FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Parkování modelu autonomního vozidla
Student:	Ondřej Gössel
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Seznamte se s modelem autonomního auta v Laboratoři inteligentních vestavných systémů a naučte se ho programově ovládat.
Proveďte rešerši algoritmů pro parkování vozidla a vyberte nejvhodnější pro existující model.
Na základě rešerše navrhněte a implementujte programové vybavení, které s využitím dat ze vzdálenostních senzorů provede zaparkování vozidla do vymezeného prostoru.
Pro dosažení cíle proveďte případné úpravy existujícího programového vybavení modelu.
Pro konkrétní model a různé typy parkování určete velkosti parkovacích prostorů.
Navržené programové vybavení odladťe v simulaci a pak otestujte přímo na modelu.
Vše řádně zdokumentujte.
Zhodnoťte dostatečnost informace ze senzorů modelu, případně potřebu přidání dat z kamery pro navigaci vozidla při parkování.
Rozsah práce upravte po dohodě s vedoucím práce.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 9. prosince 2022 v Praze.

Bakalářská práce

PARKOVÁNÍ MODELU AUTONOMNÍHO VOZIDLA

Ondřej Gössel

Fakulta informačních technologií
Katedra číslicového návrhu
Vedoucí: Ing. Miroslav Skrbek, Ph.D.
29. června 2023

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Nejprve bych chtěl poděkovat vedoucímu mé práce Ing. Miroslavu Skrbkovi, Ph.D. za ochotnou pomoc při tvorbě této práce.
Dále bych chtěl poděkovat své manželce Tereze za trpělivost a své dceři Magdaléně za potřebnou motivaci.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 29. června 2023

Abstrakt

Tato bakalářská práce se zabývá vývojem programového vybavení pro autonomní parkování modelů vozidel v Laboratoři inteligentních vestavných systémů na FIT ČVUT. V rámci práce byla provedena rešerše algoritmů pro autonomní parkování a bylo zanalyzováno současné hardwarové a programové vybavení modelů. Na základě rešerše a analýzy byl navržen algoritmus pro přesné řízení modelů vozidel při parkování, který byl implementován v jazyce Python. Pro snadnější práci s modely byl implementován terminál, který obsahuje příkazy pro různé funkcionality, například ovládání modelu či jeho kalibraci. Pro snadné testování byl v enginu Godot vytvořen simulátor replikující reálný model vozidla. Navržený systém řízení a parkovací algoritmus byly implementovány a následně systematicky otestovány v simulaci a poté na hardwarovém modelu. Testování parkování bylo prováděno do obdélníkových parkovacích míst různých délek z různých výchozích pozic. V simulaci byla úspěšnost parkování vysoká, kvůli hardwarovým nepřesnostem a nedostatkům byla na hardwarových modelech o něco nižší. Hlavním přínosem této práce je celkové rozšíření programové výbavy modelů, které s nimi do budoucna usnadní práci.

Klíčová slova autonomní vozidlo, autonomní řízení, kinematika, parkování, Godot, Python

Abstract

The aim of this thesis is the development of autonomous parking software for the vehicle models at the Laboratory of Intelligent Embedded Systems at FIT CTU. The thesis includes a research study of algorithms for autonomous parking and an analysis of the current hardware and software equipment of the models. Based on the research and analysis, an algorithm for precise control of the vehicle models during parking was designed and implemented in the Python programming language. To facilitate working with the models, a terminal was implemented, which includes commands for various functionalities, such as model control and calibration. For easy testing, a simulator replicating the real vehicle model was created in the Godot engine. The designed control system and parking algorithm were implemented and systematically tested in the simulation and later on the hardware model. Parking tests were performed in rectangular parking spaces of varying lengths from different starting positions. The success rate of parking in the simulation was high, while on the hardware models, it was lower due to hardware inaccuracies and limitations. The main contribution of this work is the overall enhancement of the software equipment for the models, which will facilitate future work with them.

Keywords autonomous vehicle, autonomous control, kinematics, parking, Godot, Python

Seznam zkratek

PWM	Pulse Width Modulation
PID	Proportional-Integral-Derivative

Úvod

Parkování je narůstajícím problémem po celém světě. Obzvláště v hektickém prostředí městské automobilové dopravy se potýkáme s nedostatkem parkovacích míst, parkování za provozu narušuje plynulost dopravy, drobné nehody mající za následek poškrábaný lak či promáčklý nárazník nejsou neobvyklé. Autonomní parkování je aktuální téma představující jednou z možností, jak řidičům od těchto problémů ulevit. Auto které parkuje autonomně neplýtvá místem, takže šetří cenná parkovací místa, parkuje rychle, takže nezdržuje provoz, a parkuje přesně, takže neohrožuje ostatní zaparkovaná auta.

V této práci technologie autonomního parkování prozkoumám a výsledky své rešerše aplikuji na modely autonomních vozidel v laboratoři inteligentních vestavných systémů.

Práce je určena zejména pro další využití v rámci FIT ČVUT. Očekávám, že by její výsledky mohly být využity například na dnech otevřených dveří a že mnou vytvořený software bude nadále rozvíjen v semestrálních a závěrečných pracích jiných studentů fakulty.

Tato práce navazuje na soubor závěrečných prací rozšiřujících hardwarovou a softwarovou výbavu modelů autonomních vozidel studentů FIT ČVUT, převážně na diplomovou práci *Řízení modelů autonomních vozidel* od Ing. Petra Koláře a nedokončenou bakalářskou práci *Programové vybavení pro autonomní vozidlo* od Davida Ondruška.

Cíl práce

Cílem práce je navrhnout a implementovat software pro modely autonomních vozidel v Laboratoři inteligentních vestavných systémů umožňující jejich parkování do vymezeného prostoru s využitím dat ze vzdálenostních senzorů vozidla. Tento hlavní cíl je možné rozložit na několik dílčích cílů. Zaprvé je třeba zhodnotit současnou výbavu modelů a vytvořit způsob, jak je dostatečně přesně ovládat. Zadruhé je třeba provést rešerši existujících prací na téma autonomního parkování a na základě nabytých poznatků navrhnout a implementovat algoritmus. Zatřetí je třeba vybrat vhodné prostředí pro odladění algoritmů a pak vytvořený software otestovat na modelech.

Struktura práce

V teoretické části nejprve zanalyzuji současnou hardwarovou a softwarovou výbavu modelů vozidel, které mám pro práci k dispozici. Pak prozkoumám existující algoritmy autonomního parkování a vyberu nejvhodnější pro implementaci na dostupné vybavení modelů.

V analytické části navrhnu schéma softwarové architektury celé práce. Dále navrhnu systém přesného řízení modelu. Nakonec navrhnu, jak přesně bude parkování probíhat.

V praktické části implementuji potřebné úpravy stávajícího softwarového vybavení, pak bude následovat provedení implementace zvoleného parkovacího algoritmu. Ladění a testování bude probíhat převážně v simulátoru, jehož tvorba bude rovněž součástí této části mé práce.

V experimentální části vyzkouším vytvořený software jak v simulátoru, tak na modelu. Na základě těchto pokusů zhodnotím dostatečnost současného hardwaru a mnou vytvořeného softwaru.

Kapitola 1
Teoretická část

V této kapitole jsem prozkoumal současné vybavení modelu a provedu rešerši problematiky autonomního parkování. Dále jsem vybral vhodné prostředí pro experimenty.

1.1 Současná výbava vozidla

Laboratoř inteligentních vestavných systémů je vybavena čtyřmi modely automobilů založených na modelech od čínské firmy Sunfounder. Převážně v rámci závěrečných prací jiných studentů byla rozšířena jejich hardwarová i softwarová výbava a od své původní podoby se značně liší.

Výbavu vozidla rozdělím na tři části – samotný hardware, firmware obsluhující elektroniku a software psaný v Pythonu, který spouští uživatel pro ovládání vozidla.

1.1.1 Hardware

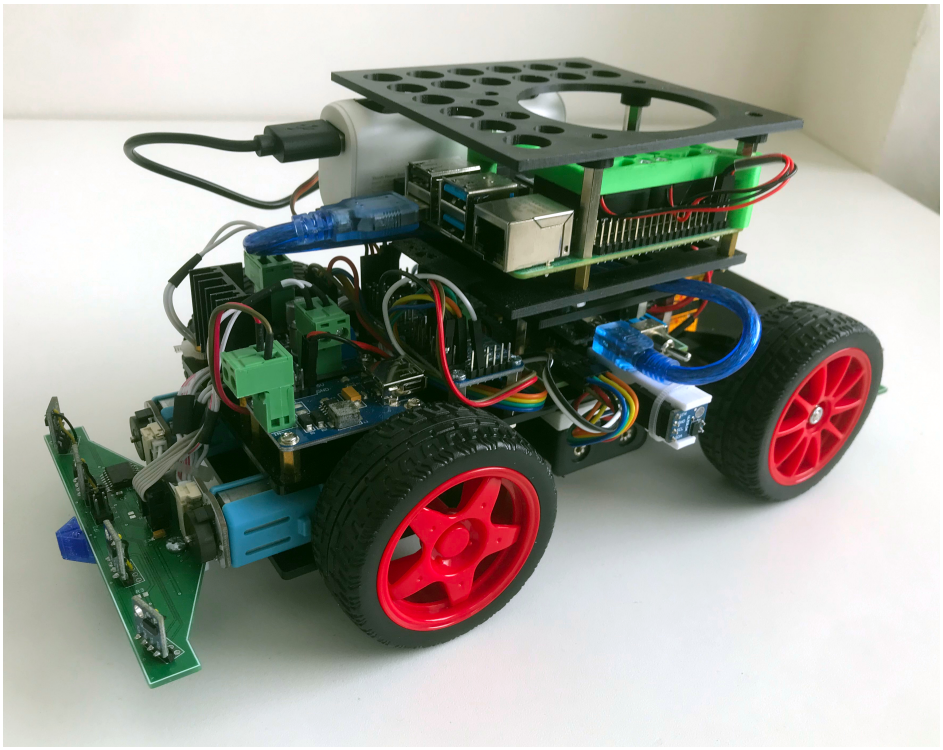
Základním modelem aut v laboratoři je model Sunfounder Smart Car, starší verze 1.0 z roku 2017. Je to model vozidla připomínající osobní automobil s hnanou zadní nápravou a řiditelnou přední nápravou. Největší úpravy na modelu provedl Ing. Kolář v rámci své diplomové práce *Řízení modelů autonomních vozidel* [1], kdy navrhl pro vozidlo tištěný spoj (dále „senzorová lišta“), na který rozmístil 4 senzory vzdálenosti VL53L0X. Pro každé vozidlo zkompletoval dvě tyto lišty, jednu umístil na přední části, druhou na zadní. Další dva senzory umístil po stranách vozidla, směřující přímo doleva a doprava.

Na horní části modelu je umístěno Raspberry Pi, ze kterého uživatel po připojení přes SSH vozidlo ovládá. Pod ním je umístěno Arduino Nano, které zajišťuje komunikaci se servy a motory. Tyto dvě jednotky jsou spolu propojeny pomocí sériové linky.

Zadní náprava je poháněna TT motory řízenými pomocí H-můstku L298N. Přední náprava nemá vlastní pohon a je možné řídit úhel jejího natočení pomocí serva. Serva i H-můstek řídí PWM driver PCA9685. Na přední části modelů je také umístěna USB web kamera, připevněná na podstavci se dvěma servy, což umožňuje její ovládání ve dvou osách [1]. Jelikož se tato práce zaměřuje na práci se senzory vzdálenosti, kamera nebude využita.

Senzor VL53L0X je laserový ToF (Time of Flight) senzor s uváděným maximálním dosahem až 2m. Senzor má několik operačních režimů, které se liší vzorkovací frekvencí, přesností a účelem. V současnosti je na modelech nastaven obecný režim, který poskytuje rozumný kompromis mezi frekvencí a přesností. Tento režim pracuje na frekvenci 30 Hz, na alespoň takové frekvenci bude tedy muset pracovat můj software při provádění měření, aby byla zachycena všechna data[1]. Jelikož

[1]Senzory pracují nezávisle na sobě a při dokončení měření vyšlou do Arduina data v přerušení – pokud by hodnota nebyla včas přečtena, naměřená hodnota by byla přepsána a ztracena.



Obrázek 1.1 Současná podoba modelu. Vlevo na fotografii je na zadní části modelu vidět senzorová lišta a motory s enkodéry. Nahoře je Raspberry Pi, připojené na Arduino modrým USB kabelem.

senzor měří vlastní laserový paprsek, je zde závislost na odrazivosti povrchu a na světelných podmínkách. Uvnitř dokáže přesně změřit vzdálenost bílého objektu, který je až 2 m daleko, venku a proti tmavšímu cíli však může být maximální použitelná vzdálenost pouhých 40 cm. Rovněž vykazuje proti tmavým objektům znatelně vyšší směrodatnou odchylku měření [2]. Tato skutečnost bude muset být při testování zohledněna a v případě nutnosti budou všechny prvky parkovacího prostoru vyměněny za bílé.

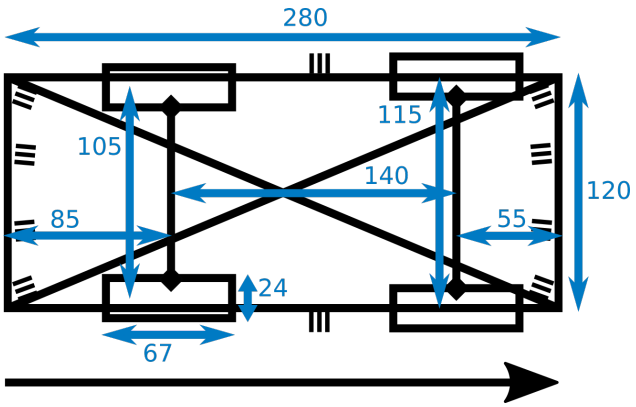
Model využívá Ackermannovo řízení a servem je řízeno jen levé přední kolo. Při poslání příkazu pro nastavení úhlu natočení kol je tedy levé přední kolo autoritativní. Důsledkem je, že model opisuje při stejném nastaveném úhlu při zatáčení doprava menší kružnici, než při zatáčení doleva. Nastavení serva tedy bude muset být přepočítáno tak, aby auto zatáčelo pod stejným úhlem stejně do obou směrů, k čemuž potřebujeme znát jen rozvor a rozchod. Podstatným důsledkem tohoto faktu je také to, že je rozsah řízení asymetrický.

Změřil jsem podstatné rozměry auta a podle nich vytvořil tabulku 1.1 a schéma 1.2. Údaje jsem místo hledání v údajích výrobce raději změřil sám kvůli mnohým úpravám, které na modelu proběhly. Tabulka obsahuje stejný překlad rozměrů jako v kódu a konfiguračním souboru (bude představen později). Pomocí těchto rozměrů půjdou odvodit některé kritické údaje, například poloměr otáčení či minimální délka parkovacího místa.

