. These two functions have the task of checking whether a new node or the target node is in the free configuration space (see program code 7-1). The first function "inObstacle" is given a point and calculates the distance of the point to every obstacle in the room. A new list is created for this purpose. This stores whether the point is in an obstacle with "True" or whether it is in free space. If this is the case, the Boolean operator "False" is entered in the list. The list with the Boolean operators "False" and "True" is returned as the return value. The function "inWorkspace", on the other hand, works with a status of "AR_OK".

If this status is set to "True", the point is in the workspace. First, the status is set to "True". Then, if queries are used to check whether it is actually between the specified limits of the workspace. As soon as an x-coordinate or y-coordinate is outside the limits, the status is set to "False". Finally, the status is returned.

```
592  def inHindernis(p_check):
593      in_Hindernis=[]
594      for a in range (H):
595          distanz=berechne_distanz(Hindernisse[a][0],p_check)
596
597          if distanz <= Hindernisse[a][1]:
598              in_Hindernis.append(True)
599
600          else:
601              in_Hindernis.append(False)
602
603      return in_Hindernis
604
605  def inArbeitsraum(p1):
606      print ("Funktion AR")
607      AR_OK=True
608      print("p_x: "+ str(p1[0]))
609      print("p_y: "+ str(p1[1]))
610      if AR_x[1] <p1[0] or p1[0] < AR_x[0]:
611          AR_OK=False
612      if AR_y[1] <p1[1] or p1[1] < AR_y[0]:
613          AR_OK=False
614
615      return AR_OK
```

Program code 7-1: Path planning "Functions for AR control and obstacle control"

Next, two functions will be introduced that describe the position of a point in relation to two other points or to the vector between them. This is necessary to check whether a new edge between two nodes passes through an obstacle. However, this requires some background knowledge first.

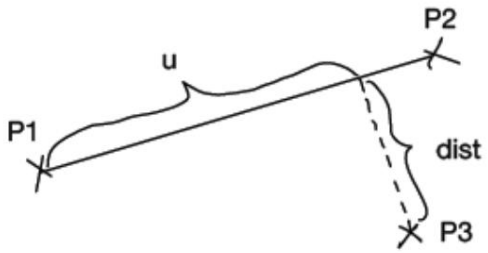


Figure 7-5: Position of three points in space according to Borke, 1988

Figure 7-5 shows the task. Three points P1, P2 and P3 are given. Points P1 and P2 represent the nodes in the RRT tree, while point P3 acts as an obstacle.
A line is drawn between the two points P1 and P2. This is calculated using equation 7-1. (see Borke, 1988)

$$= 1 + (2 \ddot{y} 1) \quad (7\text{-}1)$$

In this equation, u is the unknown quantity. This indicates the position on the line where a certain point is located. In this case, we are interested in where the point P3 is located in relation to P1 and P2. To do this, we first use the formula to calculate the quantity u and then calculate the x-coordinate and y-coordinate of the intersection point. The quantity u here takes on a value between 0 and 1.

$$= \frac{(3\ddot{y} 1)\ddot{y}(2\ddot{y} 1)+(3\ddot{y} 1)(2\ddot{y} 1)}{(1\ddot{y} 2)2} \quad (7\text{-}2)$$

$$= 1 + \ddot{y} (2 \ddot{y} 1) \quad (7\text{-}3)$$

$$= 1 + \ddot{y} (2 \ddot{y} 1) \quad (7\text{-}4)$$

The function that describes this calculation in the code is defined as "Point_to_Line". Since the functionality has already been explained, the function itself will not be discussed in detail here. However, it is used for another function "Collision_Path_Obstacle" (see
Program code 7-2). This function goes through the list of all obstacles again and checks whether an obstacle is too close to the new path. This path is generated from two points of the RRT tree. It should also be checked whether an obstacle is between the points of the new path or outside of it. For this purpose, two statuses "too close" and "between" are defined. These initially have the value "False", which means there is no collision between the path and the obstacle.

In addition, a “line collision” list is introduced, which stores the status for each obstacle.
The size u and the distance from the obstacle to the path are then determined for each obstacle using the “point_to_line” function. It is then checked whether u has a value between 0 and 1. If so, the “in between” status is set to “True”. It is then checked whether the distance between the path and the obstacle is smaller than the radius of the obstacle. If the distance is smaller than the radius of the obstacle, the “too_close” status is set to “True”. The path is only in the obstacle if both statuses are “True”. The status “True” is therefore added to the “LineCollision” list, otherwise the status “False”. As soon as the path crosses just one obstacle, the new path is discarded. This will be explained later. The function returns the list of statuses.

```
633  def Kollision_Pfad_Hindernis(p1,p2):
634      LinienKollision=[]
635      for s in range(H):
636          zu_nah=False
637          dazwischen=False
638
639          u, dist =Punkt_zu_Linie(p1,p2,Hindernisse[s][0])
640
641          if 0<=u <=1:
642              dazwischen=True
643
644          if dist <= Hindernisse[s][1]:
645              zu_nah=True
646
647          if zu_nah and dazwischen:
648              LinienKollision.append(True)
649          else:
650              LinienKollision.append(False)
651
652      return LinienKollision
```

Program code 7-2: Path planning "Collision control function"

In the next step, the workspace is defined and obstacles are created. In this work, the workspace is limited to the size of the calibration template, as the test environment is quite small. The sprint requirements defined that virtual obstacles would be created. In this work, therefore, no obstacles are initially detected by the tracking camera. Instead, the user enters the position and size of the obstacle (see program code 7-3). To do this, the user is first asked how many obstacles the user would like to have in their environment. The number of obstacles is assigned to the variable H. An empty list is then created for the obstacles. This is then filled. To do this, the user enters the respective position and radius of the obstacle for each obstacle. These parameters are determined for each obstacle and then added to the "Obstacles" list.

```
657  AR_x=(-525,525)
658  AR_y=(-320,320)
659
660  #Hindernisse erzeugen
661  H=int(input('Bitte gib Anzahl an Hindernissen ein: '))
662  Hindernisse=[]
663  R_FTS=550/2
664  for i in range (H):
665      x_Hindernis=float(input("x_Koordinate des Hindernis "+str(i+1)+":"))
666      y_Hindernis=float(input('y_Koordinate des Hindernis '+str(i+1)+': '))
667      MP_Hindernis=(x_Hindernis,y_Hindernis)
668      R_Hindernis=float(input("Radius des Hindernis "+str(i+1)+": "))+R_FTS
669      HindernisPara=[MP_Hindernis,R_Hindernis]
670      Hindernisse.append(HindernisPara)
671      print("Hinderniss: "+str(Hindernisse))
```

Program code 7-3: Path planning "Definition of the configuration space"

Before the radius can be added to the obstacle parameter list, it is added to the radius of the vehicle. The reasoning behind this is illustrated in Figure 7-6. When checking whether a point is in the obstacle, it must be taken into account that this point could later be driven to by the vehicle. However, only the coordinate of the center of the vehicle is considered. If the center of the vehicle is outside the obstacle, this does not mean that the entire vehicle is outside the obstacle. Therefore, the occupied configuration space is directly adjusted to include the radius of the vehicle.

This extension means that it is sufficient to only consider the center of the vehicle. The radius is calculated from half the diagonal of the chassis.

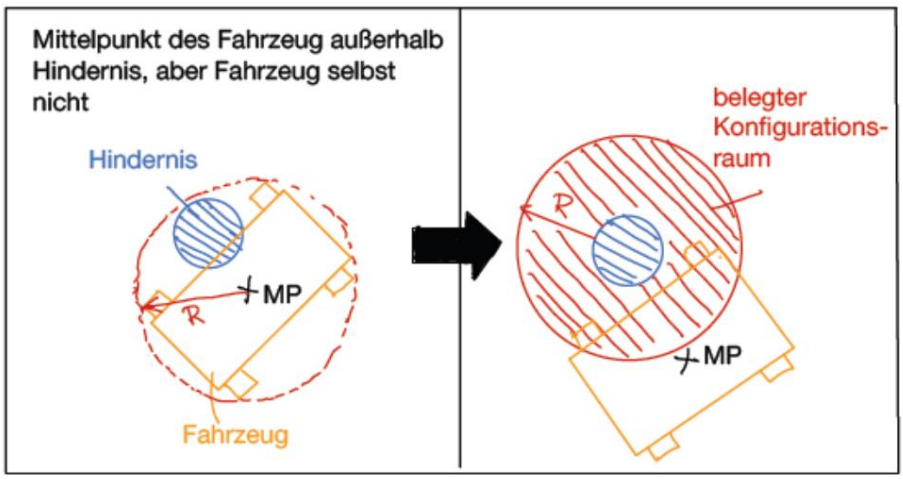


Figure 7-6: Adjusting the occupied configuration space

Next, the starting point is determined and the end point is defined (see program code 7-4). The coordinates of the starting point are requested and output using an http request, as in the previous chapters. The position is now added to the point "Start". This is given the name "k0".
A new list is then introduced that stores all new points. The information name, coordinates and *parent* are stored for each point. The term *parent* always refers to the previous point, i.e. the "parent" point. Since the starting point is the first point in the list and in the RRT tree, it has no *parent.* The parent is therefore specified as *None* .

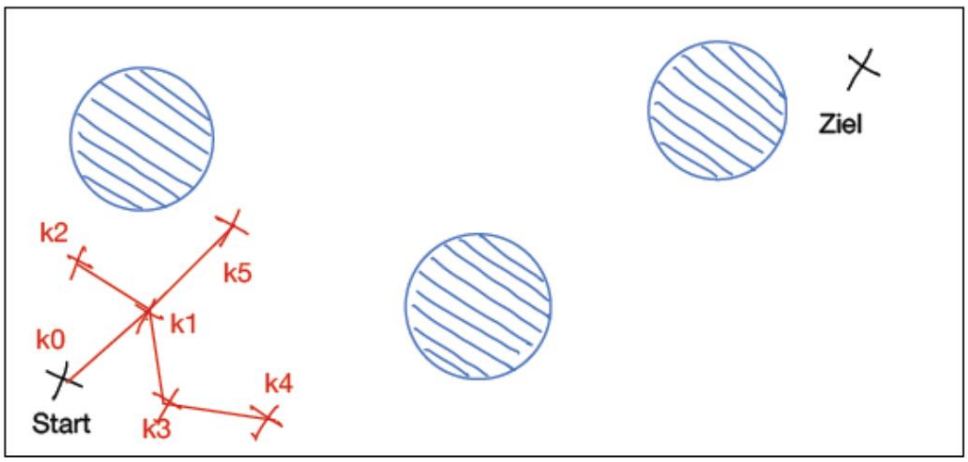
After the starting point has been determined and entered into the list, the destination point must now be defined.
To do this, the user must enter the target point until it is in the free configuration field. After each entry, the functions "inObstacle" and "inWorkspace" are used to check whether this is the case. As soon as the entered point is in the free configuration space, the target is accepted and receives the status "True". It is then determined whether a direct path between the start and the target can be mapped without crossing an obstacle. For this, the status

"Path_to_destination" is introduced, which initially has the value "False". Then the function "Collision_Path_Obstacle" is used to check whether there is an obstacle between the start and the destination. If a direct path is found, the status "Path_to_destination" is set to "True" and the path search is already completed. Otherwise the status remains unchanged. The path search is continued.

```
680  Start=(x_start,y_start)
681  print("Start: "+str(Start))
682  Start_Name='k0'
683  Punkte=[[Start_Name,Start,'None']]
684
685  #Endpunkt angeben
686  Ziel_OK=False
687  while not Ziel_OK:
688      x_Ziel=float(input("x-Koordinate des Zielpunktes: "))
689      y_Ziel=float(input("y-Koordinate des Zielpunktes: "))
690      phi_Ziel=float(input("Orientierung des Zielpunktes: "))
691      Ziel=(x_Ziel,y_Ziel)
692
693      Kollisonskontrolle=inHindernis(Ziel)
694      print("Kollision: "+str(Kollisonskontrolle))
695      AR_Kontrolle=inArbeitsraum(Ziel)
696      print("AR_Kontrolle:" +str(AR_Kontrolle))
697
698      if any (Kollisonskontrolle) or (AR_Kontrolle !=True):
699          print("ERROR: Ziel liegt nicht im Konfigurationsraum!")
700          pass
701      else:
702          Ziel_OK=True
703      print("Ziel_OK: "+str(Ziel_OK))
704
705  Weg_zum_Ziel=False
706
707  #Prüfen ob direkter Weg zu Ziel möglich
708  if any (Kollision_Pfad_Hindernis(Start,Ziel)):
709      print("Kein direkter Pfad zu Ziel möglich")
710
711  else:
712      print("Direkter Pfad zu Ziel gefunden!")
713      Weg_zum_Ziel=True
714  print("Weg zum Ziel: "+str(Weg_zum_Ziel))
```

Program code 7-4: Path planning "Defining the target point"

Now the actual path search begins. Since the program code for this is extensive, it will not be explained in detail. Instead, the RRT search is illustrated using a flow chart and the individual steps are supported by a visualization of the situation if necessary. The starting point considered for describing the process is shown in Figure 7-7. There are already a few new points in the RRT tree, but none of these points lead to the goal. The points are stored in the "Points" list and a new point is to be added.