Rozměr česky	Rozměr v kódu	Hodnota v cm
Délka auta	car_length	28
Šířka auta	car_width	12
Rozvor	wheelbase	14
Rozchod[1]	track	11,5
Zadní přesah	rear_overhang	8,5
Přední přesah	front_overhang	5,5
Průměr kola	wheel_diameter	6,7
Šířka kola[2]	wheel_width	2,4

Tabulka 1.1 Tabulka rozměrů modelu

[1]Zjednodušení, které by nemělo mít velký vliv. Reálně se vzdálenosti na nápravách liší o centimetr.
[2]Nebylo využito.



Obrázek 1.2 Schéma modelu s hlavními rozměry v mm. Trojitá čára označuje umístění senzoru.

Výpis kódu 1.1 Nedokončený firmware

```
...
#define constrain(a, b, c) a
...
set_pid_parameters(0.01, 0.1, 0.001, 20e-3);
// Off the top of my head parameters
// just so the PIDs do at least something
...
void loop(){
    [30 radku zakomentovaneho kodu ktery ovladal PID]
...
```

1.1.2 Firmware

Firmwarem nazývám kód pro Arduino psaný v C zajišťující komunikaci s motory, servy a senzory a předpokládající komunikaci s Raspberry Pi pomocí sériové linky, na němž uživatel spouští Pythonové příkazy. V práci navážu na firmware, který vytvořil David Ondrušek. Tento firmware se zaměřuje na implementaci obecných ovládacích funkcí pro model. Kromě mnoha jednoúčelových ovládacích funkcí (obsluhujících např. pouze natočení kol) implementoval i obecný příkaz, který přijímá strukturu s parametry pro všechny ovladatelné prvky a odesílá zpět po sériové lince odpověď obsahující kompletní stav vozidla, včetně stavů enkodérů a senzorů. Tento univerzální příkaz (nazvaný jednoduše „L-příkaz“) byl navržen pro využití ve skriptech a přesně tak jsem ho ve své práci využil. Firmware byl evidentně navržen tak, aby byl na vyšší úrovni skriptů v Pythonu spuštěn cyklus, který opakovaně posílá vozidlu příkazy, které upravuje na základě dat, které obdrží z vozidla. Tomu nasvědčuje také to, že firmware drží od přijetí příkazu hodnotu pro PWM motorů pouze po dobu jedné vteřiny. To je z důvodu bezpečnosti užitečné, například v případě náhlého výpadku sériového spojení, kdy po vteřině začne vozidlo samo brzdit. Ocenil jsem, že firmware po připojení na sériovou linku provede kompletní kontrolu vozidla – zkontroluje napěťovou úroveň napájení, funkci motorů pomocí enkodérů, přítomnost senzorových lišt apod.

Firmware znatelně trpí tím, že daná bakalářská práce nebyla dokončena. Některá volaná makra nebyla implementována a vrací pouze své nezměněné parametry a bez dokumentace jsem neměl šanci zjistit jejich plánovaný účel. Velké části kódu, které na první pohled působí funkčně, jsou bez vysvětlení zakomentovány. Je implementována PID regulace ujeté vzdálenosti, obsahuje však pouze smyšlené a neotestované parametry a nepodařilo se mi ji zprovoznit (regulaci ujeté vzdálenosti jsem nakonec implementoval na vyšší úrovni). Ilustrační příklady jsou uvedeny ve výpisu 1.1.

Přes tyto drobnější nedostatky však firmware hodnotím jako použitelný. Po analýze funkcionalit jsem zjistil, že bylo třeba provést jedinou znatelnou úpravu. Z datasheetu H-můstku [3] jsem vyčetl, že existuje podpora aktivního brzdění, toto však ve firmwaru podporováno doposud nebylo a musel jsem tuto funkcionalitu přidat.

1.1.3 Python API

Pan Ondrušek vytvořil krátký Pythonový modul implementující připojení na Arduino se svým firmwarem pomocí sériové linky. Veškerá komunikace s Arduinem je zajištěna pomocí jediné funkce, která implementuje komunikaci s výše zmíněným univerzálním příkazem z firmwaru. Jejími parametry jsou PWM motorů, úhel natočení kol a parametry natočení kamery, tedy vše, co jde na modelu ovládat. Vrací strukturovaný současný stav vozidla. Celá struktura vstupních a výstupních dat je uvedena ve výpisu kódu 1.2. Za každou položkou je uvedena velikost v bytech. Hlavička funkce a příklady použití jsou uvedeny ve výpisu 1.3. Tento modul tvoří dostatečný základ pro mou práci.

Výpis kódu 1.2 Vstupní a výstupní data univerzálního příkazu

```
Lcommand = { #Tuto strukturu Python API posila na seriovou linku.
    "command_id" : "c", # 1
    "left_pwm" : "h",   # 2 # Povsimneme si, ze levy a pravy motor
    "right_pwm" : "h",  # 2 # lze ovladat oddelene.
    "steering" : "b",   # 1
    "pan" : "b",        # 1
    "tilt" : "b",       # 1
}

Lresponse = { #Tuto strukturu Python API ocekava od seriove linky...
    "command_id" : "c", # 1
    "micros" : "I",     # 4
    "left_count" : "i", # 4
    "right_count" : "i",# 4
}
for i in range(10): #...vcetne udaju ze vsech deseti senzoru
    Lresponse[f"sensor{i}"] = "H" # 2
```

Výpis kódu 1.3 Použití univerzálního příkazu

```
def L_command_and_response(
    port,       #Cilove zarizeni
    left_pwm,   #PWM pro levy motor v intervalu <-1;1>
    right_pwm,  #PWM pro pravy motor
    steering,   #Uhel natoceni kol ve stupnich
    pan,        #Uhel natoceni kamery
    tilt        #Uhel naklonu kamery
):
    ...
L_command_and_response(serial, 1, 1, 0,0,0) #Pohyb vozidla rovne
L_command_and_response(serial,-1,-1,30,0,0) #Couvani doprava
L_command_and_response(serial, 0, 0, 0,0,0) #Do motoru vyslana 0 => brzdeni
```

1.2 Autonomní parkování

Problematika automatického parkování již byla zkoumána mnoha způsoby, jak ve výzkumu, tak v průmyslu. Rozsah sofistikace těchto způsobů je velmi široký. V průmyslu už nějakou dobu existuje ta nejjednodušší možnost, kdy auto za řidiče provede jen samotné přednastavené pohyby a to, zda-li je parkování vůbec možné a bezpečné, je zodpovědnost řidiče. V simulacích už byla naopak zpracována velmi působivá řešení s využitím umělé inteligence, kdy se model naučí zaparkovat sám na základě jemu dostupných dat.

Moje řešení svou úrovní složitosti leží mezi těmito dvěma extrémy. Prozkoumal jsem práce, které na problém nahlížejí čistě geometricky. S využitím znalostí rozměrů vozidla a potenciálního parkovacího místa se zjistí, zda-li je parkování možné. Pokud ano, je pak možné vypočítat parametry bezpečné parkovací trajektorie.

Po dohodě s vedoucím práce jsem podrobně prozkoumal a implementoval pouze podélné parkování, jakožto reprezentativní vzorek celé problematiky parkování, který řidičům činí potíže nejčastěji. Ostatní typy parkování je možné implementovat analogicky a některé [4] citované zdroje pro to poskytují dostatečné teoretické podklady.

1.2.1 Historický přehled

Technologie autonomního parkování jsou řidičům dostupné už dvacet let. První výrobce, který začal prodávat auto pomáhající řidičům s parkováním, byla Toyota v roce 2003 se svým modelem Prius Hybrid. Vyvinutý systém nazvala „Intelligent Parking Assist System“. Na přední a zadní části vozu byla usazena kamera. Pomocí obrazů z kamer bylo odhadnuto parkovací místo a nabídnuto řidiči na počítači osazeném dotykovou obrazovkou. Řidič si parkovací místo dotykem vybral a auto převzalo ovládání volantu, pedály zůstaly zodpovědností řidiče. První auto relevantní pro tuto práci vydal Lexus v roce 2006 svým modelem Lexus LS 460. Jeho parkovací asistent nepotřeboval kamery a stačily mu pouze senzory vzdálenosti. Pedály stále ovládal řidič. Od té doby se tato funkcionalita na běžných osobních automobilech stávala čím dál tím častější. V dnešní době mají automobilky ambicióznější cíl zvaný Automatic Valet Parking (AVP). Tento systém automatizuje celý parkovací proces, vozidlo má být schopno samo v garáži najít parkovací místo a zaparkovat do něj bez asistence či dokonce přítomnosti řidiče [5].

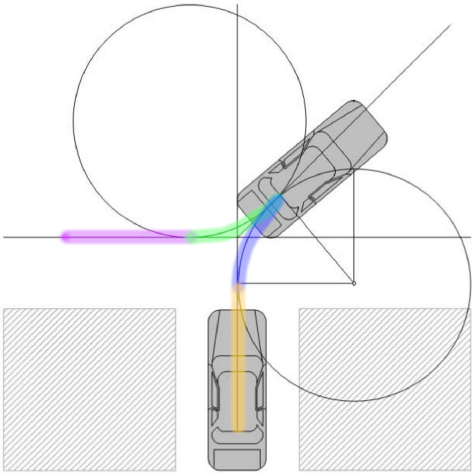
1.2.2 Princip a použité značení

Průběh provedení celého parkovacího pohybu ze zadaného výchozího bodu do cílového parkovacího místa se dá na teoretické rovině rozdělit na posloupnost několika dílčích pohybů, které budu dále nazývat také úseky. Podle literatury [4] zavádím pro úseky následující značení:

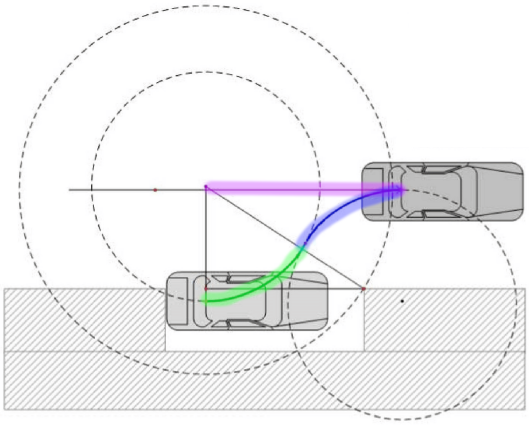
- S značí pohyb rovně, tedy s koly nastavenými na nulový úhel.
- R značí pohyb doprava, tedy s koly nastavenými na kladný úhel.
- L značí pohyb doleva, tedy s koly nastavenými na záporný úhel.

Dále:

- Horní index + značí pohyb vpřed. S^{+} je tedy rovný pohyb vpřed.
- Horní index − značí pohyb vzad. S^{-} je tedy rovný pohyb vzad.
- Horní index $*$ značí pohyb předem neznámým směrem. S^{*} je tedy obecný rovný pohyb. Toto značení se v literatuře nevyskytuje, ukázalo se pro mě však jako potřebné, zavádím ho tedy já.



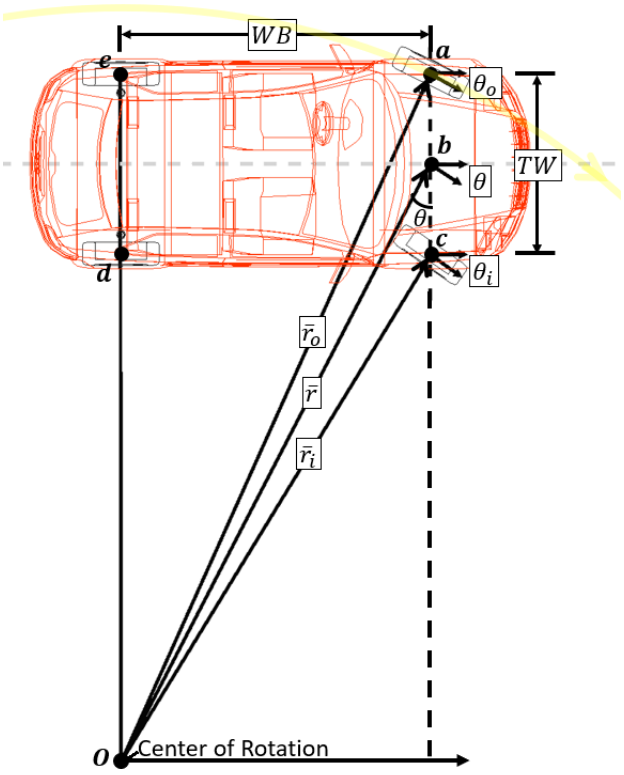
Obrázek 1.3 Kolmé parkování rozděleno na úseky (S^+, L^+, R^-, S^-). (Převzato z: [4])



Obrázek 1.4 Podélné parkování rozděleno na úseky (S^+, R^-, L^-). (Převzato z: [4])

Pro ilustraci uvádím dva příklady, úseky jsou barevně odlišeny. Obrázek 1.3 zobrazuje kolmé parkování složené ze čtyř úseků, toto parkování lze zapsat jako (S^+, L^+, R^-, S^-). Obrázek 1.4 zobrazuje podélné parkování (S^+, R^-, L^-). Je nutné zmínit, že toto není obecně platný popis daných typů parkování. Podle počátečního umístění vozidla může mít kolmé parkování tři až pět úseků [4]. Podélné parkování bude mít vždy tři úseky, není však předem známé, jestli vozidlo v prvním úseku pojede vpřed, či dozadu. Proto uvádím obecný tvar podélného parkování jako (S^*, R^-, L^-). Volitelně je možné ho provádět jako (S^*, R^-, L^-, S^+), kdy poslední pohyb vyrovná velikost mezer před a za vozidlem.

Pro výkon podélného parkování tedy stačí zjistit rozměry parkovacího místa, najít vhodné délky a úhly pro tyto úseky a pak vypočítané pohyby co nejpřesněji provést.



Obrázek 1.5 Vztah mezi poloměry opisovaných kružnic. (Převzato z: [7])

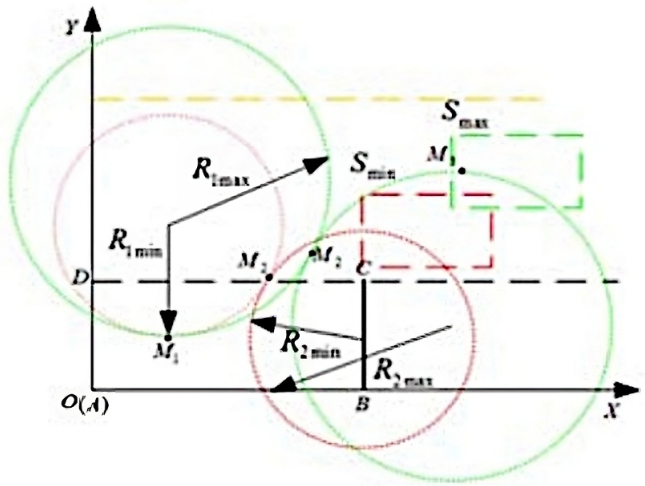
1.2.3 Kinematika

Jakákoliv práce s touto tématikou musí předpokládat nějaký model kinematiky, kterým se vozidlo řídí, ať už je reálné, či v simulátoru. Budeme jednoduše jako v literatuře [6] předpokládat, že kola vůbec neprokluzují a že na kolech nastavený úhel je přesně ten úhel, podle kterého se vozidlo pohybuje. Pro naše lehké a pomalu se pohybující modely je tento předpoklad rozumný. Díky velmi nízkým rychlostem při parkování je tento předpoklad rozumný i pro parkování osobních automobilů [4], pro jejich běžný pohyb však ne. Pak platí rovnice 1.1 [4] pro odvození poloměru kružnice, kterou vozidlo opisuje při daném úhlu natočení kol. WB značí rozvor (wheelbase), θ úhel natočení kol.

$$r = \frac{\mathrm{WB}}{\tan(\theta)} \tag{1.1}$$

Situace modelů je však složitější, úhel θ neznáme, známe pouze úhel levého kola za předpokladu Ackermannova řízení. Úhel θ značí úhel pro střed nápravy, tedy takový úhel, jaký by pro kružnici o daném poloměru potřebovalo jednostopé vozidlo, jak ilustruje 1.5. Vozidlu budeme zadávat úhel θ, reálně však budeme při zatáčení doprava nastavovat vnější úhel θ_o a při zatáčení doleva vnitřní úhel θ_i. Převod [7] v rovnicích 1.2 pro vnitřní úhel a 1.3 pro vnější úhel vyžaduje kromě zadaného úhlu θ_i také rozvor WB a rozchod TW (track width). Tento převod je čistě geometrický.

$$\tan(\theta_i) = \frac{\mathrm{WB}}{r - \frac{\mathrm{TW}}{2}} = \frac{\mathrm{TW}}{\frac{\mathrm{WB}}{\tan(\theta)} - \frac{\mathrm{TW}}{2}} \tag{1.2}$$



Obrázek 1.6 Zvětšování poloměrů kružnic. (Převzato z: [8])

$$\tan(\theta_o) = \frac{\mathrm{WB}}{\frac{\mathrm{WB}}{\tan(\theta)} + \frac{\mathrm{TW}}{2}} \tag{1.3}$$

Zajímavost, která vyplývá z mého předpokladu, je také to, že jsou všechny pohyby vratné. Po vykonání libovolného úseku X^+ je možné auto vrátit do původní pozice příslušným úsekem X^-.

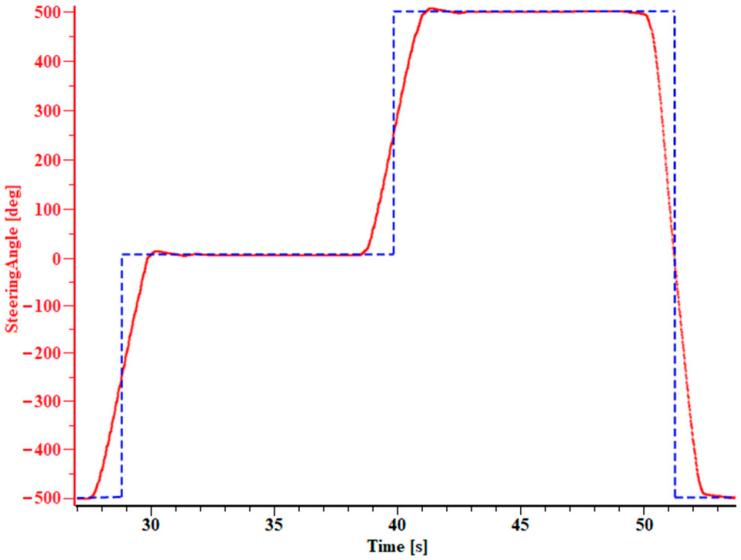
1.2.4 Natočení kol

Při parkování do teoreticky nejmenšího parkovacího prostoru co nejblíže, jak je možné, je jasné, že bude muset být použit nejostřejší možný úhel natočení kol. Pokud je však parkovací místo dále či větší, máme na výběr jak postupovat.

Jak navrhli Maoyue et al. [8], můžeme zmenšovat úhel natočení kol a opisovat tak s rostoucí vzdáleností od parkovacího místa větší kružnici, jak lze vidět na obrázku 1.6. Středy obou kružnic i bod jejich dotyku se posouvají. Tento postup je výhodný v tom, že se auto pohybuje plynuleji a působí přirozeněji, zároveň přečnívá do provozu jen tak málo, jak je nezbytně nutné.

Druhá možnost předpokládá, že je úhel kol při zatáčení konstantní, jak uvažoval například Han [4]. Kružnice mají vždy stejný poloměr, bez ohledu na vzdálenost od parkovacího místa. Střed kružnice pro konečný úsek L^- také zůstává na místě. Jediné, co se mění, je bod dotyku kružnic pro R^- a L^-. Tento postup vytvoří trajektorii, která je obzvláště ve větších vzdálenostech prudší a levý přední roh vozidla při tomto prudkém zatočení přečnívá do provozu více. Pokud však uvažujeme vzdálenosti jen v rámci jednoho typického jízdního pruhu, toto za problém nepovažuji. Velkou výhodu má tato možnost v tom, že se úplně odstraní jedna proměnná a následující sekce s popisem a výpočtem trajektorií se zjednoduší a proto jsem ji také zvolil. Úseky R^* a L^* jsou od tohoto bodu dále definovány jako pohyby doprava a doleva s největším možným natočením kol, tedy při opisování nejmenší možné kružnice.

Li et al. [6] poznamenali, že v prostředí reálného osobního automobilu se naše teoretická trajektorie nejeví jako dostatečná, jelikož obsahuje mezi jednotlivými úseky náhlé změny natočení kol, mezi R^- a L^- dokonce o celý rozsah otáčení. V reálném automobilu by toto ostré otáčení na místě zapříčinilo velice rychlé opotřebení pneumatik, což není přípustné. Odvodili proto takový



Obrázek 1.7 Vyhlazení změn úhlu natočení kol za jízdy dle Li et al. (Převzato z: [6])

přepočet úhlu natočení kol, že vozidlo opisuje stejnou trajektorii, aniž by vozidlo muselo otáčet kola, když stojí, jak je ilustrováno na obrázku 1.7. Nepředpokládám, že by toto mělo být na našich lehkých modelech nutné. Jediné, co udělám je že před výkonem daného úseku zlomek vteřiny po zadání úhlu kol vyčkám, aby se servo stihlo nastavit.

1.2.5 Minimální vzdálenosti

Se znalostí parametrů vozidla včetně jeho poloměru otáčení lze vypočítat rozměry vzdálenost parkovacího místa pro podélné parkování. Předpokládáme obdélníkový tvar parkovacího místa, zjišťujeme tedy šířku a délku. Z obrázku 1.8 je lze odvodit [4]. Nejdřív budeme potřebovat pomocné poloměry. V první rovnici uvádím pro kontext i značení z dřívější podsekce o kinematice.

$$R = \frac{l}{\tan(\theta)} (= r = \frac{\mathrm{WB}}{\tan(\theta)}) \tag{1.4}$$

$$R_A = \sqrt{\left(R + \frac{w_0}{2}\right)^2 + p_r^2} \tag{1.5}$$

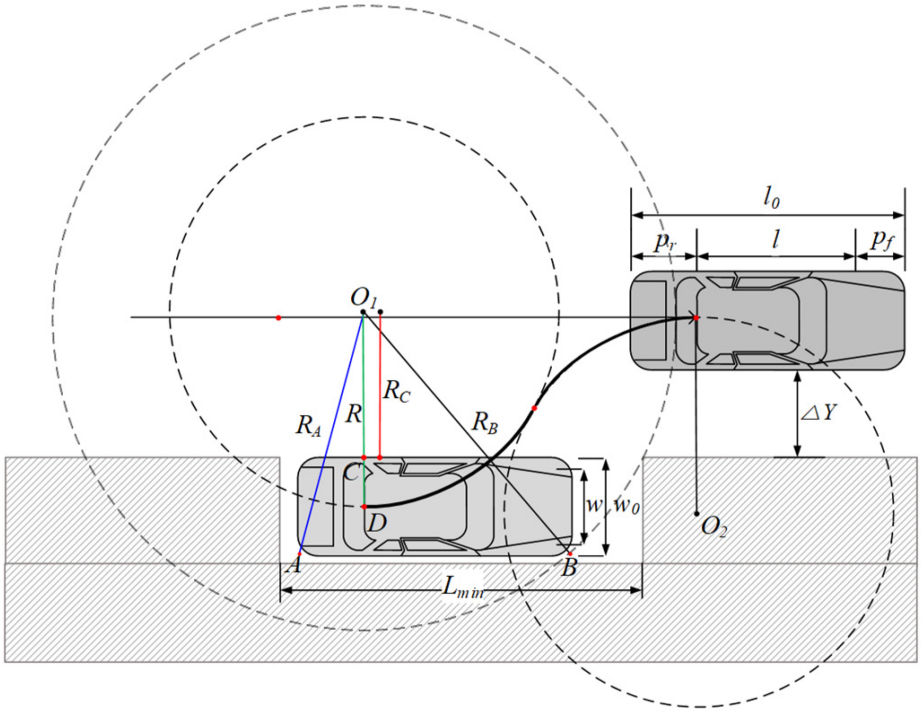
$$R_B = \sqrt{\left(R + \frac{w_0}{2}\right)^2 + (l + p_f)^2} \tag{1.6}$$

$$R_c = R - \frac{w_0}{2} \tag{1.7}$$

Minimální délka je odvozena od toho, že bod B nesmí zavadit o překážku před autem a že se do místa musí vejít jeho zadní přesah:

$$L_{\min} = \sqrt{R_B^2 - R_C^2} + p_r \tag{1.8}$$

Jako poznámku uvádím, že jelikož jsou pohyby vratné, je toto zároveň minimální délka parkovacího místa pro to, aby z něj šlo vyjet pouhým (L^+, R^+). Han [4] navrhl možná řešení toho, jak postupovat, pokud je parkovací místo delší než délka vozidla, avšak kratší než L_{min}. Vypočítal například, jak vozidlo do takovéhoto místa zaparkovat tak, že jeho část po straně přečnívá, kdy se



Obrázek 1.8 Odvození minimálních rozměrů. (Převzato z: [4])

přečnívající část vozidla zmenšuje, čím víc se délka daného místa blíží té minimální. Dále navrhl metodu opakovaného parkování dle obrázku 1.9, touto metodou je teoreticky možné zaparkovat vozidlo úplně i pokud je dané parkovací místo jen nepatrně delší než délka vozidla. Pro parkování v provozu ji nepovažuji za příliš praktickou, naopak velký potenciál pro její využití vidím díky vratnosti pohybů pro vyjetí z parkovacího místa v situaci, kdy řidič auta před námi zaparkoval příliš blízko našeho auta.

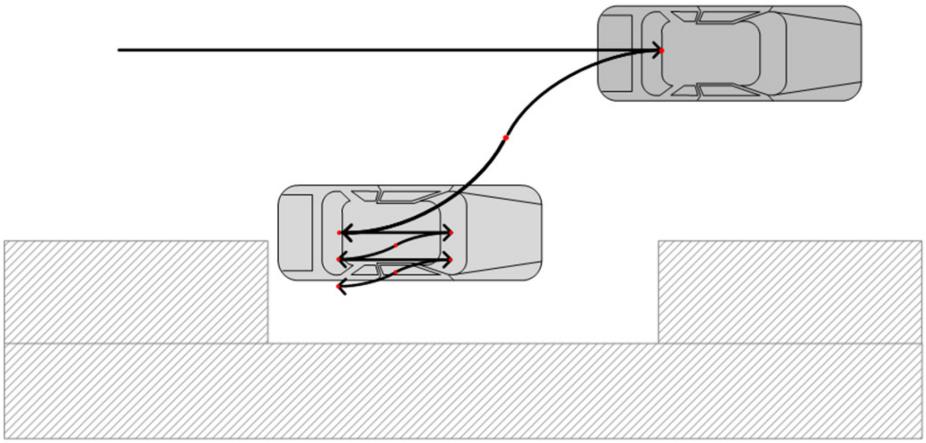
Minimální šířka je odvozena od toho, že bod A nesmí zavadit o spodní hranu parkovacího místa. V tomto případě je odvození jednodušší:

$$W_{\min} = R_A - R_C \tag{1.9}$$

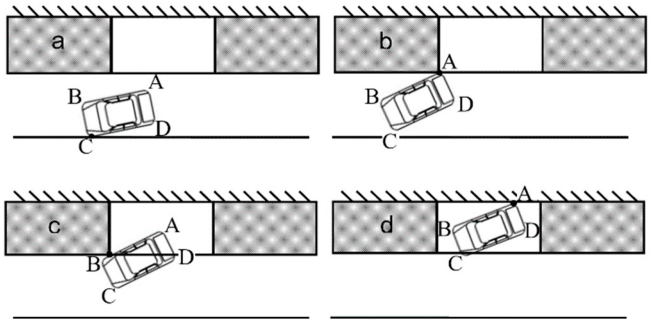
Li et al. [6] uvedli dle obrázku 1.10 možné body kolize auta s okolím parkovacího místa:

- Situace „a“ nepředstavuje reálnou kolizi, nýbrž reprezentuje přečnívání auta ze svého jízdního pruhu. Tato situace nebude ošetřena, jen pomocí senzorů vzdálenosti to není možné.
- Situace „b“ a „c“ jsou ošetřeny dodržením minimální délky parkovacího místa a správnou délkou všech úseků.
- Situace „d“ je ošetřena dodržením minimální šířky.

Je samozřejmé, že vše výše zmíněné platí jen při dokonale přesném řízení, které je obzvlášť na reálném modelu nedosažitelné. Při implementaci bude třeba zavést tolerance, v praxi konstanty, které se k minimálním rozměrům přičtou. Všechny poznatky z této sekce budou využity při konkrétním výpočtu parametrů parkování v analytické části.



Obrázek 1.9 Parkování opakovanými pohyby. (Převzato z: [4])



Obrázek 1.10 Možné kolize. (Převzato z: [6])

1.3 Výběr simulačního prostředí

1.3.1 Existující řešení

Existujících řešení pro simulaci kinematiky vozidel je celá řada. Popíšu CarSim a Webots.

CarSim [9] je komerční balíček software pro simulaci vozidel. Umožňuje velice přesnou a detailní simulaci kinematiky osobních automobilů. Umožňuje simulovat všechny hlavní ovládací prvky, včetně spojky či převodovky. Má snadné napojení na ostatní software a používá se několika univerzitami ve výzkumu, používají ho některé automobilky pro testování funkcionalit svých vozidel, např. bezpečnostních prvků jako ABS, dále nachází uplatnění jako software pro závodní simulátory. Předpokládám, že by tento software pro účely této práce uplatnit šel, trvalo by však dlouho se s ním naučit a bylo by potřeba upravit některý ze vzorových modelů tak, aby měl podobné vlastnosti jako modely v laboratoři.