Punkte= [[k0,Start,None], [k1,(x,y),k0], [k2, (x,y), k1],[k3,(x,y),k1], [k4, (x,y), k3], [k5,(x,y), k1]]

Figure 7-7: Initial situation of the considered RRT algorithm

The flow chart or path search begins with a status query (see Figure 7-8). This checks whether a path already leads to the destination. If the status is "True", a path already exists, otherwise the status is "False" and the path search begins. For this purpose, a new status for a point "Point_OK" is added. At the beginning, this status is always set to "False". When the status is checked, the status is "False" in the first run. Therefore, an arbitrary point is created in space and the distance between the point and every other point in the RRT tree is compared. The point in the RRT tree to which the new point is the shortest distance is considered the *parent* . The vector is then calculated between the *parent* and *child* and from this the unit vector is calculated. Since the unit vector is always one, it can be multiplied by any factor. This factor specifies the distance from the parent at which the new node or point should be added. This means that the previously created arbitrary point only serves to specify the direction for the new point or node in the RRT tree and is no longer needed afterwards. This has the advantage that all points in the RRT tree have the same distance apart from the last point and the target.

As soon as the new point has been generated, a collision and workspace check is carried out. The following two questions are answered:

1. Is the new point in an obstacle or in the occupied configuration space?
2. Is the point in the specified working space?

If the first question is answered with “True” and/or the second question with “False”,
the new point is discarded. The status "Point_Ok" remains on "False". The point search starts again from the beginning. Otherwise the point is OK and the status "Point_Ok" is set to "True". The new point is added to the list of points in the RRT tree with the information name, coordinates and *parent*
appended. It is also checked whether there is a path from the new point to the destination that does not pass over an obstacle. If such a path exists, the status “Path_to_destination” is set to “True”, otherwise it remains at “False”. Then the loop starts again and the

Status check "Path_to_destination". If this status check is positive this time, then a complete path from the start to the destination exists and the path search can be aborted. If the answer to the status query is negative, the procedure starts again from the beginning and a new arbitrary point is created.

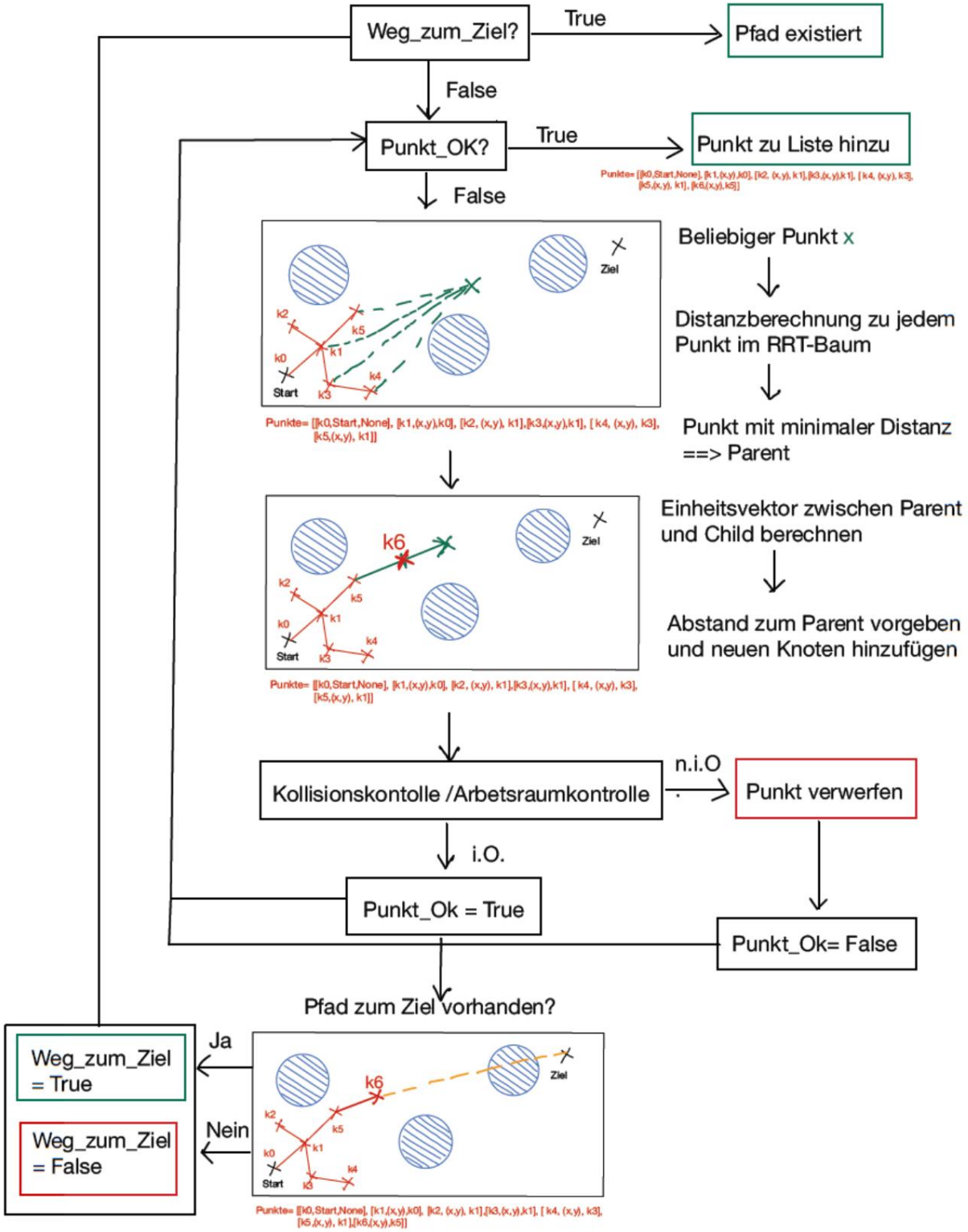


Figure 7-8: Path search flowchart

Now there is a list with many points, but it is not yet known which of the points the vehicle must drive over to reach the destination. This is determined by a backward search (see
Program code 7-5). To do this, a new list is first created for the path. The target point is assigned directly to this list. In addition, a new status "am_Start" is introduced and assigned the value "False". The target point is then specified as the current point and its predecessor point as *the parent.* A while loop is then started. This checks the status "am_Start" before each run and includes a for loop with the running variable l. The for loop is run as long as there are number of points in the "Points" list. Within the for loop, each point is checked to see whether it corresponds to the parent of the current point. For example, if the target point has the following parameters, then k8 is the name of the target point and k3 is its *parent.*

[k8, (-300,300), k3]

As soon as the for loop has reached the point with the name "k3", this is inserted into the path list at the first position and set as the current point, and its *parent* is specified as the current *parent* . The for loop then continues to run through. Only when the for loop reaches the start point is the parent while loop terminated. This is terminated by comparing the current *parent* with the value "None". When the point "Start" was introduced, the value "None" was assigned to it as *the parent* . Therefore, the status "am_Start" is set to "True". The condition of the while loop is met, so that ultimately a list "Path" is present. This list includes all points that lead from the start to the destination.

```
789  #Weg zurück vom Ziel zum Start
790  Pfad=[Punkte[-1][1]]
791  am_Start=False
792
793  aktueller_Punkt=Punkte[-1][1]
794  aktueller_Parent=Punkte[-1][2]
795
796  while not am_Start:
797      for l in range(len(Punkte)):
798          if Punkte[l][0]==aktueller_Parent:
799              Pfad.insert(0,Punkte[l][1])
800              aktueller_Punkt=Punkte[l][1]
801              aktueller_Parent=Punkte[l][2]
802          if aktueller_Parent=='None':
803              am_Start=True
804
805  Pfad.append(Ziel)
```

Program code 7-5: Path planning "Selection of points for path"

7.4 Generation of a movement along a path

In the previous chapters, a script for path search and a script for driving a point-to-point path were already developed. By combining the two scripts, a movement along the entire path via the via points should now take place. Since this is no longer a pure point-to-point movement, the script for path planning or

Trajectory planning needs to be revised.

Figure 7-9 shows that for a point-to-point movement, the initial conditions are zero speed at the start and at the destination. For a multi-point movement, this results in a braking and acceleration movement when passing through each via point, so that the speed at each point is zero. To avoid this, the trajectory planning should be revised so that acceleration only occurs at the start and braking at the destination. In between, the vehicle should travel at a constant speed.

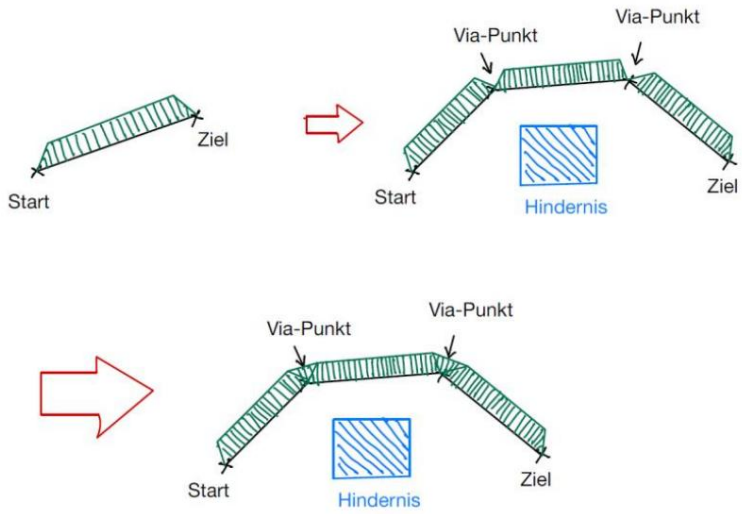


Figure 7-9: Velocity curve at via points

To do this, the path planning script needs information about which point it is at at the current time and how many points in total there are to be traveled. Depending on this, the points for a route between two points are interpolated in different ways. This results in four different speed curves (see Figure 7-10). Interpolation is to be carried out according to speed curve a if the path still only consists of a start and destination point. Interpolation is carried out according to b if more than two points are included in the path and the point in question is the start point. As soon as there are more than three points in the path and the route section is spanned between two via points, interpolation should be carried out according to speed curve c. If there are more than two points and the last route section, interpolation should be carried out according to speed curve d.