Webots [10] je open source simulátor robotiky, který se používá jak ve výzkumu, tak v průmyslu. Kromě simulace automobilů a jejich modelů nachází Webots uplatnění například při simulaci průmyslových robotů jako Unimate či humanoidních robotů jako Nao. Webots má svou vlastní knihovnu veřejně dostupných modelů, žádný z modelů od firmy Sunfounder tam však zatím není. Tento simulátor je pro tuto práci vhodnější – kromě práce s předem vytvořenými modely je možné relativně rychle zkonstruovat vlastní model z předem vytvořených komponent, což jsou například strukturní prvky, serva, motory, ale i různé senzory. Stále má však podle mého názoru pro tuto práci až moc funkcionalit a trvalo by dlouho v něm začít efektivně pracovat.

1.3.2 Godot

Mé požadavky na simulační software byly nenáročné (naprosto základní kinematika vozidla, možnost propojení s pythonovými skripty), avšak vcelku specifické (nestandardní vozidlo a jeho rozměry, preference 2D prostředí pro lepší ilustraci měření a trajektorií). Dospěl jsem k závěru, že nejefektivnější možnost je tvorba vlastního simulačního prostředí, které bude umět jen ty funkcionality, které budu potřebovat. Pro tvorbu simulátoru byl zvolen Godot. Godot [11] je svobodný a open source engine pro vývoj her, jehož funkcionality umožnily velice rychlý vznik simulátoru – 2D fyzika pro pohyb a kolize modelu, raytrace pro senzory vzdálenosti, komponenty uživatelského rozhraní, které umožnily nejen jednoduché ovládání, ale i výpis stavu senzorů přímo na ně apod. Umožňuje vytvářet složitější objekty skládáním základních komponent, pro jejich programování pak podporuje několik programovacích jazyků, kromě vlastního jazyka enginu zvaného GDScript je to například C# či C++. Budu používat GDScript, je to jazyk velice podobný Pythonu optimalizovaný pro specifické prostředí enginu. Existuje několik možností, jak se spojit se skripty psanými v Pythonu, například pomocí modulu GDSercomm pro sériovou linku či pomocí websocketů.

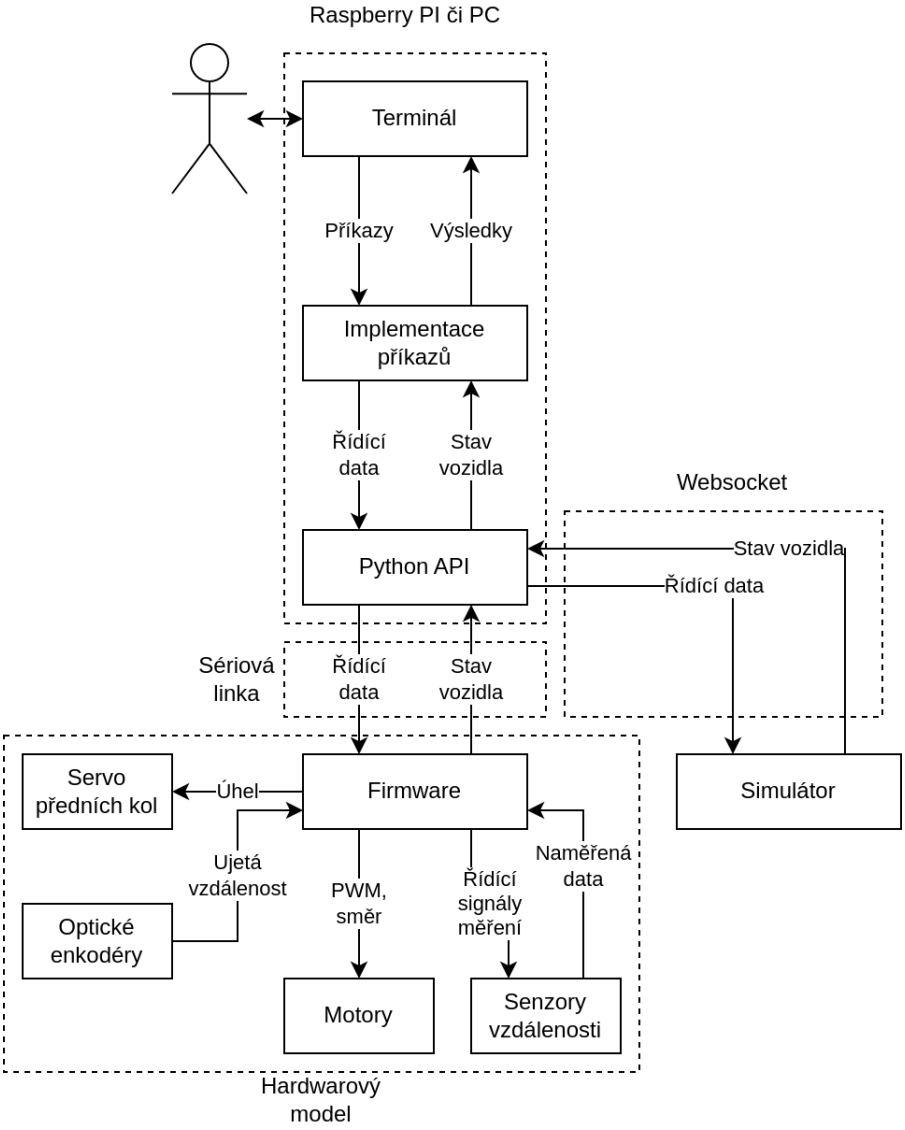
Kapitola 2
Analytická část

V této kapitole se věnuji návrhu těm částem mé práce, které vyžadují před implementací hlubší analýzu. Popsal jsem schéma celého projektu, systém přesného řízení a samotný parkovací algoritmus.

2.1 Schéma řídícího systému

Na obrázku 2.1 je schéma částí řídícího systému a jejich vztahů. Obecně řečeno, směrem dolů putují řídící data (PWM pro motory, úhel natočení pro servo kol), směrem nahoru putuje stav vozidla (ujetá vzdálenost, údaje ze senzorů). Následuje stručný přehled:

- Terminál slouží uživateli k zadávání příkazů, pomocí něj jsou uživateli vypisovány výsledky příkazů a případné chyby i z nižších úrovní.
- Funkce implementující jednotlivé příkazy je možné volat z terminálu. Přes Python API vysílají řídící parametry a rozhodují se dle přijatého stavu.
- Python API zajišťuje příjem a odesílání strukturovaných dat do modelu či simulátoru. Provádí serializaci odchozích dat a interpretaci příchozích dat.
- Simulátor simuluje hardware a firmware dohromady. S Python API komunikuje přes websocket, přičemž si předávají data s naprosto stejnou strukturou, jako kdyby probíhala komunikace s reálným modelem přes sériovou linku.
- Hardware a firmware jsem popsal dostatečně v teoretické části. Firmware pomocí dat přijatých po sériové lince řídí elektroniku, zpět do Python API posílá stav vozidla.



Obrázek 2.1 Schéma projektu

2.2 Řízení ujeté vzdálenosti

Navrhl jsem algoritmus přesného a bezpečného řízení ujeté vzdálenosti pomocí odhadu brzdné dráhy, lineární interpolace PWM motoru a cyklické opravy chyby.

2.2.1 Princip

Princip algoritmu jsem popsal vývojovým diagramem 2.2, který je dále vysvětlen. Obecně toto řešení spočívá v tom, že na začátku vozidlo zjistí doposud ujetou vzdálenost pomocí enkodérů na motorech a vypočítá její kýžený cílový stav přičtením požadované vzdálenosti k ujetí. V cyklu se opakovaným pohybem v jednom směru pokouší této hodnoty enkodérů dosáhnout, přičemž v každém pokusu vznikne nevyhnutelně nějaká chyba. Tato chyba je opakovaně v cyklu opravována tím, že je vozidlu zadáno ujet opačnou hodnotu této chyby. Tento proces probíhá do té doby, dokud se chyba nedostane pod konstantu definující tolerovanou úroveň chyby, nebo dokud oprava nezabere příliš mnoho cyklů, aby byla zaručena konečnost tohoto algoritmu.

2.2.2 Odhad brzdné dráhy

Vozidlo má při pohybu setrvačnost a pokud bychom se ho pokusili zabrzdit až v momentu dosažení cílové vzdálenosti, vozidlo by dojelo nezanedbatelně dále, což není přípustné. Odhad brzdné dráhy plní dvě funkce, zaprvé významně snižuje počet potřebných korekcí, protože se vozidlo s každým pokusem dostane blíže k cíli, zadruhé přispívá k bezpečnosti tím, že vozidlo významně nepřesáhne ujetou vzdálenost, což by mohlo mít za následek kolize. Tento odhad probíhá statisticky, kdy se naměří dostatečný objem dat párů okamžité rychlosti při začátku brzdění a příslušné brzdné dráhy a tato data se dosadí do vhodného statistického modelu.

Brzdná dráha sice teoreticky roste s druhou mocninou rychlosti, podle naměřených dat (bylo zpracováno 5 měření, každé čítající přibližně 30 vzorků) však vykazuje aktivně brzděný model spíše lineární závislost brzdné dráhy na rychlosti. Po provedení kvadratické regrese byl vždy kvadratický člen velmi blízko nuly. Toto ilustruje obrázek 2.3, kde byla na naměřená data aplikována jak lineární, tak kvadratická regrese a kvadratický člen vyšel v tomto případě dokonce záporný. Lineární regrese konzistentně poskytovala dobře použitelný odhad – rozdíl naměřené a odhadnuté brzdné dráhy pro danou rychlost nikdy nepřesáhl 1 cm a kolem tohoto pozorování budou odvozeny bezpečnostní odstupy.

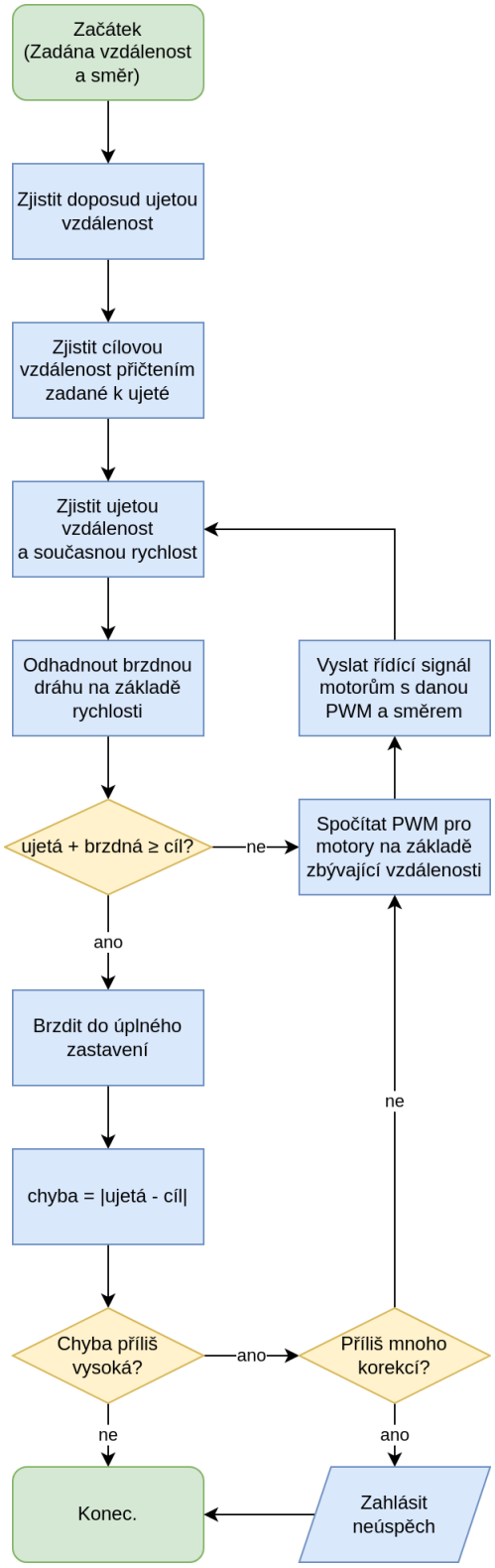
Absolutní člen výsledné přímky by měl být nulový (pokud nebude, značí to, že má vozidlo jiné vlastnosti při pohybu vpřed a vzad, předpokládám, že vzorky budou obsahovat brzdnou dráhu z obou směrů pohybu).

Tento odhad je samozřejmě směrodatný jen pro konkrétní typ povrchu vozovky, významnější změna typu povrchu tedy vyžaduje zopakovat měření a tvorbu modelu.

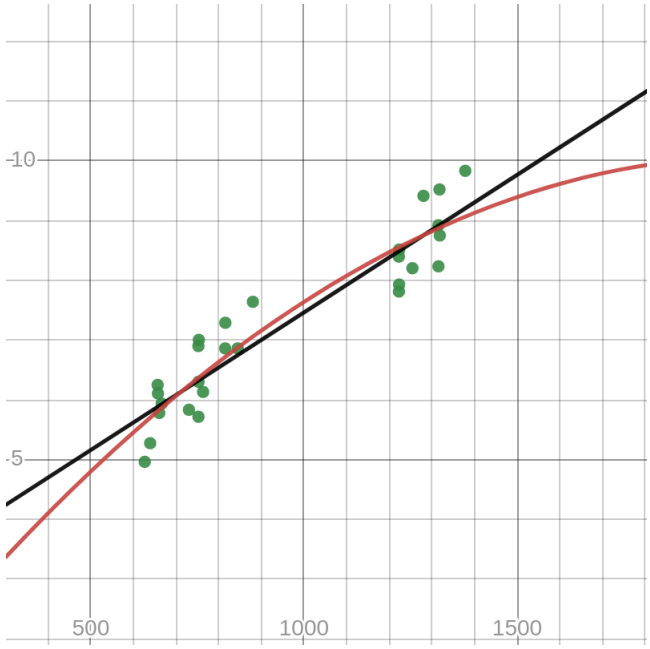
2.2.3 Interpolace PWM

Lineární interpolace PWM motorů zajišťuje to, že když se vozidlo blíží k cíli, průběžně zpomaluje. Hodnota pro PWM je tedy závislá na zbývající vzdálenosti. Tato interpolace potřebuje čtyři hodnoty – dolní a horní hranice vzdáleností a minimální a maximální hodnoty PWM. Tyto hodnoty budou zjištěny experimentálně. Celý systém je na správném nastavení těchto hodnot závislý, přičemž nejkritičtější hodnotou je minimální hodnota PWM. Pokud bude nastavena příliš vysoko, vozidlo nebude schopné udělat nejjemnější konečné korekce a bude jen oscilovat kolem cílové vzdálenosti podobně jako špatně nastavený PID regulátor.

Tímto systémem je možné zajistit velmi přesné řízení, samozřejmě za předpokladu, že jsou hodnoty na enkodérech správné.



Obrázek 2.2 Vývojový diagram řízení ujeté vzdálenost



Obrázek 2.3 Odhad brzdných drah pomocí kvadratické a lineární regrese. Na ose x rychlost v interních jednotkách, na ose y brzdná dráha v centimetrech.

2.3 Parkovací algoritmus

Na základě rešerše z teoretické části jsem navrhl následující parkovací algoritmus, který využívá data ze senzorů vzdálenosti.

2.3.1 Obecný princip

Na začátku předpokládám, že je na začátku auto umístěno rovnoběžně s vozidlem, před které parkujeme, tedy že parkovací místo leží napravo před naším autem. Do tohoto stavu musí uživatel vozidlo dostat sám. Samotný algoritmus se dá rozfázovat následovně:

1. Vozidlo ze své výchozí pozice jede vpřed o nějakou konstantní vzdálenost (např. 2,5-násobek délky vozidla), která však musí být taková, aby vozidlo projelo kolem celého parkovacího místa a zastavilo vedle auta před parkovacím místem. Během tohoto kroku je parkovací místo změřeno pomocí postranního senzoru vzdálenosti.

2. Údaje ze senzorů jsou vhodně zpracovány k výpočtu odhadu rozměrů potenciálního parkovacího místa. Jelikož vozidlo při každém kroku posílá jak stavy senzorů, tak stavy enkodérů, víme, jak jsou od sebe jednotlivá čtení vzdálenosti daleko, toto nám poskytuje dostatečný dvojrozměrný obraz.

3. Údaje o parkovacím místu jsou porovnány se zjištěnými minimálními rozměry parkovacího místa pro dané vozidlo. V případě, že se vozidlo do parkovacího místa vejde, algoritmus dále pokračuje, v opačném případě hlásí neúspěch.

4. Je vypočítána délka prvotního rovného úseku S^* a následujících oblouků R^- a L^-.

5. Vozidlo provede uvedené úseky. Kdyby vozidlo při provádění jakéhokoliv z úseků zahlásilo neúspěch, například z důvodu prevence kolize či kvůli příliš velké chybě, celý algoritmus

v tomto bodě skončí a zahlásí neúspěch. V opačném případě, pokud se všechny tři úseky povedou, vozidlo by mělo být zaparkované tak, aby ze svého parkovacího místa nevyčnívalo a je velice blízko auta za ním.

6. Volitelně, vozidlo nakonec provede vyrovnávací pohyb S^+, který pomocí údajů z předních a zadních senzorů vyrovná vzdálenost mezi vozidlem před a za naším autem.

2.3.2 Parkovací trajektorie

V této části geometricky popíšu samotnou parkovací trajektorii a odvodím výpočet parametrů jejích částí. Budu vycházet z obrázku 2.4[1] Délka úsečky IJ je vzdálenost středu auta od jeho pozice po zaparkování. Toto je kromě poloměru otáčení náš jediný výchozí rozměr, který dostaneme drobnými úpravami měření ze senzorů vzdálenosti. Přímka určená body I a M je ta přímka, po které se bude vozidlo pohybovat při prvotním pohybu S^*. Délka úsečky IM, které budu říkat délka obloukové fáze, nám umožní zjistit délku prvotního pohybu S^* (tato hodnota také bude muset být při implementaci použita nepřímo – auto se po změření parkovacího místa nenachází v bodě I, ale někde okolo bodu M). Kružnice se středem v bodě S_2 je kružnice pro pohyb popisuje úsek R^-, který probíhá od bodu M do bodu P. Kružnice se středem v bodě S_1 popisuje úsek L^-, který probíhá od bodu P do bodu J.

Jako první je nutné odvodit úhel α, ten získáme pomocí následující rovnice. Úhel lze vyjádřit pomocí známých rozměrů $|IJ|$ (vzdálenost od parkovacího místa přečtená ze senzoru) a R (poloměr otáčení):

$$\alpha = \operatorname{acos}(\frac{|S_1L|}{|S_1P|}) = \operatorname{acos}(\frac{R - \frac{|IJ|}{2}}{R}) \tag{2.1}$$

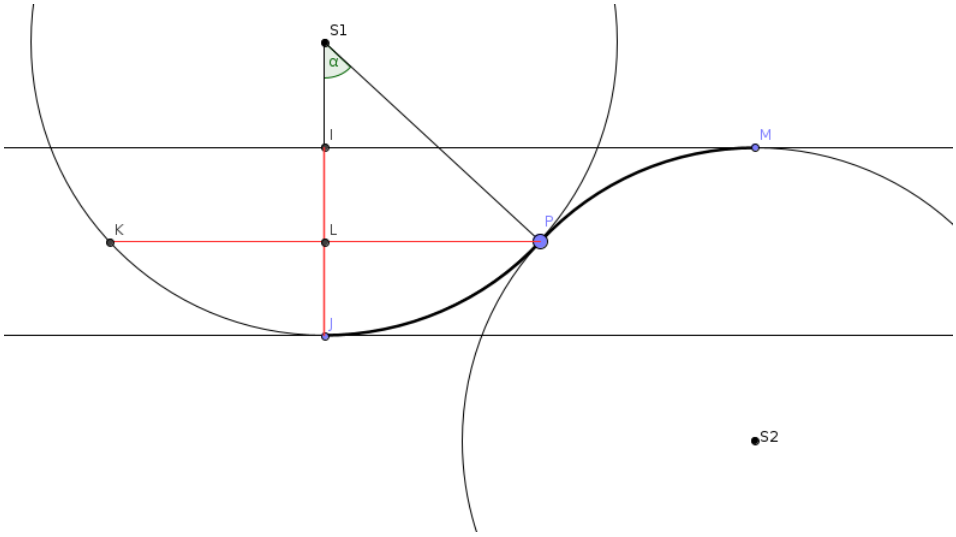
Dále zjistíme délku jednoho kruhového oblouku, nazveme ji l:

$$l = \alpha R \tag{2.2}$$

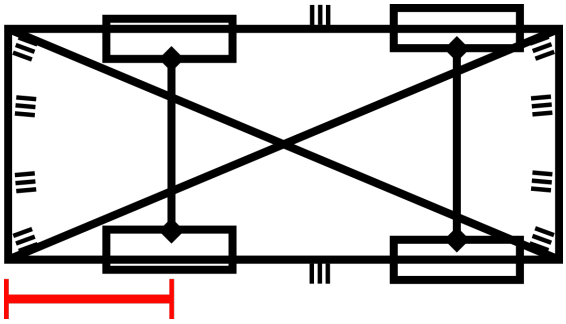
Nakonec zjistíme šířku obloukové fáze, nazveme ji w:

$$w = |IM| = |KP| = 2|LP| = 2\sqrt{|S_1P|^2 - |S_1L|^2} = 2\sqrt{(R - \frac{|IJ|}{2})^2 - R^2} \tag{2.3}$$

[1] Příslušný soubor pro program Geogebra je v příloze. V něm je možné posouvat bod dotyku kružnic a sledovat, jak se tím změní tvar trajektorie.



Obrázek 2.4 Odvození trajektorie

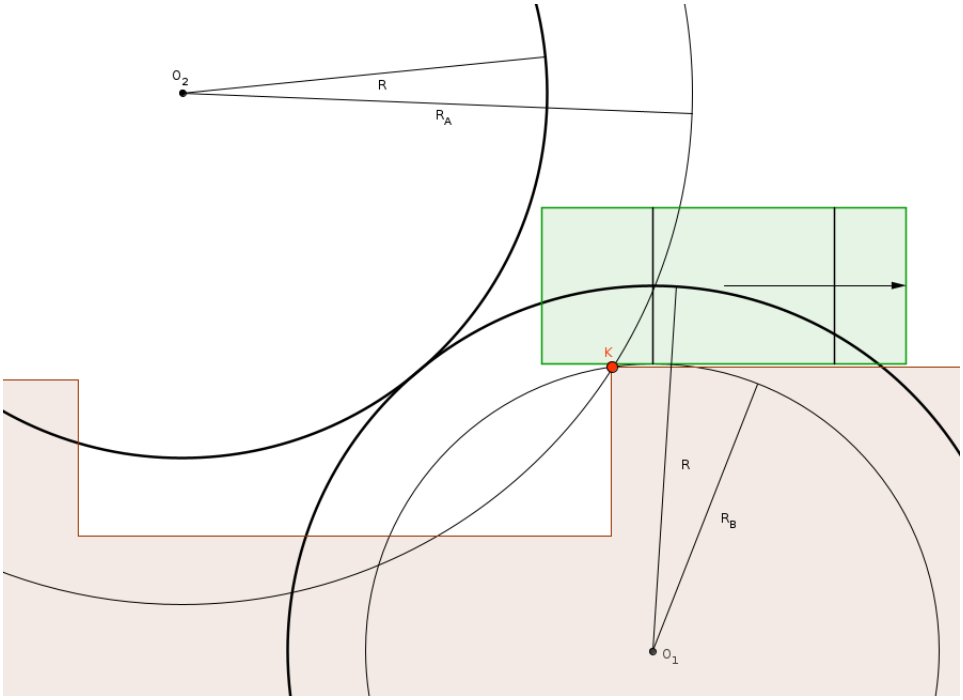


Obrázek 2.5 Možné kolize při couvání

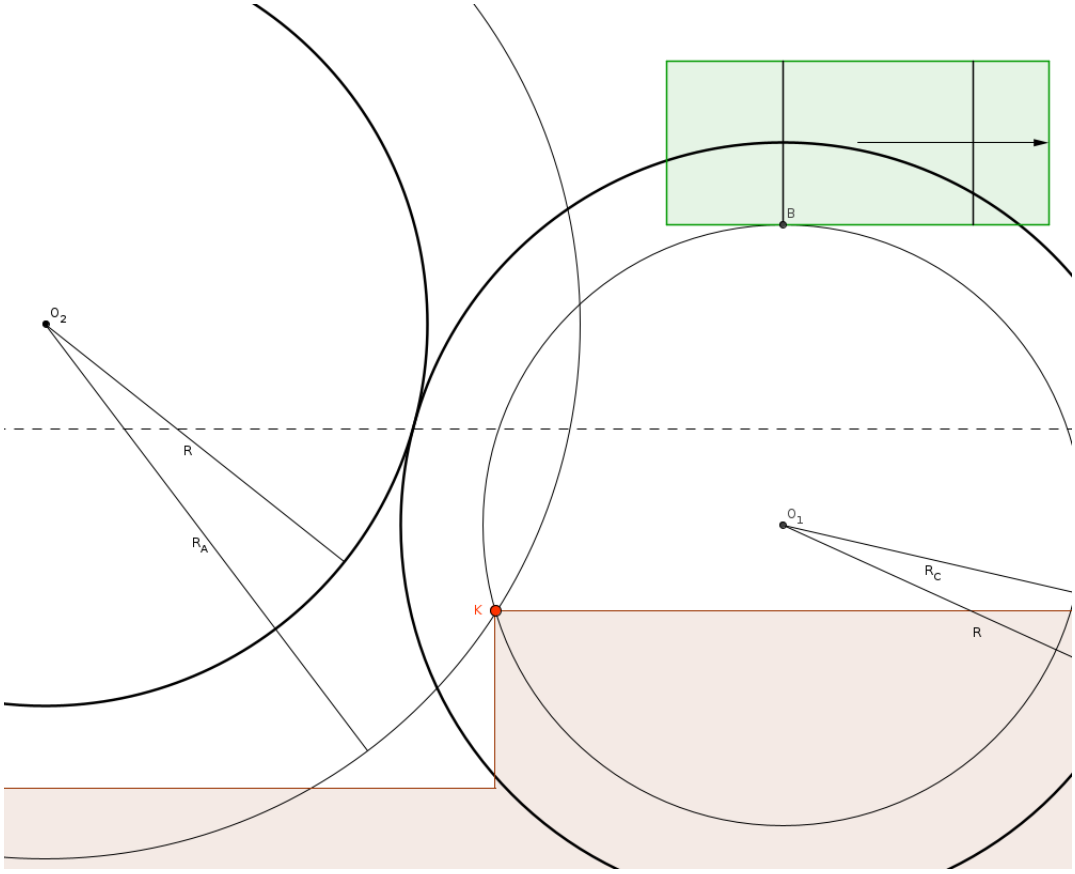
Při rozkreslování jsem identifikoval další možné místo kolize – ukázalo se, že při couvání nehrozí kolize jen na pravém zadním rohu, jak je na obrázku 1.10 z teoretické části. Kolize v tomto případě hrozí po celé úsečce mezi pravým zadním rohem a koncem zadní nápravy (kolem ní se totiž vozidlo otáčí, dále tedy kolize nehrozí), jak ilustruji na obrázku 2.5. Samotný roh ošetřený je, je třeba tedy ošetřit možné kolize s pravým koncem zadní nápravy. Kružnice otáčení auta kolem tohoto bodu protíná bod kolize ve dvou možných situacích, které dále popíšu. Obě předpokládají, že vozidlo již vykonalo správně úsek S^* a dále proběhnou jen oblouky couvání. Bod kolize je označen K. Poloměry jsou označeny stejně jako v teoretické části.

Situace na obrázku 2.6 popisuje situaci, když auto zahájí manévr R^- příliš blízko parkovacího místa. Tato vzdálenost je teoreticky téměř nulová, pro praktické účely bude však nastavena na vyšší hodnotu, například z toho důvodu, že auto téměř jistě nebude s překážkami vyrovnané naprosto rovnoběžně. Podobné tolerance budou pro jistotu z důvodu bezpečnosti zavedeny pro všechny parkovací rozměry napříč celou prací.

Situace na obrázku 2.7 popisuje druhý možný bod, kde kružnice pravého okraje zadní nápravy protne bod K. K tomuto protnutí však dochází v části kružnice, která neproběhne – od přerušované čáry níže se vozidlo pohybuje po levé kružnici. Ke kolizi tedy nemůže dojít a oba body jsou ošetřeny.



Obrázek 2.6 Kolize když auto začíná příliš blízko. Vozidlo je umístěno na teoretické minimální vzdálenosti od parkovacího místa.



Obrázek 2.7 Kolize když auto začíná příliš daleko

Kapitola 3
Praktická část

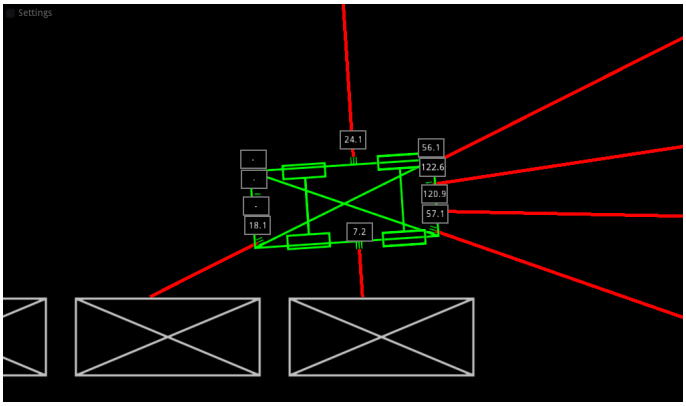
V této kapitole se věnuji samotné implementaci na základě poznatků z předchozích dvou kapitol. Implementoval jsem simulátor, terminál, parkovací algoritmus a několik podpůrných funkcionalit.