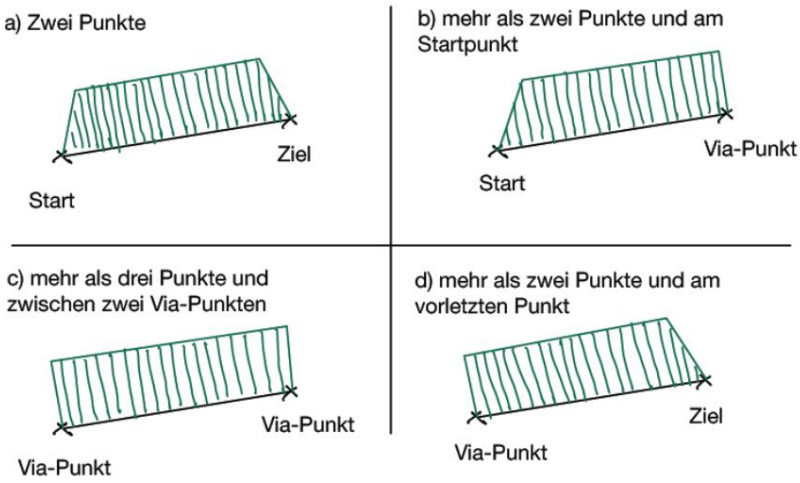


Figure 7-10: Speed profiles of the track sections

A function for interpolation a has already been introduced in the path planning script.
Accordingly, interpolation functions must be introduced for the remaining speed curves. The equations for the individual phases are determined analogously to the first interpolation.
A derivation is omitted at this point. The following equations result for the individual interpolations as well as the total time and the new speed:

Velocity curve b:

total time $= \cancel{+}\,0.5\ \ddot{y}$ — (7-5)

speed $, = \quad , \ \ddot{y}\ \ddot{y}\ \ddot{y}\ , \quad \overline{\quad 2\ \dot{y} \quad 2\ \ddot{y}\ 2\ \ddot{y} \quad , \ \dot{y} \quad , \quad}$ (7-6)

$$( ) = \{ \quad , \ \ddot{y}\ (\ddot{y} \quad \frac{1}{2}\ \dot{y} \ \frac{\quad}{,}) \qquad \begin{matrix} \frac{1}{2}\ \dot{y} \quad \dot{y} \quad 2 \\ \dot{y} \end{matrix} \qquad \begin{matrix} 0 \ \ddot{y}\ \ddot{y} \\ \ddot{y}\ \ddot{y} \\ 0 \ \ddot{y}\ \ddot{y}\ \ddot{y}\ \ddot{y} \end{matrix} \tag{7-7}$$

$$( ) = \{ \tag{7-8}$$

Speed curve c:

total time $= \frac{\quad}{}$ (7-9)

speed $, = \frac{\quad}{}$ (7-10)

$$( ) = \quad , \ \ddot{y}\ t \tag{7-11}$$

$$( ) = \quad , \tag{7-12}$$

Speed curve d:

total time = + 0.5 ÿ — (7-13)

speed = (7-14)

( ) = { , (7-15)

( ) = { (7-16)

The if query determines at which point on the path the vehicle is currently located (see
Program code 7-6). The total time, the new speeds and the acceleration time as well as the braking time are calculated again for each section of the route. This is followed by interpolation and driving along the section of the route.

```
239     if len(Pfad)>2 and p==1:
240
241         '''
242         v
243         |
244         |   _________________
245         |  /|                |
246         | / |                |
247         |/__|________________|_____t
248         p0  t_a              p1
249         '''
```

Program code 7-6: Path planning "Speed profile of a route section"

In order to be able to carry out the interpolation and travel the route, the "path planning" function must first be called. In addition, only the x-coordinates and y-coordinates of the individual via points were defined in the path planning, but not the orientation.
Therefore, in addition to the list for the points in the path, another list is created for the orientation phi (see
It is specified that each section of the route should cover an nth part of the total rotation between the start and destination orientations. The corresponding orientations at the via points are calculated using a for loop.

```
686  #Abfahren des Pfades
687  delta_phi=(phi_Ziel-phi_start)/(len(Pfad)-1)
688  phi_Pfad=[]
689  for z in range(len(Pfad)):
690      phi=phi_start+z*delta_phi
691      phi_Pfad.append(phi)
692  print("Phi entlang Pfad: "+str(phi_Pfad))
693
694  p=1
695  print("Länge Pfad: "+str(len(Pfad)))
696  v=[150]
697  omega=[3]
698
699  while p < len(Pfad):
700      [v_neu,omega_neu]=Bahnplanung(Pfad[p][0],Pfad[p][1],phi_Pfad[p],v[p-1],omega[p-1])
701      v.append(v_neu)
702      omega.append(omega_neu)
703      print("Am Punkt: "+str(p))
704      p=p+1
705
706  print("Am Ziel!!!")
```

Program code 7-7: Path planning "Traveling the path"

The individual points are traveled through using a while loop. The while loop is executed until the target point is reached. For this purpose, a running variable p and lists with the speeds or rotational speeds at each point are set up. The running variable p is given the value one. The functionality of the speed lists will be discussed later.

The control variable p is used in the while loop to determine which position and orientation should be passed to the path planning. The path planning is executed and the vehicle travels the section of the route between the current point and the passed point.
In addition, return values have now been introduced in the "Path Planning" function. The speed and rotational speed at the last interpolated point are returned. These speeds are added to the previously introduced lists and passed to the "Path Planning" function the next time the while loop is run through. This means that the initial speed for the next section of the route is known.

Figure 7-11 shows the need for this. If the vehicle is at a via point and the previous speed is not adopted, the system calculates a new speed that is completely independent of the previous speed. This can lead to a jump in the speed curve so that it is no longer continuous. The requirement of "continuous speed curve" is therefore no longer met. In addition, the vehicle must accelerate or brake unexpectedly. However, these processes are not provided for in the interpolation.

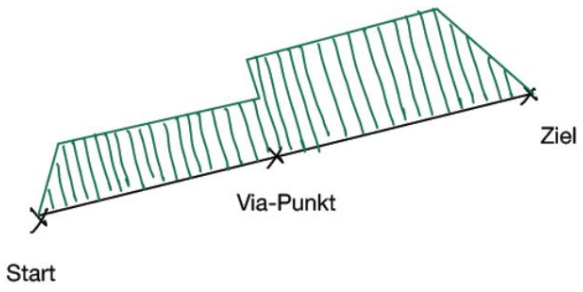


Figure 7-11: Speed difference at via point

Once the vehicle is at the target position and in the target orientation, it should continue to maintain its position. For this purpose, position control is carried out again, but this is not described in detail. The complete program code for the path search and for driving along the path can be found in Appendix G.

8 final remarks

This chapter contains a summary of the master's thesis. It addresses key content and results. In addition, the results are critically examined and discussed.
The conclusion is an outlook on possible further developments including follow-up work on the results obtained.

8.1 Summary

This paper deals with the development of a motion control system for a driverless transport system. The result of this work is program codes for the motion control. The goal is achieved and the results are created in several chapters or sections.

The first chapter describes the problem, the objective and the general workflow. The motion control of a robot has the task of receiving and processing program codes in order to generate motion commands from them. They therefore gained in importance in the course of industrialization. The present driverless transport system is divided into a driverless transport vehicle and a tracking system, which are provided by the university. A motion control system must be implemented for these.

The second chapter looks at the approach in this work. The implementation of the work is based on the agile project management method *Scrum.* In the Scrum method, a *product backlog* with requirements for motion control is described. These requirements are implemented iteratively in small *sprints .* For each *sprint,* the requirements are put into a *sprint backlog.*
summarized. There are a total of four *sprints for motion control.* Of these, three *sprints* are part of this work. In the first *sprint,* the interfaces are analyzed, the second *sprint* deals with a point-to-point movement and the last sprint includes the motion control of a multi-point movement.

The third chapter describes the initial situation of the work. The development of the motion control is carried out using an AGV from Nexus Robotics and a tracking camera. The vehicle includes Mecanum wheels and ultrasonic sensors. The control and functionality of these components are therefore supported by theoretical principles.
explained. In addition, Nexus provides various libraries and a demo script, which are also explained. In addition to the assessment of the FTF, the functionality of the tracking camera is also briefly discussed.

Based on the analyzed initial situation, a *product backlog* is created in the fourth chapter. The individual requirements are not described in detail here. Instead, they are prioritized and thus organized according to the various *sprints .* The requirements themselves are discussed in the respective chapters on the implementation of the *sprints .*

The fifth chapter covers the first *sprint.* The requirements are described in the associated *sprint backlog .* The existing interfaces of the vehicle and the tracking camera are analyzed. The tracking camera works via REST API. The vehicle integrates various options for implementing an interface, including for a Bluetooth module. The Bluetooth module HC-05 is selected for data transmission to the vehicle. The vehicle coordinates are requested and output using HTTP commands via a Python script. The motion commands are issued via the same script. To implement the motion control,

First, the hardware is implemented and the tracking system is calibrated. The program code developed afterwards takes over the motion control tasks just described.

The sixth chapter deals with the second *sprint* and its requirements. The requirements include the development of a trajectory planning for a point-to-point movement and the introduction of a position control. To do this, the theoretical principles of cascade control, general trajectory planning, trajectory planning for a point-to-point movement and the transformation of coordinate systems are first described. In addition, formulas for the interpolation and transformation are derived. These formulas are used in the program codes for the motion control. First, a program code for the position control is created.

developed. Then an Arduino sketch for receiving the movement commands is further developed and finally the program code for trajectory planning is implemented. This includes the position control and an interface for data transmission.

The seventh and final chapter examines autonomous path search. Various path planning algorithms are considered. These algorithms aim to create a suitable path between a start and end point. After comparing the algorithms, the implementation of an RRT algorithm is selected for this work and corresponding heuristics are defined.

Path planning results in a path via via points, so that a motion controller is required for a multi-point movement. To implement this movement, the existing path planning script is extended. The multi-point movement takes place in a static environment with obstacles that do not change over time. These obstacles are initially virtual and are created by the user.

specified.

8.2 Critical Review

The program codes developed in the course of this work implement motion control. This motion control meets the requirements set out in the *product backlog* or the individual *sprint backlog* with one exception. The exception here is the requirement of "constant speed progression". Some measures have already been taken to generally meet this requirement, but not all boundary conditions are taken into account.

In the course of the work, the path planning algorithm RRT is developed. This often generates a path over several via points. However, the RRT algorithm requires further optimization measures to generate a continuous movement. One of these measures is the blending of points in order to obtain a continuous transition between the points and a uniform course (see Figure 8-1 above). In addition, it can happen that the requirement is not met due to
of too high a speed when cornering is not met. The vehicle will veer off course if the centrifugal forces acting on the vehicle are too high. The centrifugal forces can be reduced by increasing the curve radius or reducing the speed (see Figure 8-1 below). All such boundary conditions or optimizations are not addressed in this work, but they do have an influence on a continuous movement.

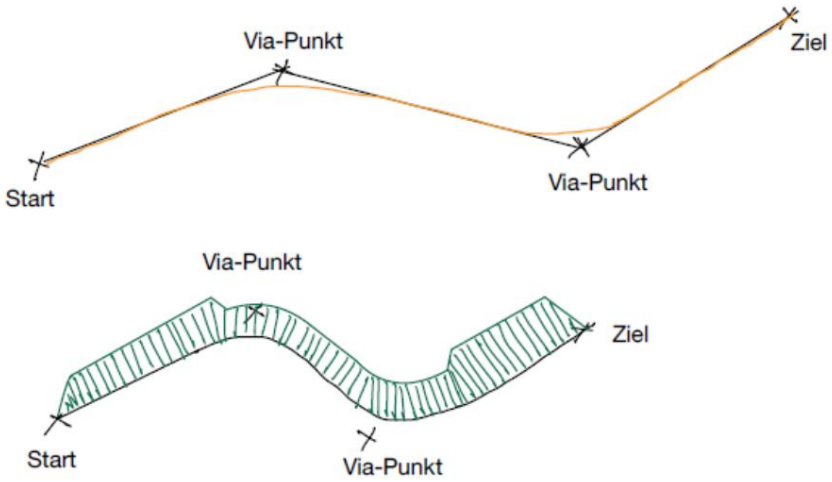


Figure 8-1: Optimization measures at via points

No specific technical requirements are specified within the *product backlog* .
Technical requirements include, for example, the accuracy of the system. Therefore, there are no precise specifications for the system developed. However, the first step towards achieving high accuracy has already been taken with the position control. However, many other factors play an important role in achieving high accuracy. However, these are not part of this work.

To verify that the requirements are met, the motion control was tested.
These tests are carried out within the available workspace. They serve to further develop and correct the program code and are aimed at checking the individual functions of the motion control. The motion control basically works without any impairment. However, the given test environment is very small in relation to the size of the vehicle and is further minimized by the use of tripods. It therefore makes sense to carry out further systematic and targeted tests in a larger environment.

During the test, it is noticeable that the vehicle tends to reflect in random places due to its coating. This causes the tracking camera to identify markers that are not there. This can temporarily confuse the system. Therefore, it is necessary to coat the vehicle with a different material to eliminate this error.

The developed program code for the motion control includes various functions such as position control or interpolation. These are defined in the script as independent functions and can therefore be called up independently of the higher-level program code. This means that they can be tested individually, but can also be integrated into higher-level program codes. The same applies to program parts that are required again and again, such as calculating a distance. Functions are also introduced for such tasks so that the same code does not have to be written over and over again, but rather the function can simply be called. This enables a structured and orderly program flow with few redundant descriptions.

Finally, it can be stated that a functioning program code for the motion control has been developed, which includes not only a point-to-point movement, but even a multi-point movement. The goal of the task "Development of a motion control for a point-to-point movement" was fully achieved. In addition, the goal of the work is to develop a motion control for a multi-point movement. For this, the program code still needs a little optimization potential. In return, the vehicle gains autonomy due to the independent path search. The structure of the program code is well structured and easy to understand.

It provides a good foundation and structure for further development and meets all criteria for movement control.

8.3 Outlook

As described in the critical review, there is still some potential for improvement with regard to the developed motion control. Among other things, the functions of the motion control are validated through tests. However, further tests are required to check the error susceptibility of the control. There is also still some potential for optimization with regard to the vehicle movement itself. This needs to be carried out in further work before the motion control can be further developed.

The *product backlog* still contains some requirements for the further development of the control system . These are suitable for one or more additional *sprints.* The requirements describe driving in dynamic environments with real obstacles and the introduction of a user interface to display and simulate the movement and to enable the user to make inputs. These requirements are listed in Table 8-1.

Table 8-1: Open requirements

Offene Anforderungen des Product Backlogs	
Nr.	**Anforderung**
1	Fahren von Kurven
2	Kollisionsfreies Fahren in einer dynamischen Umgebung
3	Dynamische Hindernisse detektieren
4	User Interface
5	Visualisierung/Simulation des Fahrweges des FTS im Konfiguartionsraum
6	Verschleifen von Punkten entlang des Fahrweges
7	Stetiger und Ruckfreier Geschwindigkeitsverlauf

These requirements include the smoothing of points and the constant and jerk-free speed curve as well as driving around curves. A jerk-free speed curve requires more than just a ramp profile. This means that the interpolation must be revised again and replaced with a sinoid profile, for example.

The use of ultrasonic sensors is relevant for driving in dynamic environments. The ultrasonic sensors themselves are given little attention in this work, as they are not required for a static environment. Therefore, their RS485 communication interface was switched off and the pins were used for the Bluetooth module interface. A suitable solution should therefore be found for using the ultrasonic sensors to detect dynamic obstacles.

9 bibliography

4D SAS. (2023). *Overview of JSON commands* . Retrieved from https://doc.4d.com/4Dv19/4D/19.6/Overview-of-JSON-commands.300-6270054.en.html on 31.08.2023

Arduino. (2023). *Libaries.* Retrieved from https://www.arduino.cc/reference/en/libraries/ on 31.08.2023

Atmel. (2015). *ATmega328P.* From https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf accessed on August 31, 2023

AZ-Delivery. (2023). *HC-05 HC-06.* Retrieved from https://www.az-delivery.de/products/hc-05-6-pin on August 31, 2023

Barr, M. (09. 01 2001). *Introduction to Pulse Width Modulation (PWM).* From https://barrgroup.com/embedded-systems/how-to/pwm-pulse-width-modulation accessed on August 31, 2023

Bartmann, E. (2012). *Discovering the electronic world with Arduino.* Cologne: O'Reilly.

Baumgarten, M. (1990). *Fundamental analysis of the Mecanum wheel.* From https://www.innorad.de/Daten/Interna/Literatur/RaederRollen/Grundsatzanalyse%20des%20Mecanum-Rades%20-%20Markus%20Baumgarten,%20Braunschweig.pdf accessed on 31.08.2023

Bies, L. (2017). *RS485 serial information.* From https://www.lammertbies.nl/comm/info/rs-485#intr accessed on August 31, 2023

Bohlin, R., & Kavraki, L. (2000). Path Planning Using Lazy PRM. *International Conference on Robotics and Automation (ICRA)* (pp. 521-528). San Francisco: IEEE.

Borke, P. (10 1988). *Points, lines, and planes.* From https://paulbourke.net/geometry/pointlineplane/ accessed on August 31, 2023

Bräunl, T. (2022). *Embedded Robotics (Fourth Edition).* Singapore: Springer Nature.

Bruhlmann, T. (2012). *Arduino practical introduction covers Arduino 1.0.* Heidelberg: Verlagsgruppe Hüthig Jehle Rehm.

Carrasco, D. (March 24, 2021). *PCINT on arduino.* From https://www.electrosoftcloud.com/en/pcint-interrupts-on-arduino/ accessed on August 31, 2023

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L., & Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementations.* London: The MIT press.

Components101. (July 16, 2021). *HC-05 - Bluetooth modules.* From https://components101.com/wireless/hc-05-bluetooth-module accessed on August 31, 2023

DroneBot Workshop. (10. 05 2022). *Using Arduino interrupt.* Retrieved from https://dronebotworkshop.com/interrupts/ on 31.08.2023

Ericson, E. (04. 04 2022). *Detecting movement direction with two ultrasonic distance sensors.* Retrieved from https://www.sensingthecity.com/detecting-movement-direction-with-two-ultrasonic-distance-sensors/ on August 31, 2023

FH Bielefeld. (2017). *REST.* From https://fh-bielefeld-mif-sw-engineerin.gitbooks.io/script/content/embedded-computing/IOT-Protokolle/Rest.html retrieved on August 31, 2023

fonial GmbH. (2023). *World Wide Web.* Retrieved from https://www.fonial.de/wissen/begriff/www/ on August 31, 2023

Gammon, N. (08 2015). *Atmega328P Timer bit-pattern charts for download.* From https://forum.arduino.cc/t/atmega328p-timer-bit-pattern-charts-for-download/328377 accessed on 31.08.2023

Generation Robotics. (2023). *Right Mecanum wheel.* From https://www.generationrobots.com/de/403135-rechtes-mecanum-rad-100-mm.html accessed on 31.08.2023

Hinzmann, T. (2011). *Path planning in indoor spaces and integration into a simulation environment for Autonomous ground vehicles.* Karlsruhe: KIT.

Hirzel, T. (03. 08 2023). *Basics of PWM (Pulse Width Modulation).* By https://docs.arduino.cc/learn/microcontrollers/analog-output accessed on 31.08.2023

Hofschulte, J. (September 17, 2022). Robotics basics. *Lecture notes.* Hannover: HSH.

Hofschulte, J., & Waldt, N. (10. 11 2021). Workshop Advanced Control Technology/Cyberphysical Systems. *Script.* Hannover: HSH.

Intellinet. (2023). *A Guide to Power over Ethernet (PoE) Switches.* By https://intellinetnetwork.de/pages/poe-switches-technologie accessed on August 31, 2023

Jänisch, R., & Donges, J. (2017). *Do something with Arduino.* Munich: Hanser.

Kanjanawanishkul, K. (02 2015). Omnidirectional wheeled mobile robots: Wheel types and practical applications. *International Journal of Advanced Mechatronic Systems,* pp. 289-302.

Kavraki, L.E., & LaValle, S. (2016). Motion planning. In B. Siciliano, & O. Khatib, *Springer Handbook of Robotics* (pp. 139-162). Heidelberg: Springer.

Kollmorgen. (2023). *Servo motors.* From https://www.kollmorgen.com/de-de/products/motors/servo/servomotoren?utm_source=google&utm_medium=cpc&utm_campaign=DE-servo-motors-nonbrand&utm_term=servomotoren&matchtype=e&utm_content=&gad=1&gclid=EAIaIQobChMI3fGMyL3AgAMVQolQBh354AIiEAAYAiAAEgL1L_D_BwE accessed on August 31, 2023

Koren, Y., & Borenstein, J. (1991). *Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation.* Sacramento: IEEE Conference on Robotics and Automation.

Kuffner James, & LaValle, S. (2000). RRT-Connect: An Efficient Approach to Single-Query Path Planning. *International Conference on Robotics and Automation (ICRA)* (pp. 1-8). San Francisco: IEEE.

Kunbus. (2023). *Baud rate basics.* From https://www.kunbus.de/baudraten-grundlagen#:~:text=Die%20Baudrate%20ist%20eine%20Einheit,Datenendeinrichtung%20und%20of%20the%20data%C3%BCtransmission%20device. accessed on August 31, 2023

Künemund, F. (2017). *Navigation system for holonomic driverless transport vehicles with multi-criteria optimization.* Düsseldorf: VDI Verlag.

Lackes, R. (February 19, 2018). *World Wide Web.* From https://wirtschaftslexikon.gabler.de/definition/world-wide-web-www-49260 accessed on August 31, 2023

Leidner, D. (2011). *Path planning for real-world tasks performed by real robot systems.* Mannheim: DLR.

Linde. (2023). *Automatically on the move.* From https://www.linde-mh.de/de/Loesungen/Intralogistik-Automation/Driverless Transport Systems/ accessed on August 31, 2023

Mareczek, J. (2020). *Fundamentals of Robot Manipulators – Volume 2.* Berlin: Springer Vieweg.

Margolis, M. (2012). *Arduino Cookbook.* Cologne: O'Reilly.

mwwalk. (11. 08 2014). *Arduino Pin Change Interrupts.* From https://thewanderingengineer.com/2014/08/11/arduino-pin-change-interrupts/ accessed on 31.08.2023

Natrual Point. (2010). *S250e Quick Start Guide.* From https://d111srqycjesc9.cloudfront.net/S250e%20Quick%20Start%20Guide.pdf accessed on August 31, 2023

NatrualPoint Inc. (2012). *Data sheet S250e.* From https://optitrack.com/support/hardware/s250e.html accessed on August 31, 2023

NatrualPoint Inc. (2023). *General FAQs.* From https://optitrack.com/support/faq/general.html accessed on August 31, 2023

Nexus Robot. (2012). *Robot Kits User's Manual.* Retrieved from https://www.nexusrobot.com/product/4wd-mecanum-wheel-mobile-arduino-robotics-car-10011.html on August 31, 2023

Odendahl, M., Finn, J., & Wenger, A. (2010). *Arduino-Physical Computing for Hobbyists.* Cologne: O'Reilly.

Parmar, N. (June 3, 2022). *API, REST API and RESTful API.* From https://wirtschaftslexikon.gabler.de/definition/world-wide-web-www-49260 accessed on 31.08.2023

Preußig, J. (2020). *Agile project management.* Freiburg: Haufe Verlag.

Probst, U. (2022). *Servo drives in automation technology (3rd edition).* Wiesbaden: Springer Vieweg.

RN-Wissen. (June 23, 2023). *Control engineering.* Retrieved from https://rn-wissen.de/wiki/index.php/Regelungstechnik on August 31, 2023

Sauter, M. (2022). *Basic Course in Mobile Communication Systems (8th edition).* Wiesbaden: Springer Vieweg.

Seobility. (2023). *REST API.* Retrieved from https://www.seobility.net/en/wiki/REST_API on August 31, 2023

Swift, N. (28. 02 2017). *Easy A* (star) Pathfinding.* Retrieved from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 on 31.08.2023

Tobola, A. (01. 09 2014). *Control engineering.* Retrieved from https://tnotes.de/ControlPage

Weis, O. (October 21, 2021). *RS485 communication guide.* Retrieved from https://www.eltima.com/de/article/rs485-communication-guide/ on August 31, 2023

Wenz, M. (2008). *Automatic configuration of the motion control of industrial robots.* Karlsruhe: TH Karlsruhe.

Winkelmann, P. (2000). *Definition of CRM.* German Direct Marketing Association in cooperation with the Landshut University of Applied Sciences.

Winkler, A. (2016). *Sensor-guided movements of stationary robots.* Chemnitz: Universitätsverlag Chemnitz.

Wirdemann, R., & Mainusch, J. (2017). *Scrum with User Stories.* Munich: Hanser Verlag.