3.1 Simulátor

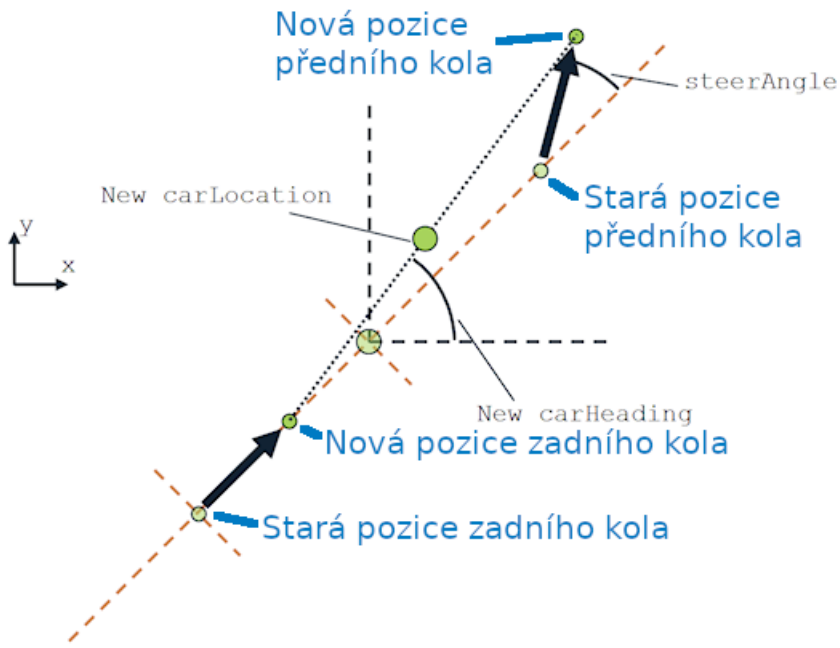
Pro snadnější testování všech funkcionalit jsem v enginu Godot vytvořil simulátor, který se dokáže spojit s pythonovými skripty a replikuje chování reálného modelu, včetně enkodérů a senzorů. Ukázky zdrojového kódu uvádím v jazyce enginu zvaném GDScript, což je jazyk syntakticky velmi podobný Pythonu.

3.1.1 Kinematika

Vozidlo se řídí jednoduchým modelem kinematiky [9], ve kterém je vnitřně implementováno jako jednostopé vozidlo podle obrázku 3.2. Spočítá se nová pozice zadního kola podle současného směru pohybu a nová pozice předního kola podle současného směru pohybu a natočení kol. Vektory pozic kol se od sebe odečtou a normalizací výsledku je získán nový směr vozidla. Jediná síla působící proti pohybu vozidla je tření. Relevantní okomentované výňatky z kódu jsou ve výpisu 3.1.



Obrázek 3.1 Simulátor



Obrázek 3.2 Kinematika simulátoru (Převzato z: [9])

Výpis kódu 3.1 Kinematika simulátoru v kódu

```
func _physics_process(delta): #Delta je cas od posledniho snimku simulace
    ...
    get_input()
    ...
    calculate_steering(delta)
    #^Na zaklade vstupu se vypocita nove natoceni vozidla.
    apply_friction(delta)
    #^Vozidlo je zpomaleno trenim
    ...
    #Nasleduje samotny pohyb vozidla, detekce kolizi atd.
    #To zajistuje samostatna funkce, kterou poskytuje engine.

func apply_friction(delta):
    ...
    #Na zacatku je vypocet friction_force. Ta je vetsinou
    #konstantni, pri aktivnim brzdeni je dvojnasobna
    #a pri velice nizke rychlosti vozidlo zastavi uplne,
    #aby simulace pusobila plynule.
    acceleration -= friction_force

func calculate_steering(delta):
    ...
    rear_wheel += velocity * delta
    #^Zadni kolo jede rovne
    front_wheel += velocity.rotated(steer_angle) * delta
    #^Predni kolo jede podle uhlu natoceni kol
    var new_heading = (front_wheel - rear_wheel).normalized()
    #^Rozdil vektoru a normalizace urci novy smer.
    ...
```

3.1.2 Funkcionality

Simulátor obsahuje několik vedlejších funkcionalit pro lepší použitelnost.

Zaprvé je možné auto ovládat klávesnicí, šipky nahoru a dolů ovládají motor a šipky doleva a doprava ovládají natočení kol. Pokud není aktivní žádný impulz pro motor, je automaticky spuštěno aktivní brzdění. Režim řízení z klávesnice je dočasně zablokován, pokud právě probíhá komunikace s terminálem.

Pozici auta je možné manuálně nastavit pomocí myši – stisknutí prostředního tlačítka myši či mezerníku umístí střed modelu na pozici kurzoru, pravé tlačítko myši otočí model ke kurzoru, pokud je přitom stisknuta klávesa Shift, je úhel modelu při tomto otáčení uzamknut na násobky 45 stupňů.

Je možné vizualizovat hodnoty přečtené senzory vzdálenosti. Na příslušný senzor lze vypisovat právě přečtenou hodnotu, lze rovněž z každého senzoru vysílat červený paprsek, pomocí kterého je vidět, o jaký povrch se paprsek senzoru odrazil. Tyto vizualizace se dají povolit či vypnout v panelu nastavení v levém horním rohu simulace. Obě tyto vizualizace je možné vidět na obrázku 3.1.

V panelu nastavení se dá rovněž určit chybovost každého měření vzdálenosti v procentech. Tím je možné zkoušet, nakolik jsou navržené algoritmy odolné vůči chybám, které v reálném modelu existují.

3.1.3 Spojení s terminálem

Spojení s pythonovým software probíhá přes websocket. Na autě v simulaci od momentu spuštění běží server[1], podle výchozího nastavení na portu 9123. Po připojení klienta (terminálu) se vynulují hodnoty virtuálních enkodérů, jako když se restartuje reálný model.

Simulátor byl implementován tak, aby s terminálem komunikoval pomocí stejných dat, jako firmware reálného modelu. S terminálem si tedy posílá stejnou datovou strukturu, jako ve výpisu kódu 1.2. Terminál simulátoru nastavuje PWM motorů a natočení kol, simulátor zpět posílá stav vozidla. Nastavení řídících hodnot a následné zabalení a poslání dat do terminálu ilustruji ve výpisu 3.2. Před voláním této funkce probíhá příjem binárních dat z websocketu, jejich příjem a rozbalení jsou implementováno analogicky.

[1] Websocketový server běží na *každém* autě v simulaci. Jednoduchými úpravami simulace, například naklonováním objektu auta a přiřazeným různého portu každému autu, by šlo simulovat chování více aut pohromadě v rámci jedné simulace.

Výpis kódu 3.2 Funkce dálkového řízení v simulátoru

```
func rc_drive(engine_input, steering):
        rc_engine_input = engine_input
        rc_steering = steering
        rc_active = true
        #^Aktivuje se rezim dalkoveho rizeni
        # Tim se zablokuje ovladani z klavesnice
        ...
        var output := StreamPeerBuffer.new()
        #^Vytvori se buffer na binarni data
        output.put_8(76)  #L
        #^Prikaz "L" z firmwaru ocekava na
        # zacatku znak "L" pro identifikaci.
        output.put_u32(Time.get_ticks_usec())
        #^Soucasny cas
        output.put_32(floor(elapsed))
        output.put_32(floor(elapsed))
        #^Hodnoty enkoderu
        # Realny model ma enkoder na obou kolech,
        # data proto posilame dvakrat.
        for s in sensors:
                output.put_u16(s.collision_dist * 10) #sensors
                #^Cteni ze vsech senzoru.
        server.get_peer(id).put_packet(output)
        #^Data posleme po socketu do terminalu.
```

3.2 Terminál

V návaznosti na existující pythonové API jsem vytvořil terminál, který umožňuje snadnou komunikaci s vozidlem. Implementoval jsem příkazy pro ovládání, kalibraci i parkování.

3.2.1 Připojení

Terminál se po svém spuštění automaticky pokusí připojit na výchozí sériové zařízení pro hardwarový model či výchozí websocketový port pro simulátor. V případě neúspěchu je tato skutečnost uživateli oznámena a je poučen jak se připojit ručně. Jelikož komunikace s hardwarem a simulací probíhá přes rozdílné protokoly, po úspěšném připojení jsou podle příslušného protokolu nastaveny funkce, které se budou při komunikaci volat. Toto ilustruji na výpisu kódu 3.3. Jako základ připojování jsem použil Python API od Davida Ondruška, které jsem upravil. Výchozí sériové zařízení a port je možné změnit v konfiguračním souboru.

3.2.2 Příkazy

Implementoval jsem téměř dvě desítky příkazů pro připojování, řízení, kalibraci a parkování.

Snazší zadávání příkazů umožňuje zabudovaný modul `readline`, který při zadávání příkazů zpřístupňuje při čtení vstupu některé funkcionality, na které je uživatel zvyklý z běžného shellu, například použití šipek pro navigaci v rámci příkazu či pro procházení historie příkazů.

Následuje výčet příkazů a jejich stručný popis. Případný název v hranatých závorkách uvádí alias příkazu pro kratší zadávání. Případné parametry funkcí jsou vyznačeny kurzívou.

- `exit` – Ukončí sériové/socketové spojení a uzavře terminál. Terminál je rovněž možné uzavřít vysláním signálu SIGINT, typicky zkratkou Ctrl+C.

Výpis kódu 3.3 Výběr protokolu v terminálu

```
async def connect_socket(path):
    state.socket = await websockets.connect(path)
    ...
    readf = socket_read
    writef = socket_write
    #^Po uspesnem pripojeni komunikujeme
    # pres socket.
async def connect_serial(path):
    port = serial.Serial(path, 500000, timeout = 0.1)
    ...
    readf = escaped_readline
    writef = escaped_write
    #^Po uspesnem pripojeni komunikujeme
    # pres seriovou linku.
async def L_command_and_response(port, left_pwm, ...):-
    line = struct.pack(...)
    await writef(port, line)
    response = await readf(port)
    #^Komunikace pak probiha pomoci vybranych funkci
    ...
```

- `help` – Vypíše seznam příkazů strukturovaných dle kategorie s příklady užití.
- `autoconnect [ac]` – Pokusí se znovu připojit na výchozí sériové zařízení či výchozí socketový port a podle typu připojení se nastaví komunikační funkce.
- `sercon` *device* – Pokusí se připojit na sériové zařízení *device*.
- `soccon` *port* – Pokusí se připojit na port *port*.
- `drive [d]` *distance steering* – Ujede *distance* centimetrů pod úhlem *steering* s použitím algoritmu pro řízení ujeté vzdálenosti.
- `direct_control [dc]` – Aktivuje režim přímého řízení. Více v samostatné podsekci.
- `scan` – Projede kolem potenciálního parkovacího místa. V případě úspěchu vypíše posloupnost příkazů pro provedení parkování.
- `park` – Projede kolem potenciálního parkovacího místa. V případě úspěchu zaparkuje a vyrovná vzdálenosti před a za autem.
- `ssz`, `fbd`... – Kalibrační příkazy. Popis v samostatné sekci.
- `wheels`, `steering`... – Příkazy pro ovládání auta na té nejnižší úrovni bez jakékoliv kontroly ujeté vzdálenosti či prevence kolizí.

3.2.3 Struktura příkazů

Při tvorbě terminálu jsem kladl důraz na jeho rozšiřitelnost. Okomentovaná ukázka je na výpisu kódu 3.4. Příkazy jsou organizovány do pythonového slovníku `commands`, kde klíčem je samotná volaná funkce a hodnotou je pomocná struktura, která udává, jaký název (či názvy) bude příkaz mít a kolik bude mít parametrů. Pro přidání nového příkazu tedy pro něj stačí přidat záznam do struktury `commands`. Tato struktura je po spuštění terminálu ještě zpracována do jiné struktury pro volání příkazů ve funkci `prepare_commands`. Výsledná struktura `command_names` má již jako

Výpis kódu 3.4 Struktura příkazů

```
def prepare_commands():
    ret = {}
    for cmd in commands:
        for alias in commands[cmd][0]:
            ret[alias] = cmd
            #^Nova struktura ma jako klic alias,
            # jako hodnotu cely prikaz.
    return(ret)
async def terminal():
    command_names = prepare_commands()
    #^Na zacatku prikazy zpracujeme.
    ... #Od uzivatele zjistime
    ... #cmd_name a cmd_args.
    if cmd_name in command_names:
    #^Prikaz nalezen ve strukture
        command = command_names[cmd_name]
        #^Zjistime volanou funkci.
        argc = commands[command][1]
        #^A pocet parametru.
        if(argc != len(cmd_args)):
            print("E: Incorrect argument count.")
            continue
        try:
            command(*cmd_args)
            #^Nalezeny prikaz zavolame s argumenty
            # od uzivatele.
        ... #Osetrovani chyb.
    else:
    #^Prikaz nenalezen.
        print("E: Unknown command.")
    ...
commands = { #Struktura prikazu
        #Format prikazu:
        #nazev_funkce      : [ ["alias1", "alias2"], pocet_parametru]
        parking.park       : [ ["park"], 0],
        control.drive_dist : [ ["drive","d"], 2],
        ...
```

klíče aliasy příkazů. Když uživatel zadá příkaz, terminál ověří, zda-li existuje v `command_names` klíč s jeho názvem. Pokud odpovídá i počet parametrů, funkce je zavolána. V opačném případě (neexistující příkaz, špatný počet parametrů) hlásí terminál chybu a čeká na další příkaz.

Do budoucna by tento systém šel vylepšit přidáním typových kontrol pro parametry. V současném stavu jsou parametry předávány tak, jak je zadal do terminálu uživatel, tedy typu `string`. Volaná funkce si je převádí na příslušný typ sama. Lepší možnost by byla místo uvádění počtu parametrů uvádět výčet jejích typů, kontrolu i převody by pak řešil terminál (na jednom místě) a do funkcí by předával již převedené parametry výsledných typů. Další chybějící funkcionalita je podpora výchozích hodnot parametrů (v současném stavu musí mít příkaz vždy stejný počet parametrů, uživatel musí vždy vypsat všechny).

Název	Účel
python_api	Komunikace s modelem a simulací
control	Řízení ujeté vzdálenosti, brzdění
parking	Parkovací funkce
calibration	Funkce pro kalibraci řízení
state	Udržování stavu, čtení a zápis konfigurace
util	Pomocné matematické převody

Tabulka 3.1 Moduly terminálu

Výpis kódu 3.5 Konfigurační soubor

```
[COMMON]
#Connection
default_serial_device = /dev/ttyUSB0
default_simulation_port = 9123
#Dimensions
car_length = 28.0
...
[SIMULATOR]
#Don't clamp angle on sim
clamp_angle = False
...
[MODEL]
steering_zeropoint = 9
...
```

3.2.4 Moduly

Adresář terminálu obsahuje kromě souboru pro spuštění terminálu několik modulů, které implementují příkazy a poskytují další potřebné funkce. Jejich účely jsou popsány v tabulce 3.1

3.2.5 Konfigurační soubor

V kořenovém adresáři terminálu je umístěn soubor `config.ini`, ve kterém je možné nastavit parametry vozidla a chování algoritmů. Výňatek je ve výpisu kódu 3.5. Záznamy jsou ve formě párů klíče a hodnoty, přičemž datový typ hodnoty není omezen. Záznamy jsou rozděleny do tří sekcí – `common`, `simulator` a `model`. Společné (výchozí) hodnoty parametrů jsou v sekci `common`, podle současného typu připojení ustupují záznamům v sekcích `simulator` a `model`, které mají přednost. Konfigurační soubor upravuje jak uživatel ručně, tak sám terminál při volání některých funkcí (konkrétně kalibračních). Položky jsou od sebe ještě volitelně odděleny komentáři, význam samotných položek je definován v příslušných sekcích této práce podle kategorie položek (např. rozměry, kalibrace).