Yadav, A. (14. 11 22). *RRT.* Retrieved from https://github.com/nimRobotics/RRT on 31.08.2023

10 Attachment

Appendix A: FTF data sheet

Appendix B: Demo script

Appendix C: Arduino script for driving movement Sprint #1

Appendix D: Arduino script for driving movement Sprint #2

Appendix E: Attitude Control Script

Appendix F: Script Path Planning for Point-to-Point Movement

Appendix G: Script Path Planning and Driving Movement along Path

Appendix A: FTF data sheet

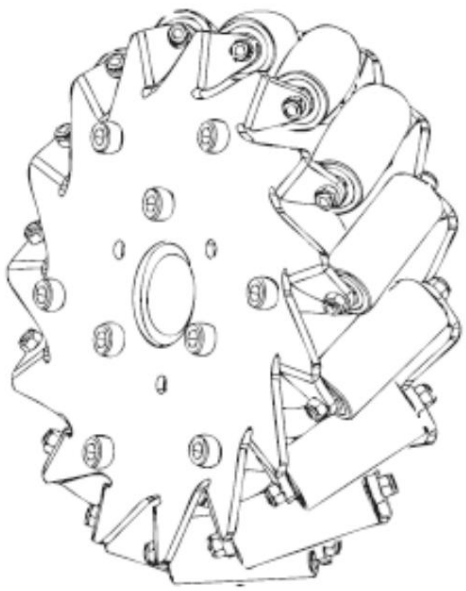
NEXUS robot

4WD 100mm Mecanum wheel robot kit 10011

This robot kit is based on a 4 wheels driving mecanum wheels chassis. the vehicle is stable and can be made to move in any direction and turn by varying the direction and speed of each wheel. Moving all four wheels in the same direction causes forward/backward movement, running left/right sides in opposite directions causes rotation, and running front and rear in opposite directions causes sideways movement. Combined motion is also possible. Its rear wheels are mounted in a particular way, this simple suspension structure ensures its four wheels can adhere to the ground even when the ground is uneven .This is very important for an mecanum wheels system. It also includes a microcontroller and motors with encoders.

Features:

- 4 wheels drive
- Mecanum wheels
- Aluminum alloy body
- Suspension ensure 4 wheel adhere to ground
- Mounts reserve for Ultrasonic and Infrared sensor
- DC motors with encoders
- Includes Microcontroller and IO expansion board
- Capable of moving forward, backward, sideway, rotation



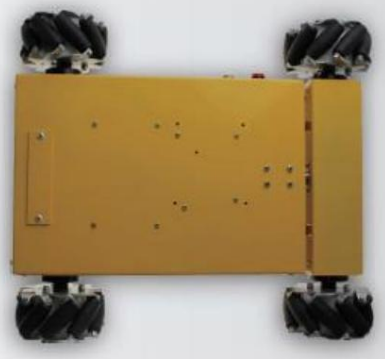
Parts included:

- 4 100mm Aluminum Mecanum Wheels
- Faulhaber 12V DC Coreless Motor
- Arduino 328 Controller
- Arduino IO Expansion V1.1
- 2 12V Ni-Mh Battery
- 12V Charger
- 4 100mm Aluminum Mecanum Wheels
- 4 Faulhaber 12V DC Coreless Motor
- Arduino 328 Controller
- Arduino IO Expansion V1.1
- 2 12V Ni-Mh Battery
- 12V Charger

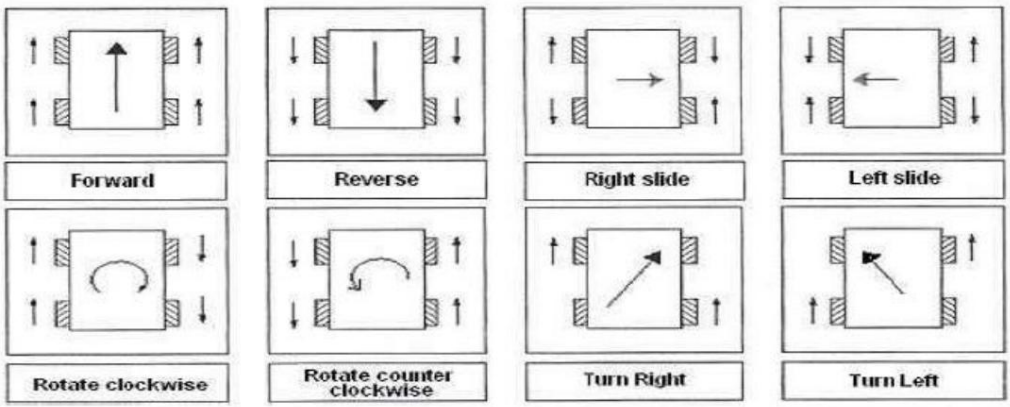
www.nexusrobot.com

NEXUS robot

4WD 100mm Mecanum wheel robot kit 10011



Co-effect of 4 Mecanum wheels:



Specifications:

Chassis	Appearance	Rectangle
	Length	400mm
	Max Width	360mm
	Height	100mm
	Chassis Height	22mm
	Wheel Base	300mm
	Wheel span	300mm
	Material	Aluminium
	Max Speed	0,6m/s
	Drive Mode	4 wheel drive
	Climbing Capacity	20 degree
	Load Capacity	20kg
	min-ITX Compatible	Yes
Wheel	Type	45 degree Mecanum Wheel
	Diameter	100mm
	Thickness	50mm
	Material	Aluminum Alloy and Rubber
	Load Capacity	15kg
	Roller Material	Rubber
	Diameter of Roller	29mm
	Length of Roller	47mm
	Coupled Mode	2 x 684ZZ bearings

Motor	Type	Faulhaber 12V DC Coreless Motor
	Power	17W
	RPM	120rpm
	Diameter	30mm
	Length	42mm
	Total Length	85mm
	Diameter of Shaft	6mm
	Length of Shaft	35mm
	No Load Current	75m
	Load Current	1400mA
	Gearbox Ratio	64:1
Encoder	Type	Optical
	Encoder Phase	AB
	Encoder Resolution	12CPR
Battery and Charger	Battery	12V Ni-Mh
	Slow Charger	100~240V In, 2,4~12V Out
	Duration of Charge	2 hours
	Running Time	0,5 hour

Microcontroller Specification	Atmega 328
	14 Channels Digital I/O
	6 PWM Channels (Pin11,Pin10,Pin9,Pin6,Pin5,Pin3)
	8 Channels 10-bit Analog I/O
	USB interface
	Auto sensing/switching power input
	ICSP header for direct program download
	Serial Interface TTL Level
	Support AREF
	Support Male and Female Pin Header
	Integrated sockets for APC220 RF Module
	Five IIC Interface Pin Sets
	Two way Motor Drive with 2A maximum current
	7 key inputs
	DC Supply:USB Powered or External 7V~12V DC
	DC Output:5V /3,3V DC and External Power Output
	Dimension:90x80mm
IO expansion board	To support RS485 interface or drive 4 motors

Appendix B: Demo script

```
#include <EEPROM.h>
#include <MotorWheel.h>
#include <Omni4WD.h>
#include <PID_Beta6.h>
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>
#include <SONAR.h>

/*
************************************************** **********************
******
                                        Sonar:0x12
                                  ------------------------------
                                  |                             |
                                M3| |                           |M2
                                                                |
                                                                |
                                  | |                           |
                        Sonar:0x13| | |                         |Sonar:0x11
                                                                |
                                                                |Power Switch
                                                                |
                                  | |                           |
                                  ------------------------------
                                  |                             |
                                M4|                            |M1
                                  |                             |
                                  ------------------------------
                                        Sonar:0x14
************************************************** **********************
******
*/
irqISR(irq1,isr1);
MotorWheel wheel1(3,2,4,5,&irq1);

irqISR(irq2,isr2);
MotorWheel wheel2(11,12,14,15,&irq2);

irqISR(irq3,isr3);
MotorWheel wheel3(9,8,16,17,&irq3);

irqISR(irq4,isr4);
MotorWheel wheel4(10,7,18,19,&irq4);

Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4); //All motors included

SONAR sonar11(0x11),sonar12(0x12),sonar13(0x13),sonar14(0x14);
```

```
unsigned short distBuf[4];

unsigned char sonarsUpdate() { static
    unsigned char sonarCurr = 1; if(sonarCurr==4)
    sonarCurr=1; else ++sonarCurr;
    if(sonarCurr==1)

    { distBuf[1]=sonar12.getDist();
        sonar12.showDat();
        sonar12.trigger(); } else
    if(sonarCurr==2)
        { distBuf[2]=sonar13.getDist();
        sonar13.showDat();
        sonar13.trigger(); } else
    if(sonarCurr==3)
        { distBuf[3]=sonar14.getDist();
        sonar14.showDat();
        sonar14.trigger(); } else

    { distBuf[0]=sonar11.getDist();
        sonar11.showDat();
        sonar11.trigger();
    }

    return sonarCurr;
}

void goAhead(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_ADVANCE) Omni.setCarSlow2Stop(300);
        Omni.setCarAdvance(0);
        Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnLeft(unsigned int speedMMPS)
    { if(Omni.getCarStat()!=Omni4WD::STAT_LEFT) Omni.setCarSlow2Stop(300);
        Omni.setCarLeft(0);
        Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnRight(unsigned int speedMMPS)
    { if(Omni.getCarStat()!=Omni4WD::STAT_RIGHT) Omni.setCarSlow2Stop(300);
        Omni.setCarRight(0);
        Omni.setCarSpeedMMPS(speedMMPS, 300);
}


void rotateRight(unsigned int speedMMPS)
    { if(Omni.getCarStat()!=Omni4WD::STAT_ROTATERIGHT)Omni.setCarSlow2Stop(300);
        Omni.setCarRotateRight(0);
        Omni.setCarSpeedMMPS(speedMMPS, 300);
```

```
}

void rotateLeft(unsigned int speedMMPS)
        { if(Omni.getCarStat()!=Omni4WD::STAT_ROTATELEFT) Omni.setCarSlow2Stop(300);
                Omni.setCarRotateLeft(0);
                Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void allStop(unsigned int speedMMPS){
        if(Omni.getCarStat()!=Omni4WD::STAT_STOP) Omni.setCarSlow2Stop(300);
                Omni.setCarStop();
}

void backOff(unsigned int speedMMPS)
    { if(Omni.getCarStat()!=Omni4WD::STAT_BACKOFF) Omni.setCarSlow2Stop(300);
                Omni.setCarBackoff(0);
                Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void(*motion[16])(unsigned int speedMMPS) = {goAhead, turnRight, goAhead, turnRight,
turnLeft,
        goAhead, turnLeft, goAhead, rotateRight,
        rotateRight, turnRight, turnRight, rotateLeft, backOff, turnLeft,
        allStop};

unsigned long currMillis=0; void
demoWithSensors(unsigned int speedMMPS,unsigned int distance) { unsigned char
        sonarcurrent = 0; if(millis()-
        currMillis>SONAR::duration + 20) { currMillis=millis();
                sonarcurrent =
                sonarsUpdate();
        }

        if(sonarcurrent == 4){
                unsigned char bitmap = (distBuf[0] < distance); //right bitmap |= (distBuf[1]
                < distance) << 1; // back bitmap |= (distBuf[2] < distance) <<
                2; // left bitmap |= (distBuf[3] < distance) << 3; //front


                (*motion[bitmap])(speedMMPS);
                Serial.println(bitmap);
        }

        Omni.PIDRegulate();
}

void setup()
        { delay(2000);
        TCCR1B=TCCR1B&0xf8|0x01;        // Pin9,Pin10 PWM 31250Hz
        TCCR2B=TCCR2B&0xf8|0x01;        // Pin3,Pin11 PWM 31250Hz
```

```
        SONAR::init (13 );
        Omni .PIDEnable (0.35 ,0.02 , 0 ,10 );
}

void loop()
        { demoWithSensors (160 ,20 );
}
```

Appendix C: Arduino script for driving movement Sprint #1

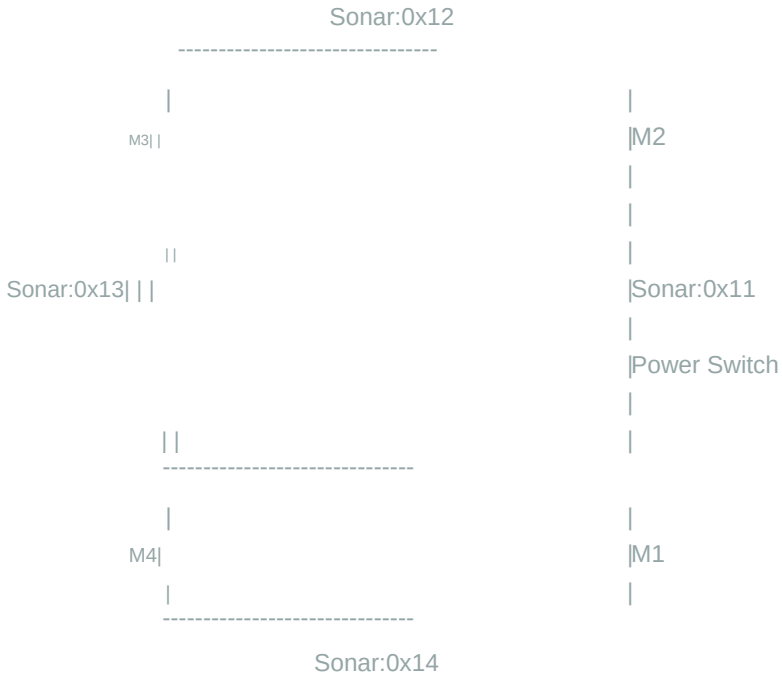
```
#include <EEPROM.h>
#include <MotorWheel.h>
#include <Omni4WD.h>
#include <PID_Beta6.h>
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>

/*
********************************************** **********************
******
                                        Sonar:0x12
                                  ------------------------------
                                  |                              |
                                M3| |                            |M2
                                  |                              |
                                  |                              |
                                  | |                            |
                        Sonar:0x13| | |                          |Sonar:0x11
                                                                 |
                                                                 |Power Switch
                                                                 |
                                  | |                            |
                                  ------------------------------
                                  |                              |
                                M4|                             |M1
                                  |                              |
                                  ------------------------------
                                        Sonar:0x14
********************************************** **********************
******
*/
irqISR(irq1,isr1);
MotorWheel wheel1(3,2,4,5,&irq1);

irqISR(irq2,isr2);
MotorWheel wheel2(11,12,14,15,&irq2);

irqISR(irq3,isr3);
MotorWheel wheel3(9,8,16,17,&irq3);

irqISR(irq4,isr4);
MotorWheel wheel4(10,7,18,19,&irq4);

Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4);

void goAhead(unsigned int speedMMPS){
    if(Omni.getCarStat()!=Omni4WD::STAT_ADVANCE) Omni.setCarSlow2Stop(300);
```

```
            Omni.setCarAdvance(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnLeft(unsigned int speedMMPS)
      { if(Omni.getCarStat()!=Omni4WD::STAT_LEFT) Omni.setCarSlow2Stop(300);
            Omni.setCarLeft(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void turnRight(unsigned int speedMMPS)
      { if(Omni.getCarStat()!=Omni4WD::STAT_RIGHT) Omni.setCarSlow2Stop(300);
            Omni.setCarRight(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void rotateRight(unsigned int speedMMPS)
      { if(Omni.getCarStat()!=Omni4WD::STAT_ROTATERIGHT)
Omni.setCarSlow2Stop(300);
            Omni.setCarRotateRight(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void rotateLeft(unsigned int speedMMPS)
      { if(Omni.getCarStat()!=Omni4WD::STAT_ROTATELEFT) Omni.setCarSlow2Stop(300);
            Omni.setCarRotateLeft(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void allStop(unsigned int speedMMPS)
      { if(Omni.getCarStat()!=Omni4WD::STAT_STOP) Omni.setCarSlow2Stop(300);
            Omni.setCarStop();
}

void backOff(unsigned int speedMMPS)
   { if(Omni.getCarStat()!=Omni4WD::STAT_BACKOFF) Omni.setCarSlow2Stop(300);
            Omni.setCarBackoff(0);
            Omni.setCarSpeedMMPS(speedMMPS, 300);
}

void(*motion[7])(unsigned int speedMMPS) =
{goAhead,turnLeft,turnRight,backOff,rotateLeft,rotateRight,allStop};

void driveCar(unsigned speedMMPS)
   { unsigned int direction;
   while(Serial.available()==0){ }


   direction=Serial.readStringUntil('\n').toInt(); (*motion[direction])
   (speedMMPS);
```

```
    Omni.PIDRegulate();

}

void setup()
{ Serial.begin(19200);

delay(2000);
TCCR1B=TCCR1B&0xf8|0x01;        // Pin9,Pin10 PWM 31250Hz
TCCR2B=TCCR2B&0xf8|0x01;        // Pin3,Pin11 PWM 31250Hz

Omni.PIDEnable(0.35,0.02,0,10); }


void loop() {

driveCar(100); }
```

Appendix D: Arduino script for driving movement Sprint#2

```
#include <EEPROM.h>
#include <MotorWheel.h>
#include <Omni4WD.h>
#include <PID_Beta6.h>
#include <PinChangeInt.h>
#include <PinChangeIntConfig.h>


/*
************************************************** **********************
******
```




```
************************************************** **********************
******
*/
unsigned int speed;
unsigned int distance;
float direction;
float rotation;
int time;
String cmd;
int i,j,k;

irqISR(irq1,isr1);
MotorWheel wheel1(3,2,4,5,&irq1);

irqISR(irq2,isr2);
MotorWheel wheel2(11,12,14,15,&irq2);

irqISR(irq3,isr3);
MotorWheel wheel3(9,8,16,17,&irq3);
```

```
irqISR(irq4,isr4);
MotorWheel wheel4(10,7,18,19,&irq4);

Omni4WD Omni(&wheel1,&wheel2,&wheel3,&wheel4);

int count=0;
ISR(TIMER0_COMPA_vect)
    { count++;
    if (count==100) {
        digitalWrite(13,!digitalRead(13)); count=0;

    }
    Omni.PIDRegulate();
}

void setup()
{ pinMode(13,OUTPUT);
Serial.begin(57600);
delay(2000);
TCCR1B=TCCR1B&0xf8|0x01;            // Pin9,Pin10 PWM 31250Hz
TCCR2B=TCCR2B&0xf8|0x01;            // Pin3,Pin11 PWM 31250Hz
OCR0A=100;
TIMSK0|=2;
Omni.PIDEnable(0.35,0.02,0,10); }


void loop()
{ while(Serial.available()==0){
    }
cmd=Serial.readStringUntil('\n');

i=cmd.indexOf(':');
speed=cmd.substring(0,i).toInt();
j=cmd.indexOf(':',i+1);
direction=cmd.substring(i+1,j).toFloat(); k=cmd.indexOf('\n');

rotation=cmd.substring(j+1,k).toFloat();

Omni.setCarMove(speed,direction,rotation);

}
```

Appendix E: Attitude Control Script

```
import requests import
serial import time
import math
import json



def position control(x_soll,y_soll,phi_soll): kp=1.2 k_phi=1



    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_FTS
    = float(obj["x"]) y_FTS = float(obj
    ["y"]) phi_FTS = float(obj["phi"])



    alpha=phi_FTS-math.pi/2


    #Target related to FTS dx=-
    math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS) dy=-math.sin(alpha)*(x_soll-x_FTS)
    +math.cos(alpha)*(y_soll-y_FTS)


    #Calculation of the speed v_x=kp*dx v_y=kp*dy



    v_ges=math.sqrt(v_x**2+v_y**2)

    if (v_ges>255):
        v_ges=255
    else:
        v_ges

    #Calculation of the direction of travel rad =
    math.atan2(dy,dx)

    #Calculation of the angle change and angular velocity omega d_phi=(phi_soll-phi_FTS)



    if (abs(d_phi)>math.pi):
        if (d_phi>0):
            d_phi=-2*math.pi+d_phi
        else:
            d_phi=2*math.pi+d_phi
    else:
        d_phi=d_phi
```

```
        omega=d_phi*k_phi #[1/s]

        #Transfer of data via serial interface
        cmd="{:.0f}".format(v_ges)+"{:.5f}".format(rad)+"{:.5f}".format(omega) cmd=cmd+'\n'

        print(str(cmd))
        arduinoData.write(cmd.encode())


arduinoData=serial.Serial('COM5',56700)
while True:
        Position control(-200,0,0)
        time.sleep(0.5)
```

Appendix F: Script Path Planning for Point-to-Point Movement

```
import requests import
serial import time
import math
import json
import
matplotlib.pyplot as plt


#============================================== ====================
#================================Attitude control=========================
#============================================================================

def position control(x_soll,y_soll,phi_soll): kp=1.2 k_phi=0.7



        print("x_soll: "+str(x_soll)) print("y_soll:
        "+str(y_soll)) print("phi_soll: "+str(phi_soll))


        r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_FTS
        = float(obj["x"]) y_FTS = float(obj
        ["y"]) phi_FTS = float(obj["phi"])



        print("Coordinates FTS:                    "
                + str(x_FTS), str(y_FTS), str(phi_FTS))

        alpha=phi_FTS-math.pi/2 #*180/math.pi print("Alpha:
        "+str(alpha))

        #Target related to FTS dx=-
        math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS) dy=-math.sin(alpha)*(x_soll-x_FTS)
        +math.cos(alpha)*(y_soll-y_FTS)

        dist = math.sqrt(dx**2 + dy**2) print("Distance:
        "+str(dist))

        #Calculation of the speed v_x=kp*dx v_y=kp*dy
```

```
        #Target related to FTS
        d_phi=(phi_soll-phi_FTS)

        if (abs(d_phi)>math.pi):
                if (d_phi>0):
                        d_phi=-2*math.pi+d_phi
                else:
                        d_phi=2*math.pi+d_phi


        print("Rotation angle: "+str(d_phi)) d_omega=k_phi*d_phi #[1/
        s]

        print("Command angle: "+str(d_omega)) print(" ")



        return v_x,v_y,d_omega

#============================================== ====================
#======================Function for path planning====================
#============================================================================

def interpolation(t,t_a,t_b,a,v):

        if v<0:
                a=-a
        else:
                a


        if(t <=t_a):
                x=0.5*a*(t)**2 v_neu=a*t


        if(t_a<t<=t_b):
                x=v*t-0.5*v**2/a
                v_neu=v

        if(t>t_b):
                x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2 v_new=va*(t-t_b)


        return [x,v_neu]
```

```
#================================================ ====================
#====================Path planning=========================================
#============================================================================
def path planning(x_target, y_target, phi_target):
        r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_start =
        float(obj["x"]) y_start= float(obj ["y"])
        phi_start= float(obj["phi"])



        print("Coordinates FTS: str(phi_start))       "      + str(x_start), str(y_start),


        s_x_total=x_target-x_start s_y_total=y_target-
        y_start
        s_total=math.sqrt(s_x_total**2+s_y_total**2) phi_total=phi_target-phi_start


        if (abs(phi_total)>math.pi): if (phi_total>0):
                phi_total=-2*math.pi+phi_total

                else:
                        phi_total=2*math.pi+phi_total
        else:
                phi_gesamt

        print("Angle_new: "+str(phi_total))


        '''
        v


        "       _____________________
        |      /|                                  |\
        | / | | \ |/__|_____________________|__\____t
        t_b
                t_a
        '''


        v_trans=200 #mm/s
        a_trans=a_x=a_y=100 #mm/s^2 teta=2.5
        #rad/s^2 omega=3 #rad/s


        #t_ges_trans=s_total/v_trans+v_trans/a_trans t_ges_x=abs(s_x_total)/
        v_trans+v_trans/a_trans t_ges_y=abs(s_y_total)/v_trans+v_trans/a_trans
        t_ges_phi=abs(phi_total)/omega +omega/teta
```

```
#for synchronous movement all positions must be reached simultaneously
become
#determine which component takes the longest and use it as the new
Set time
t_ges=max(t_ges_phi,t_ges_x,t_ges_y)

print("x time: "+str(t_ges_x)) print("y time:
"+str(t_ges_y)) print("phi time: "+str(t_ges_phi))
print("New time: "+str(t_ges ))


#Calculate the new speeds based on the new time v_x_neu=((a_x*t_ges)/2-math.sqrt((a_x**2*t_ges**2)/
4- abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt


v_y_new=((a_y*t_total)/2-math.sqrt((a_y**2*t_tot**2/4)-
abs(s_y_total)*a_y))*abs(s_y_total)/s_y_total

omega_new=((teta*t_total)/2-math.sqrt((teta**2*t_total**2/4)- abs(phi_total)*teta))*abs(phi_total)/
phi_total

#Acceleration times
t_ax=abs(v_x_neu)/a_x
t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta


#Time from which braking should take place t_bx=t_ges-
t_ax t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi


print("Time to slow down in x: "+str(t_bx)) print("Time to slow down
in y: "+str(t_by)) print("Time to slow down in phi: "+str(t_b_phi))


#empty lists in which the coordinates of the interpolated points and the target speeds are written
x_soll=[] vx_soll=[]



y_soll=[]
vy_soll=[]


phi_soll=[]
omega_soll=[]

t_=[]
```

```
delta_t=0.1 #Interval time t=0 #Start time


n=int(round((t_tot/delta_t),0)) delta_t=t_tot/n


print("delta t: "+str(delta_t))

z=n+1

for w in range(z): [x_t,
        vx_soll_neu]=Interpolation(t,t_ax,t_bx,a_x,v_x_neu) x_soll_neu=x_t+x_start x_soll.append(x_soll_neu)
        vx_soll.append(vx_soll_neu)



        [y_t, vy_soll_neu]=Interpolation(t,t_ay,t_by,a_y,v_y_neu) y_soll_neu=y_t+y_start
        y_soll.append(y_soll_neu)
        vy_soll.append(vy_soll_neu)


        [phi_t, omega_soll_neu]= interpolation(t,t_a_phi,t_b_phi,teta,omega_neu) phi_soll_neu=phi_t+phi_start
        phi_soll.append(phi_soll_neu)
        omega_soll.append(omega_soll_neu)


        t_.append(t)
        t+=delta_t

print("x_soll: "+str (x_soll)) print ("vx_soll:
"+str(vx_soll)) print ("vy_soll: "+str(vy_soll))
print("y_soll: "+str(y_soll)) print("phi_soll: "+str(phi_soll))


plt.plot(t_,x_soll, color='red') plt.plot(t_,y_soll,
color='green') plt.plot(t_,phi_soll, color='blue') plt.show()


plt.plot(t_,vx_soll, color='red') plt.plot(t_,vy_soll,
color='green') plt.plot(t_,omega_soll, color='blue') plt.show()


plt.plot(x_soll, y_soll) plt.show()

k=0
while True:
```

```
        if (k<(n)):

            d_v=position control(x_soll[k],y_soll[k],phi_soll[k]) print("Values from position control: "
            +str(d_v)) d_v_x=d_v[0] d_v_y=d_v[1] d_omega=d_v[2 ]




            v_x_korr=vx_soll[k]+d_v_x v_y_korr=vy_soll[k]
            +d_v_y omega_korr=omega_soll[k]+d_omega


            v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)

            if (v_ges_korr>255): v_ges_korr=255

            else:

                v_ges_korr


            rad = math.atan2(v_y_korr,v_x_korr)
            k=k+1
        else:
            d_v=position control(x_target,y_target,phi_target) print("Values from
            position control: " +str(d_v)) d_v_x=d_v[0] d_v_y=d_v[1] omega_korr=d_v[2]
            rad =
            math.atan2(d_v_y,
            d_v_x)
            v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)


        print("Speed: "+str(v_ges_korr)) print("Direction of travel: "+str(rad))
        print("Rotational speed: "+str(omega_korr))


        #Transfer of data via serial interface
        cmd="{:.0f}".format(v_ges_korr)+":{:.5f}".format(rad)+":{:.5f}".format(omega_korr) cmd=cmd+'\n'

        print(str(cmd))
        arduinoData.write(cmd.encode())

        time.sleep(delta_t)

#============================================== ====================
#=======================Calling program==========================
#======================================================================
arduinoData=serial.Serial('COM5',57600)
path planning (-200,300,2)
```