Pro práci s konfiguračním souborem jsem zvolil modul `configparser`. Jak s ním pracuji je možné vidět na výpisu kódu 3.6. Na začátku je konfigurační soubor přečten a zpracován do interní struktury modulu `configparser`. Ve funkci `prepare_cfg` jsou záznamy přesunuty z této interní struktury do obyčejného slovníku podle současného typu připojení. Z tohoto slovníku jsou potom ještě dále roztříděny do sedmi dalších slovníků podle typu pro snazší použití (není ve výpisu). Modul si drží veškerá data jako typ `string` a neinterpretuje datové typy, převod provádím sám ve funkci `parse_cfg_value` podle posloupnosti priorit definované v kódu. Funkce pro změnu hodnoty položky hodnotu změní v interní struktuře a pak přepíše konfigurační soubor na disku pomocí poskytnuté funkce.

Výpis kódu 3.6 Čtení a zápis konfigurace

```
def parse_cfg_value(s):
#^Vse v konfiguracnim souboru je string.
# Tato funkce se pokusi hodnotu interpretovat
# int > float > bool > string
    if(not s):
        return
    try:    #Int
        return int(s)
    except ValueError:
        try:    #Float
            return float(s)
        except ValueError:
            if(s == "True"): #Bool
                return True
            elif(s == "False"):
                return False
            else: #String
                return s
...
parser = configparser.ConfigParser(...)  #Interni struktura modulu
parser.read(cfg_file) #Precteme konfiguracni soubor
def prepare_cfg():
    if ("COMMON" in parser.sections()):
        for key in parser["COMMON"]:
            cfg[key] = parse_cfg_value(parser["COMMON"][key])
    #^Vychozi hodnoty ze sekce common
    if(serial):
    #^Pokud jsme v soucasnosti pripoojeni na seriovou linku,
    # precteme sekci model a pripadne shody klicu prepiseme.
        if ("MODEL" in parser.sections()):
            for key in parser["MODEL"]:
                cfg[key] = parse_cfg_value(parser["MODEL"][key])
        ...#Analogicky pro simulator
def update_cfg_field(section, field, val):
#^Upravime jednu hodnotu a zapiseme na disk.
    parser[section][field] = val
    update_cfg() #Zapis na disk
```

Pokud při spuštění terminálu není konfigurační soubor nalezen, je vytvořen nový s výchozími parametry. Toto je užitečné při experimentování, pro obnovu konfiguračního souboru do původního stavu ho stačí jednoduše smazat, v případě hardwarového modelu je však třeba znovu provést kalibraci.

3.3 Podpůrné funkcionality

3.3.1 Kalibrace

Implementoval jsem systém interaktivní kalibrace vozidla, která je nezbytná k jeho přesnému řízení. Na začátek vypíšu používané příkazy, které dále vysvětlím:

- `ssz` *angle* – Set steering zeropoint. Nastaví nulový bod kol na *angle* a zakmitá servem.
- `fbd` *dist* – Find brake distance. Po vzdálenost *dist* se rozjíždí a zjistí brzdnou dráhu při dané rychlosti.
- `fbp` – Find brake parameters. Spočítá parametry regresní přímky podle naměřených dat a vypíše je.
- `sbp` a b – Set braking parameters. Nastaví odhad brzdné dráhy na přímku $av + b$.
- `rtd` – Reset training data. Vymaže množinu naměřených dat brzdných drah.

Jako první je příkaz `ssz`, který slouží k nastavení nulového bodu kol. Serva předních kol na modelech nejsou typicky přesně vycentrována a tento příkaz slouží jako softwarové řešení problému. Po zavolání příkazu jsou kola nastavena na maximální možný úhel a krátce poté na nový nulový úhel. Toto se ukázalo nutné z toho důvodu, že servo není dostatečně silné na to, aby se dokázalo natočit o velmi malé úhly. Příkaz jsem navrhl k opakovanému volání s různými hodnotami, dokud nebudou přední kola rovnoběžná se zadními. Tento proces doporučuji dělat s modelem otočeným na střechu z důvodu možného opotřebení kol. Nový nulový úhel se při každém zavolání příkazu zapíše do konfiguračního souboru do záznamu `steering_zeropoint`. V simulátoru tento krok kalibrace není nutný.

Další příkazy slouží ke kalibraci odhadu brzdné dráhy. Základem je příkaz `fbd`, který se po danou vzdálenost rozjíždí, začne brzdit a po zastavení vypíše brzdnou dráhu a rychlost při které začal brzdit. Výsledné páry dat se průběžně ukládají do pomocné struktury, po každém zavolání se vypíše, kolik obsahuje struktura vzorků. Každý vzorek je třeba kontrolovat, vzácně je naměřená rychlost nerealisticky vysoká. Pro případné zahození naměřených vzorků je dostupný příkaz `rtd`. Po naměření dostatečného počtu vzorků (alespoň několik desítek) při různých zadaných vzdálenostech rozjíždění je množina vzorků připravena na statistické zpracování dle analytické části, kde byla zvolena lineární regrese.

Tvorba tohoto odhadu se vykoná zavoláním příkazu `fbp`, jehož implementace ukazuji na výpisu 3.7. Jako naprosté minimum vyžaduje, aby byly naměřeny alespoň dva vzorky, v opačném případě hlásí chybu. Implementaci lineární regrese poskytuje modul `scikit-learn`. Položky z párů naměřených dat jsou rozděleny do samostatných polí a těmito poli je natrénován model. Na konci jsou vypsány parametry vypočítané přímky a uživateli je oznámeno, jak parametry aplikovat. Nejsou aplikovány automaticky, aby mohl uživatel odchytit alespoň ty na první pohled nesmyslné odhady, například záporný lineární člen či velice vysoký absolutní člen. Samotná aplikace parametrů se vykonává příkazem `sbp`.

Naměřená data jsem potřeboval několikrát zpracovat i ručně externími programy. Vytvořil jsem jednoduchý shellový skript `csv_util.sh`, který z lidsky čitelných výpisů z terminálu vytvoří CSV seznam pro strojové zpracování. Příklad je na výpisu 3.8, na vstupu jsou data, jak je vypsal terminál, na výstupu je CSV.

Výpis kódu 3.7 Parametry odhadu brzdné dráhy

```
from sklearn.linear_model import LinearRegression
...
async def find_brake_parameters():
    if(len(brake_data) < 2):
        print("E: need at least 2 samples...")
        print("Run fbd command a few more times...")
        return
    model = LinearRegression()
    velocities = []
    distances = []
    for pair in brake_data:
    #^Rozdeli pary z namerenych dat
        velocities.append(pair[0])
        distances.append(pair[1])
    model.fit([[val] for val in velocities], distances)
    #^Trenovani modelu
    ...#Vypis vypocitanych parametru
```

Výpis kódu 3.8 Skript pro data brzdné dráhy

```
$ cat brake_data
> fbd 1
Peak velocity: 736.1269261820352
Brake distance: 5.353580253676694
Training dataset size: 1
> fbd 10
Peak velocity: 1583.1238604029963
Brake distance: 11.087649799012825
Training dataset size: 2
> fbd 15
Peak velocity: 1851.5173631547968
Brake distance: 13.025915352107614
Training dataset size: 3
>
$ cat brake_data | ./csv_util.sh
736.1269261820352,5.353580253676694
1583.1238604029963,11.087649799012825
1851.5173631547968,13.025915352107614
```

Výpis kódu 3.9 Přímé ovládání pomocí knihovny `keyboard`

```
while True:
    steering = 0
    engine = 0
    #Na zacatku vynulujeme hodnoty
    if keyboard.is_pressed('left'):
        steering = -steering_angle
    ...
    if keyboard.is_pressed('up'):
        engine = 1
    #^Ty se nastavi, pokud je prislusna
    # klavesa v danem cyklu drzena
    await drive(engine,steering)
    time.sleep(sampling_period)
```

3.3.2 Přímé řízení

Pokusil jsem se vytvořit příkaz pro přímé ovládání vozidla, který jsem nazval `dc`. Kýženým výsledkem byla možnost ovládat model přímo pomocí šipek klávesnice, což by z něj dočasně udělalo klasické auto na dálkové ovládání. Držení šipek nahoru a dolů by ovládalo motor a šipky doleva a doprava servo předních kol. Tato funkcionalita se mi pro účely testování jevila jako velice užitečná alternativa oproti ovládání pomocí psaní sekvencí příkazů k pohybu či ručnímu přesouvání vozidla.

Pro implementaci tohoto příkazu jsem zvolil nejdříve modul `keyboard`, ukázka je ve výpisu 3.9. V cyklu se zkontroluje stav každé z ovládacích kláves, proměnné se nastaví na příslušné hodnoty a na konci každého cyklu je vozidlu vyslán řídící příkaz s parametry dle proměnných. V simulátoru tato implementace fungovala bezchybně a ovládání přes tento příkaz bylo téměř identické s klávesnicovým ovládáním zabudovaným v simulátoru.

Při použití hardwarového modelu ovšem terminál běží na Raspberry Pi modelu a uživatel zadává příkazy skrze SSH spojení z vlastního zařízení. Jak jsem zjistil, nejen že není použití modulu `keyboard` při připojení přes SSH možné, možné při tomto připojení není ani celé mé očekávané chování [12]. Informace o puštění klávesy se přes SSH vůbec nepředává a v daný moment není možné držet více než jednu klávesu.

Očekávaného chování by bylo možné dosáhnout různými způsoby. Pokud bychom trvali na použití SSH, bylo by možné se připojovat s parametrem `-X` pro předávání X11. Pak by bylo možné spouštět na Raspberry Pi grafické programy a šlo by vytvořit okno, jehož účelem by bylo zaznamenávat informace z klávesnice. S Pythonem by toto šlo například za použití knihovny `PyGame`. Další možnost je nepoužívat SSH a místo toho mít na Raspberry Pi spuštěný server, který by s klientem na zařízení uživatele komunikoval nějakým vlastním protokolem. Implementace těchto nápadů je mimo rozsah této práce.

V současnosti zůstává příkaz bez funkční implementace. Implementaci pomocí modulu `keyboard` jsem ponechal ve zdrojovém kódu zakomentovanou, je možné ji odkomentovat a vyzkoušet v simulátoru.

3.3.3 Brzdění

Pro model jsem implementoval podporu aktivního brzdění, což vyžadovalo zásah do firmwaru, který je na výpisu kódu 3.10. Podle datasheetu H-můstku [3] je aktivní brzdění zapnuto, když se shodují řídící signály pro směr motorů. Rozhodl jsem se, že brzdění bude aktivováno, pokud bude vozidlu zadána nulová rychlost. PWM brzdění je pak nastavena na maximum a není možné ji regulovat.

Výpis kódu 3.10 Aktivní brzdění

```
void MotorEncoderPid::run(int power){
    int dir;
    if(power == 0){
    //^Pokud je zadana rychlost nulova,
    // brzime maximalne.
      pwm.setChannelPWM(en, 4095);
      dir = 0;
    }
    else{
      dir = (power > 0 ? 1 : -1);
      pwm.setChannelPWM(en, power*dir);
      //Pohyb vpred ci vzad
    }
    if(power != lastdir) {
        lastdir = dir;
        if (dir == 1){ //Smer vpred
            pwm.setChannelOff(in1);
            pwm.setChannelOn(in2);
        } else if (dir == -1){ //Smer vzad
            pwm.setChannelOn(in1);
            pwm.setChannelOff(in2);
        } else if (dir == 0){ //Brzdeni
            pwm.setChannelOn(in1);
            pwm.setChannelOn(in2);
        }
    }
}
```

3.3.4 Natočení kol

Implementoval jsem převod úhlu kol podle rovnic 1.2 a 1.3 z teoretické části. Jestli tento převod proběhne je možné nastavit v konfiguračním souboru záznamem `clamp_angle`. Vypočítané převody jsou tabelovány do slovníku, aby je stačilo pro daný úhel počítat jen jednou.

Maximální úhel kol jsem omezil, bez omezení přesahuje maximální úhel serva i po převodu oboustranně 40 stupňů, což není realistické. Úhel jsem omezil na 30 stupňů, což je hodnota, kolem které se pohybují maximální úhly kol běžných osobních automobilů. Jiné hodnoty je možné nastavit záznamem `steering_angle`.

3.3.5 Řízení ujeté vzdálenosti

Řízení ujeté vzdálenosti volá buď uživatel přímo příkazem `drive`, nebo ho volají jiné funkce, například `park`. Implementoval jsem ho podle vývojového diagramu 2.2 z analytické části. Zde obecně implementaci popíšu zjednodušeným kódem 3.11 a potom vysvětlím některá implementační specifika.

Příkaz se volá s parametry vzdálenosti k ujetí a úhlem natočení kol. Na začátku funkce `drive_dist` zjistí doposud ujetou vzdálenost modelu a vypočítá cílovou přičtením zadané vzdálenosti. Vozidlo se opakovaně pokouší této hodnoty co nejpřesněji dosáhnout, přičemž při každém pokusu vznikne nevyhnutelně nějaká chyba. Tyto chyby jsou cyklem opravovány tak, že je vozidlu zadáváno ujet opačnou hodnotu chyby. Toto probíhá, dokud neproběhl maximální počet opravných cyklů, či dokud není úsek vyhodnocen jako úspěšně vykonaný. Řízení ujeté vzdálenosti jednoho úseku je považováno za úspěšné, když se po provedení algoritmu liší ujetá a cílová vzdálenost nejhůře o konstantu definující maximální tolerovanou chybu.

Jednotlivé pokusy jsou popsány ve funkci `drive_cycle`. Funkce vyhodnotí, jakým směrem se bude vozidlo v tomto konkrétním pokusu pohybovat (je například možné, že vozidlo v prvním pokusu cíl lehce přejelo a v druhém je nutné mírně couvat). Modelu jsou opakovaně posílány signály pro řízení motorů, přičemž hodnota pro PWM je lineárně interpolována tak, že vozidlo zpomaluje, když se blíží k cíli. Po každém odeslání těchto dat obdrží funkce zpět z vozidla jeho stav. V každém průběhu vnitřního cyklu je zjištěna současná rychlost a na základě ní je odhadnuta brzdná dráha. Pokud je součet ujeté vzdálenosti a odhadu brzdné dráhy větší nebo roven cílové vzdálenosti, vozidlo začne okamžitě brzdit do úplného zastavení. Pokud byl odhad brzdné dráhy dostatečně dobrý, vozidlo zabrzdí velmi blízko cíle. Volající funkce `drive_cycle` vyhodnotí chybu a případně vyšle další opravný pokus.