Appendix G: Script Path Planning and Driving Movement along Path

```
import requests import
serial import time
import math
import json
import numpy as
np


#================================================ ====================
#================================Attitude control=========================
#============================================================================

def position control(x_soll,y_soll,phi_soll): kp=1.2 k_phi=0.7



        print("x_soll: "+str(x_soll)) print("y_soll:
        "+str(y_soll)) print("phi_soll: "+str(phi_soll))


        r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_FTS
        = float(obj["x"]) y_FTS = float(obj
        ["y"]) phi_FTS = float(obj["phi"])
        print("Coordinates FTS: +
        str(x_FTS), str(y_FTS), str(phi_FTS)) "


        alpha=phi_FTS-math.pi/2
        print("Alpha: "+str(alpha))

        #Target related to FTS dx=-
        math.cos(alpha)*(x_soll-x_FTS)-math.sin(alpha)*(y_soll-y_FTS) dy=-math.sin(alpha)*(x_soll-x_FTS)
        +math.cos(alpha)*(y_soll-y_FTS)

        dist = math.sqrt(dx**2 + dy**2) print("Distance:
        "+str(dist))

        #Calculation of the speed v_x=kp*dx v_y=kp*dy



        d_phi=(phi_soll-phi_FTS)

        if (abs(d_phi)>math.pi):
                if (d_phi>0):
                        d_phi=-2*math.pi+d_phi
                else:
                        d_phi=2*math.pi+d_phi
```

```
        print("Rotation angle: "+str(d_phi)) d_omega=d_phi*k_phi #[1/
        s]

        print("Command angle: "+str(d_omega)) print(" ")



        return v_x,v_y,d_omega

#================================================ ====================
#======================Function for path planning====================
#================================================ ====================

def Interpolation1(t,t_a,t_b,a,v):
        if v<0:
                a=-a
        else:
                a

        if(t <=t_a):
                x=0.5*a*(t)**2
                v_neu=a*t

        if(t_a<t<=t_b):
                x=v*t-0.5*v**2/a
                v_neu=v

        if(t>t_b):
                x=0.5*a*(v/a)**2+v*(t_b-t_a)+v*(t-t_b)-0.5*a*(t-t_b)**2 v_new=va*(t-t_b)


        return [x,v_neu]

def Interpolation2(t,t_a,a,v): if v<0:

                a=-a
        else:
                a

        if(t <=t_a):
                x=0.5*a*(t)*(t)
                v_neu=a*t

        if(t_a<t):
                x=v*t-0.5*v**2/a
                v_neu=v

        return [x,v_neu]
```

```
def Interpolation3(t,t_b,a,v): if v<0:
        a=-a
    else:
        a

    if(t<=t_b): x=v*t

        v_neu=v

    if(t>t_b):
        x=t_b*v+v*(t-t_b)-0.5*(t-t_b)**2*a v_neu=va*(t-t_b)


    return [x,v_neu]

def Interpolation4(t,v):
    x=v*t
    v_neu=v

    return [x,v_neu]
#============================================== ====================
#=======================Railway planning===================================
#============================================== ====================
def path planning(x_target, y_target, phi_target,v_old,omega_old):
    r = requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_start =
    float(obj["x"]) y_start= float(obj ["y"])
    phi_start= float(obj["phi"])



    print("Coordinates FTS:                    "  + str(x_start), str(y_start), str(phi_start))

    s_x_total=x_target-x_start
    s_y_total=y_target-y_start
    s_total=math.sqrt(s_x_total**2+s_y_total**2) phi_total=phi_target-phi_start


    print("Distance in x-direction: "+str(s_x_gesamt)) print("Distance in y-direction:
    "+str(s_y_gesamt)) print("Distance in translational direction: "+str(s_gesamt))
    print("Angle: "+str(phi_gesamt))



    if (abs(phi_total)>math.pi): if (phi_total>0):
        phi_total=-2*math.pi+phi_total

        else:
            phi_total=2*math.pi+phi_total
```

```
print("Angle_new: "+str(phi_total))

v_trans=v_alt #mm/s
a_trans=a_x=a_y=150 #mm/s^2
teta=2.5 #rad/s^2
omega=omega_alt #rad/s

#empty lists in which the coordinates of the interpolated points and the target speeds are written

x_soll=[]
vx_soll=[]

y_soll=[]
vy_soll=[]

phi_soll=[]
omega_soll=[]

delta_t=0.1 #Interval time t=0 #Start time


if len(path)==2:
    print("In first if loop")

    '''
    v
    |
        ____________________
    | | /| | /                    |\
    | |/__|                       | \
    ___________________|__\____t
        t_a                    t_b
    '''


    #t_ges_trans=s_total/v_trans+v_trans/a_trans
    t_ges_x=abs(s_x_total)/v_trans+v_trans/a_trans
    t_ges_y=abs(s_y_total)/v_trans+v_trans/a_trans
    t_total_phi=abs(phi_total)/omega +omega/teta

    #for synchronous movement all positions must be reached simultaneously

    #determine which component takes the longest and set this as the new time

    t_tot=max(t_tot_phi,t_tot_x,t_tot_y)

    print("New time: "+str(t_ges))
```

```
#Calculate the new speeds based on the new time v_x_neu=((a_x*t_ges)/2-math.sqrt(((a_x**2)*(t_ges**2)/4)-
abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt


v_y_new=((a_y*t_total)/2-math.sqrt(((a_y**2)*(t_tot**2)/4)- abs(s_y_total)*a_y))*abs(s_y_total)/
s_y_total

omega_new=((teta*t_total)/2-math.sqrt(((teta**2)*(t_tot**2)/4)- abs(phi_total)*teta))*abs(phi_total)/phi_total



#Acceleration times t_ax=abs(v_x_neu)/
a_x t_ay=abs(v_y_neu)/a_y
t_a_phi=abs(omega_neu)/teta


#Time from which braking should take place t_bx=t_ges-t_ax
t_by=t_ges-t_ay
t_b_phi=t_ges-t_a_phi


n=int(round((t_tot/delta_t),0)) delta_t=t_tot/n


z=n+1

for w in range(z):

        [x_t, vx_soll_neu]=Interpolation1(t,t_ax,t_bx,a_x,v_x_neu) x_soll_neu=x_t+x_start
       x_soll.append(x_soll_neu)
       vx_soll.append(vx_soll_neu)


        [y_t, vy_soll_neu]=Interpolation1(t,t_ay,t_by,a_y,v_y_neu) y_soll_neu=y_t+y_start
       y_soll.append(y_soll_neu)
       vy_soll.append(vy_soll_neu)


     [phi_t, omega_soll_neu]=Interpolation1(t,t_a_phi,t_b_phi,teta,omega_neu) phi_soll_neu=phi_t+phi_start
       phi_soll.append(phi_soll_neu)
       omega_soll.append(omega_soll_neu)


       t+=delta_t
```

```
if len(path)>2 and p==1: print("In second
        loop")
         '''
        v
         |
         |       ____________________
               /|
         | | / | |/
         __|____________________|______t p0 t_a p1

         '''


        t_ges_trans=s_total/v_trans+0.5*v_trans/a_trans t_ges_phi=abs(phi_total)/omega
        +0.5*omega/teta

        #for synchronous movement all positions must be reached simultaneously

        #determine which component takes the longest and set this as the new time t_ges=max(t_ges_phi,t_ges_trans)



        print("New time: "+str(t_ges))

        #Calculate the new speeds based on the new time v_x_neu=((a_x*t_ges)-math.sqrt(((a_x**2)*(t_ges**2))-
        2*abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/s_x_gesamt


        v_y_new=((a_y*t_total)-math.sqrt(((a_y**2)*(t_tot**2))- 2*abs(s_y_total)*a_y))*abs(s_y_total)/
        s_y_total

        omega_new=((teta*t_total)-math.sqrt(((teta**2)*(t_tot**2))- 2*abs(phi_total)*teta))*abs(phi_total)/
        phi_total

        #Acceleration times t_ax=abs(v_x_neu)/
        a_x t_ay=abs(v_y_neu)/a_y
        t_a_phi=abs(omega_neu)/teta


        n=int(round((t_tot/delta_t),0)) delta_t=t_tot/n


        z=n+1

        for w in range(z): [x_t,
                 vx_soll_neu]=Interpolation2(t,t_ax,a_x,v_x_neu) x_soll_neu=x_t+x_start
                x_soll.append(x_soll_neu)
                vx_soll.append(vx_soll_neu)
```

```
                [y_t, vy_soll_neu]=Interpolation2(t,t_ay,a_y,v_y_neu) y_soll_neu=y_t+y_start
                y_soll.append(y_soll_neu)
                vy_soll.append(vy_soll_neu)


                [phi_t, omega_soll_neu]=Interpolation2(t,t_a_phi,teta,omega_neu) phi_soll_neu=phi_t+phi_start
                phi_soll.append(phi_soll_neu)#/math.pi*180)
                omega_soll.append(omega_soll_neu)


                t+=delta_t

    if len(path)>2 and p==(len(path)-1): print("In third if loop")


            '''
            v
            |
            |_____ /\ | | | \ |

            _____|__\____t p0
            t_b p1
            '''


            t_ges_trans=s_total/v_trans+0.5*v_trans/a_trans t_ges_phi=abs(phi_total)/
            omega +0.5*omega/teta

            #for synchronous movement all positions must be reached simultaneously

            #determine which component takes the longest and set this as the new time
            t_ges=max(t_ges_phi,t_ges_trans)


            print("New time: "+str(t_ges))

            #Calculate the new speeds based on the new time v_x_neu=((a_x*t_ges)-
            math.sqrt(((a_x**2)*(t_ges**2))- 2*abs(s_x_gesamt)*a_x))*abs(s_x_gesamt)/
            s_x_gesamt

            v_y_new=((a_y*t_total)-math.sqrt(((a_y**2)*(t_tot**2))-
            2*abs(s_y_total)*a_y))*abs(s_y_total)/s_y_total

            omega_new=((teta*t_total)-math.sqrt(((teta**2)*(t_tot**2))- 2*abs(phi_total)*teta))*abs(phi_total)/
            phi_total

            #Acceleration times
            t_ax=abs(v_x_neu)/a_x
            t_ay=abs(v_y_neu)/a_y
            t_a_phi=abs(omega_neu)/teta
```

```
        #Time from which braking should take place t_bx=t_ges-
        t_ax t_by=t_ges-t_ay
        t_b_phi=t_ges-t_a_phi


        n=int(round((t_tot/delta_t),0)) delta_t=t_tot/n


        z=n+1

        for w in range(z): [x_t,
                vx_soll_neu]=Interpolation3(t,t_bx,a_x,v_x_neu) x_soll_neu=x_t+x_start
               x_soll.append(x_soll_neu)
               vx_soll.append(vx_soll_neu)


                [y_t, vy_soll_neu]=Interpolation3(t,t_by,a_y,v_y_neu) y_soll_neu=y_t+y_start
               y_soll.append(y_soll_neu)
               vy_soll.append(vy_soll_neu)


                [phi_t, omega_soll_neu]=Interpolation3(t,t_b_phi,teta,omega_neu)
               phi_soll_neu=phi_t+phi_start
               phi_soll.append(phi_soll_neu)
               omega_soll.append(omega_soll_neu)

               t+=delta_t


if len(Path)>2 and p<(len(Path)-1) and p>1:
        print("In fourth if loop")

        '''
        v
        |
        |_____ | | | |


        |_____|_____t p0 p1

        '''


        t_ges_trans=s_total/v_trans
        t_ges_phi=abs(phi_total)/omega

        #for synchronous movement all positions must be reached simultaneously

        #determine which component takes the longest and set this as the new time
        t_ges=max(t_ges_phi,t_ges_trans)
```

```
        print("New time: "+str(t_ges))

        #Calculate the new speeds related to the new time v_x_neu=s_x_gesamt/t_ges v_y_neu=s_y_gesamt/
        t_ges omega_neu=phi_gesamt/t_ges



        n=int(round((t_tot/delta_t),0)) delta_t=t_tot/n


        z=n+1

        for w in range(z): [x_t,
                vx_soll_neu]=Interpolation4(t,v_x_neu) x_soll_neu=x_t+x_start
                x_soll.append(x_soll_neu)
                vx_soll.append(vx_soll_neu)


                [y_t, vy_soll_neu]=Interpolation4(t,v_y_neu) y_soll_neu=y_t+y_start
                y_soll.append(y_soll_neu)
                vy_soll.append(vy_soll_neu)


                [phi_t, omega_soll_neu]=Interpolation4(t,omega_neu) phi_soll_neu=phi_t+phi_start
                phi_soll.append(phi_soll_neu)
                omega_soll.append(omega_soll_neu)


                t+=delta_t

for k in range(n):
        d_v=position control(x_soll[k],y_soll[k],phi_soll[k]) print("Values from position
        control: " +str(d_v)) d_v_x=d_v[0] d_v_y=d_v [1] d_omega=d_v[2]



        v_x_korr=vx_soll[k]+d_v_x
        v_y_korr=vy_soll[k]+d_v_y
        omega_korr=omega_soll[k]+d_omega

        v_ges_korr=math.sqrt(v_x_korr**2+v_y_korr**2)

        if (v_ges_korr>255):
                v_ges_korr=255
        else:
                v_ges_korr

        rad = math.atan2(v_y_korr,v_x_korr)
```

```
            k=k+1

            print("Speed: "+str(v_ges_korr)) print("Direction of travel: "+str(rad))
            print("Rotational speed: "+str(omega_korr))


            #Transfer of data via serial interface
            cmd="{:.0f}".format(v_ges_korr)+":{:.5f}".format(rad)+":{:.5f}".format(omega_korr) cmd=cmd+'\n' print (str(cmd))

            arduinoData.write(cmd.encode())


            time.sleep(delta_t)

        return [v_ges_korr,omega_korr]

#================================================ ====================
#======================Path planning==================================
#=====================================================================
arduinoData=serial.Serial('COM5',57600)

######Functions for path planning algorithm#############

def calculate_distance(p1,p2):
        dist=math.sqrt((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)

        return dist

def inHindernis(p_check): in_Hindernis=[]
        for a in range (H):

            distance=calculate_distance(obstacles[a][0],p_check)

            if distance <= obstacles[a][1]:
                in_Obstacle.append(True)
            else:
                in_Hindernis.append(False) return
        in_Hindernis

def inWorkspace(p1):
        print ("function AR")
        AR_OK=True
        print("p_x: "+ str(p1[0])) print("p_y: "+
        str(p1[1])) if AR_x[1] <p1[0] or p1[0] <
        AR_x [0]:
            AR_OK=False if
        AR_y[1] <p1[1] or p1[1] < AR_y[0]:
            AR_OK=False

        return AR_OK
```

```
def point_to_line(p1,p2,p3):
    Vector=(p2[0]-p1[0],p2[1]-p1[1])
    VectorSize=math.sqrt(Vector[0]**2+Vector[1]**2)

    u=((p3[0]-p1[0])*(p2[0]-p1[0])+(p3[1]-p1[1])*(p2[1]- p1[1]) )/(VectorSize**2)


    #Coordinates of the point where the vector from p1 to p2 intersects the orthogonal to p3 px=p1[0]+u*(p2[0]-
    p1[0]) py=p1[1]+u*(p2[1]-p1[1])



    distance=calculate_distance((px,py),p3) print("u,distance:
    "+str(u)+", "+str(distance))

    return u, distance

def collision_path_obstacle(p1,p2):
    LineCollision=[] for s in range(H):
    too_close=False in between=False



        u, dist =Point_to_Line(p1,p2,Obstacles[s][0])

        if 0<=u <=1:
            in between=True

        if dist <= obstacles[s][1]:
            zu_nah=True

        if too_close and in between:
            LineCollision.append(True)
        else:
            LineCollision.append(False)

    return LineCollision
```

```
########Path planning############

AR_x=(-450,450)
AR_y=(-320,320)

#Create obstacles
H=int(input('Please enter number of obstacles: '))
obstacles=[]
R_FTS=550/2 for
i in range (H):
        x_hindernis=float(input("x_coordinate of the obstacle " +str(i+1)+":"))
        y_hindernis=float(input('y_coordinate of the obstacle '+str(i+1)+': '))
        MP_Obstacle=(x_Obstacle,y_Obstacle)
        R_Hindernis=float(input("Radius of the obstacle "+str(i+1)+": "))+R_FTS
        ObstaclePara=[MP_Hindernis,R_Hindernis]
        Obstacles.append(ObstaclePara) print("Obstacle:
        "+str(Obstacles))

#Read starting point r =
requests.get("http://laptop-n1q2j4ee:1201/data?body=53") obj = json.loads(r.text) x_start =
float(obj["x"]) y_start= float(obj["y"])
phi_start= float(obj["phi"])



Start=(x_start,y_start) print("Start:
"+str(Start))
Start_Name='k0'
Points=[[Start_Name,Start,'None']]

#Specify endpoint
Ziel_OK=False while
not Ziel_OK: x_Ziel=float(input("x-
        coordinate of the target point: ")) y_Ziel=float(input("y-coordinate of the target point: "))
        phi_Ziel=float(input("Orientation of the target point: "))

        Target=(x_target,y_target)

        Collision control = inObstacle(Target) print("Collision:
        "+str(Collision control))
        AR_Control=inWorkspace(Target) print("AR_Control:"
        +str(AR_Control))

        if any (collision control) or (AR_Control !=True):
                print("ERROR: Target is not in configuration space!")
                passport
        else:
                Ziel_OK=True
        print("Ziel_OK: "+str(Ziel_OK))
```

```
path_to_goal=False

#Check if direct path to destination is possible if any
(Collision_Path_Obstacle(Start,Destination)):
        print("No direct path to target possible")

else:
        print("Direct path to target found!")
        path_to_goal=True print("Path
to goal: "+str(path_to_goal))


#Finding a path

while not path_to_goal:
        #Creating new points
        Punkt_OK=False while
        not Punkt_OK: Punkt_Name =
                "k{}".format(len(Punkte)) x_Punkt=np.random.randint(AR_x[0],AR_x[1])
                y_Punkt=np.random.randint(AR_y[0],AR_y[1]) print("Any point:
                "+str(x_Punkt)+", "+str(y_Punkt))


                #Determine which is the closest point to any point,
                Calculate distance, determine index minDist=1e10 #only as
                initial distance for t in range(len(points)):

                        dist_to_point=calculate_distance((points[t][1]),(x_point,y_point)) if dist_to_point<=minDist:
                        minDist=dist_to_point

                                ParentPoint=t

                Parent="k{}".format(ParentPoint)
                Koor_Parent=(Points[ParentPoint][1])

                #Vector from parent to any point dx=x_point-points[ParentPoint][1]
                [0] dy=y_point-points[ParentPoint][1][1]

                Shortest vector = [dx,dy]
                LengthVector=math.sqrt((dx**2)+(dy**2))
                UnitVector=[(dx/LengthVector),(dy/LengthVector)]

                #Specify new point
                Abs=150 #Distance at which the new point should be set print("ParentPoint_x: "+str(Points[ParentPoint][1]
                [0])) x_new=Points[ParentPoint][1][0]+Abs*UnitVector[0] y_new=Points[ParentPoint][1]
                [1]+Abs*UnitVector[1] print("New point: "+str(x_new)+", "+str(y_new))


                point_new=(x_new,y_new)
```

```
            Collision = inObstacle(Point_new)
            ARcheck=inWorkspace(Point_new) print("ARcheck:
            "+str(ARcheck)) print("Collision: "+str(Collision))


            if any(Collision) or (ARcheck ==False):
                passport
            else:
                Dot_OK=True
            print("Dot_OK: "+str(Dot_OK))

        #add new item to list
        Points.append([Point_Name, Point_new,Parent]) print("Points: "+str(Points))


        #see if there is an obstacle-free route from the last point on the list
        path to the goal
        LineCheck=Collision_Path_Obstacle(Points[-1][1],Target)

        if any(LineCheck):
            passport
        else:
            path_to_goal=True
        print("Path to goal: "+str(path_to_goal))

#Way back from the finish to the start
path=[points[-1][1]] at_start=False


current_point=points[-1][1] current_parent=points[-1]
[2]

while not at_start: for I in
        range(len(points)): if points[I][0]==current_parent:
            path.insert(0,points[I][1]) current_point=points[I][1]
                current_parent=points[I][2]


            if current_Parent=='None': am_Start=True


path.append(target)


print ("Path: "+str(Path)) print ("Number of
points in path: "+str(len(Path)))
```

```
#Driving the path
delta_phi=(phi_target-phi_start)/(len(path)-1) phi_path=[] for z in
range(len(path)):

        phi=phi_start+z*delta_phi phi_Pfad.append(phi)

print("Phi along path: "+str(phi_path))

p=1
print("Length of path: "+str(len(path))) v=[200] omega=[3]



while p < len(path):
        [v_new,omega_new]=Path planning(path[p][0],path[p][1],phi_path[p],v[p-1],omega[p-1]) v.append(v_new)
        omega .append(omega_new)
        print("At point: "+str(p)) p=p+1



print("At the destination!!!")

while True:
        d_v=Position control(Path[p-1][0],Path[p-1][1],phi_Pfad[p-1]) print("Values from position control: "
        +str(d_v)) d_v_x=d_v[0 ] d_v_y=d_v[1] omega_korr=d_v[2] rad =



        math.atan2(d_v_y,d_v_x)
        v_ges_korr=math.sqrt(d_v_x**2+d_v_y**2)



        print("Position control only ") print("Speed:
        "+str(v_ges_korr)) print("Direction of travel: "+str(rad)) print("Rotational
        speed: "+str(omega_korr))



        #Transfer of data via serial interface
        cmd="{:.0f}".format(v_ges_korr)+"{:.5f}".format(rad)+"{:.5f}".format(omega_korr) cmd=cmd+'\n'

        print(str(cmd))
        arduinoData.write(cmd.encode())

        time.sleep(0.1)
```