Jednotlivé průběhy vnitřním cyklem jsou od sebe časově vzdáleny o konstantu, kterou nazývám vzorkovací perioda. V konfiguračním souboru je možné nastavit její převrácenou hodnotu pomocí záznamu `sampling_frequency`. Její hodnotu jsem nastavil na 60 Hz. Musí být rovna alespoň 30 Hz, aby byla zaznamenána všechna data ze senzorů vzdálenosti, které na této frekvencí pracují. Vyšší hodnotu jsem zvolil kvůli tomu, že jsem potřeboval více dat z enkodérů na motorech, při vzorkovací frekvenci 30 Hz provádělo vozidlo příliš mnoho korekcí.

Samotné posílání řídících signálů zajišťuje Python API. Funkce `L_command_and_response`, která má 5 parametrů, byla pro snadnější používání zabalena do funkce `drive`, která má jen 2 parametry. V původním příkazu je možné řídit motory odděleně a ovládat serva kamery, to však nepotřebuji. V nové funkci `drive` jsou jen dva parametry – PWM motorů a úhel kol.

Tato funkce si udržuje pole stavů vozidla (stavy senzorů apod.) v každém okamžiku jízdy po daném úseku (včetně brzdění). Data z jednotlivých okamžiků budu dále nazývat „snímky“. Snímky jsou ještě seřazeny podle ujeté vzdálenosti, aby pořadí dat v poli odpovídalo i jejich fyzickému umístění.[2] Toto bylo z výpisu kódu vynecháno. Zpracování této velké datové struktury

[2]Pokud by například vozidlo po pohybu vpřed couvalo v rámci jednoho příkazu, poslední snímek z couvání by byl v poli umístěn za posledním snímkem z pohybu vpřed. Poslední snímek z couvání ale poskytuje informaci o bodu, který se fyzicky nachází před bodem posledního snímku z pohybu vpřed, jelikož vozidlo couvalo. Proto je třeba snímky seřadit podle ujeté vzdálenosti.

Výpis kódu 3.11 Zjednodušený kód funkce pro řízení ujeté vzdálenosti

```
def drive_dist(distance, steering): #Vzdalenost v cm, uhel ve stupnich
    elapsed = get_elapsed()  #Ziska ujetou vzdalenost pomoci dat z enkoderu
    target = distance + elapsed     #Cilovy stav
    cycles = 0                      #Pocet opravnych cyklu
    elapsed = drive_cycle(distance, steering) #Jedeme poprve...
    overshoot = elapsed - target    #Zjistime chybu
    while((abs(overshoot) > TOLERATED_ERROR) and cycles < MAX_CYCLES):
    #^Pokud chyba neni zanedbatelna, cyklime se.
        elapsed = drive_cycle(-overshoot, steering)
        #^Ujedeme opacnou hodnotu chyby, tak ji opravujeme
        overshoot = elapsed - target
        #^Zjistime novou hodnotu chyby
        cycles += 1

#Prubeh jednoho cyklu (pokusu)
def drive_cycle(distance, steering):
    elapsed = get_elapsed()         #Pocatecni ujeta vzdalenost
    target = distance + elapsed     #Cilovy stav
    direction = distance > 0 ? 1 : -1      #Zjistime smer pohybu
    remaining = direction * (target - elapsed)
    while (remaining > 0):
        motor_power = lerp(remaining)
        #^Linearni interpolace PWM motoru v zavislosti
        # na zbyvajici vzdalenosti

        run_motors(power*direction, steering) #Pustime motory
        elapsed = get_elapsed() #Zjistime stav
        remaining = direction * (target - elapsed)

        velocity = get_current_velocity() #Soucasna rychlost

        brake_distance = predict_brake_distance(velocity)
        #^Odhad brzdne drahy v zavislosti na soucasne rychlosti

        if(remaining - brake_distance <= 0):
            break #Pokud je vzdalenost do cile pri soucasne rychlosti
                  #rovna odhadu brzdne drazy, zacneme okamzite brzdit,
                  #tim dobrzdime skoro presne do cile.

        time.sleep(sampling_period) #Vyckame po vzorkovaci periodu
    brake() #Brzdime dokud uplne nezastavime.
    return encoder_status #Konecny stav

def drive(engine_pwm, steering_angle): #Zabaleni funkce z Python API
    return await python_api.L_command_and_response(
        state.port,
        float(engine_power),
        float(engine_power),
        int(clamp_angle(steering - calibration["steering_zeropoint"])),
        int(0),
        int(0)
    )
```

slouží jako výchozí bod pro parkovací algoritmus.

Firmware nezná rychlost vozidla a bylo nutné ji počítat ručně na základě informací z enkodérů a časem odměřeným mezi přijatými daty. Tento údaj je dosazen do přímky odhadující brzdnou dráhu.

Napříč funkcí používám zjednodušení při zjišťování stavu optických enkodérů na motorech ve funkci `get_elapsed`. Jako ujetou vzdálenost beru jednoduše průměr levého a pravého enkodéru, to však není zcela přesné. Při zatáčení opisuje levé a pravé zadní kolo jinak velké kružnice a poloměr opisovaný středem zadní nápravy není pouhý aritmetický průměr uvedených dvou rozměrů. V simulaci toto nepřesnosti nevytváří, enkodéry simulace počítají vzdálenosti ujeté středem vozidla. V hardwarovém modelu toto nepřesnosti vytváří, při testování se však ukázaly jako zanedbatelné. Virtuální enkodér udržující přesný stav z pohledu středu vozidla by bylo nejlepší implementovat přímo na firmwaru, bylo by však nutné, aby firmware znal přesné rozměry vozidla, což současný firmware nevyžaduje.

PWM motorů je na základě zbývající vzdálenosti lineárně interpolována tak, že vozidlo zpomaluje, když se blíží k cíli. Parametry interpolace je možné nastavit v konfiguračním souboru pomocí záznamů `lerp_lo`, `lerp_hi`, `lerp_at_lo`, `lerp_at_hi`.

Maximální počet opravných cyklů je možné omezit v záznamu `max_cycles`, maximální tolerovaná chyba v záznamu `tolerated_error`.

Je možné nastavit nejmenší vzdálenost od překážky, proti které je možné se rozjet, pomocí záznamu `minimum_traversible_distance`. Vozidlo se k překážkám které jsou dle údajů ze senzorů blíže než tato hodnota odmítne rozjet bez ohledu na to, jak situaci vyhodnotí prevence kolize. Toto slouží jako dodatečná pojistka proti tomu, aby se vozidlo rozjelo přímo proti velmi blízké překážce, protože v úplně prvním snímku nemá ještě informaci o své rychlosti.

Převod mezi interními jednotkami enkodérů a centimetry není prováděn automaticky, kvůli velkému objemu dat přicházejících z vozidla (z nichž velká část není vůbec využita) jsem to považoval za nadbytečné. Převod je vykonáván jen tam, kde je potřeba, s použitím funkcí `cm_to_count` a `count_to_cm`.

Samotné posílání řídících dat do modelu jsem zabalil do cyklu, který se opakuje, dokud se odeslání nepodaří (není ve výpisu). Bylo to nutné z toho důvodu, že komunikace po sériové lince vykazuje značnou chybovost napříč všemi modely a někdy je třeba poslat ta samá řídící data vícekrát.

3.3.6 Prevence kolizí

Hardwarový model je relativně křehký, přičemž za nejohroženější považuji nekryté senzorové lišty na přední a zadní části vozidla. Abych alespoň částečné snížil riziko jejich poškození, implementoval jsem jednoduchý systém prevence kolizí, který využívá data z prostředních dvou senzorů na senzorových lištách. Příslušná funkce `is_collision_imminent` je na výpisu kódu 3.12. Funkce je použita v podmínce v hlavním cyklu funkce pro řízení vzdálenosti, s využitím znalosti odhadu brzdné dráhy je cyklus přerušen a je zavolána funkce brzdění, pokud je brzdná dráha při dané rychlosti menší nebo rovna údaji ze senzorů na straně, jejíž směrem se vozidlo pohybuje. Je zabráněno kolizím pouze při pohybu rovně, ostatní směry nebyly ošetřeny.

Je možné začít brzdit dříve, než je nutné. Toto lze nastavit pomocí konfiguračního záznamu `braking_margin`.

Tato funkce vyžaduje existující odhad brzdné dráhy a na kalibračních funkcích či na řídících funkcích nižší úrovně není využita.

Výpis kódu 3.12 Jednoduchá prevence kolizí

```
def is_collision_imminent(scan, direction, steering, brake_distance):
#^Predavame cely scan soucasneho snimku, at mame k dispozici data senzoru
    if(steering != 0):
        return False #Kolizim je zabraneno jen v rovnem smeru
    if(direction == 1):
        dist = min(scan[SENSORS][FRONT_2], scan[SENSORS][FRONT_3])
    else:
        dist = min(scan[SENSORS][REAR_2], scan[SENSORS][REAR_3])
    #^Precteme senzory ve smeru, kterym se pohybujeme.
    if (dist) >= sensor["max_sensor_value"]:
    #^Data prevysujici maximalni hodnotu senzoru nejsou verohodna
        return False
    if (dist) <= driving["minimum_traversible_distance"]:
    #^Zde je zaroven osetrena minimalni mozna vzdalenost
        return True
    return (dist <= count_to_cm(brake_distance) + safety["braking_margin"])
    #^Pokud je brzdna draha delsi nebo rovna vzdalenosti k senzorum,
    # zacneme brzdit.
```

Výpis kódu 3.13 Funkce park

```
async def park():
    scan = await drive_dist(2.5 * CAR_LENGTH, 0)
    res = parallel_space_viable(scan)
    if(res != False):
        print("Parking is possible. Parking...")
        await perform_parallel_parking(res.initial_offset, res.arc_length)
    else:
        print("Parking not possible.")
```

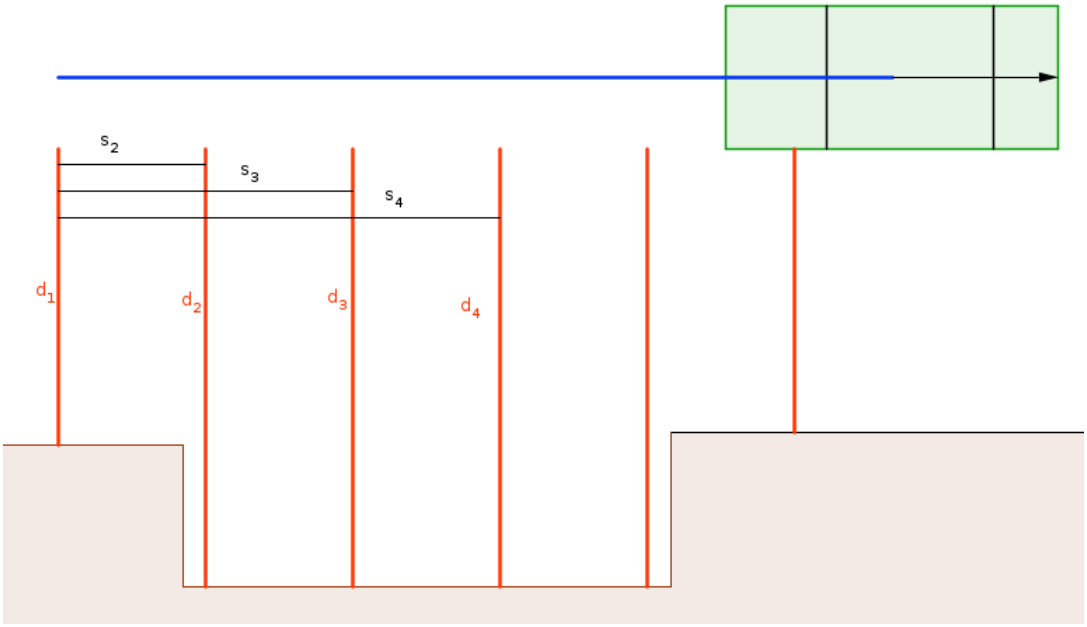
3.4 Parkování

Parkovací algoritmus jsem implementoval za použití poznatků z předchozích kapitol. Parkovací místo je vyhodnoceno, trajektorie je vypočítána a úseky jsou vykonány. Zahájit parkování je možné pomocí příkazu `park`, který parkovací místo vyhodnotí a případně zaparkuje, či pomocí příkazu `scan`, který parkovací místo jen vyhodnotí a uživateli vypíše posloupnost příkazů pro ujetí jednotlivých vypočítaných úseků.

Ve výpisu 3.13 ukazuji princip příkazu `park`, příkaz `scan` funguje analogicky. Nejdříve vozidlo ujede 2,5-násobek své délky, aby místo změřilo. Výsledkem tohoto měření je pole snímků zvané `scan`. Na základě tohoto pole je parkovací místo ve funkci `parallel_space_viable` vyhodnoceno – jsou zjištěny jeho rozměry a způsobilost pro parkování. Pokud je parkování možné, funkce vrátí strukturu se dvěma hodnotami – `initial_offset` pro prvotní rovný pohyb S^* a `arc_length` pro následné couvání R^- a L^-. Je zahlášena způsobilost parkovacího místa a na základě těchto dvou hodnot je parkování provedeno. V případě, že parkování možné není, je toto uživateli oznámeno. V takovém případě sama funkce `parallel_space_viable` ve svém průběhu oznámí, co přesně nevyhovuje – místo je příliš krátké, přílíš blízko atd.

3.4.1 Zpracovávaná data

Po spuštění parkovací funkce proběhne její první krok, tedy to, že vozidlo ujede 2,5-násobek své délky ze své výchozí pozice před parkovacím místem. Po tomto pohybu skončí za parkova-



Obrázek 3.3 Ilustrační měření

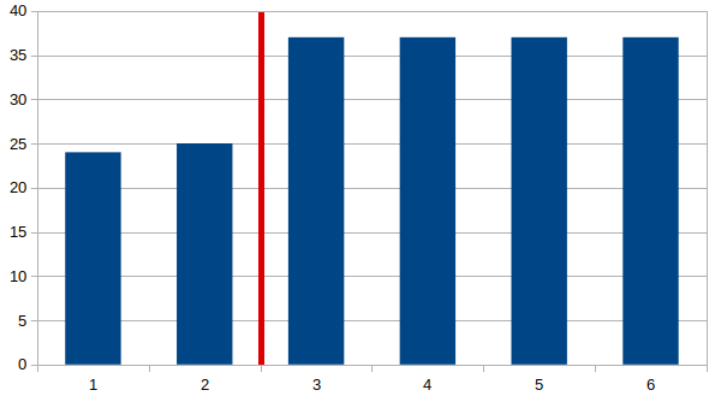
Snímek	Ujetá vzdálenost v cm	Hodnota na senzoru v cm
1	0	25
2	12	37
3	24	37
4	36	37
5	48	37
6	60	24

Tabulka 3.2 Data ilustračního měření

cím místem. V průběhu tohoto pohybu zjišťoval model pomocí senzorů informace o svém okolí s frekvencí definovanou v konfiguračním souboru jako `sampling_period`. Výsledek tohoto měření dostává algoritmus ve formě pole (čítající při současné vzorkovací periodě okolo 50 vzorků), jehož každá položka (snímek) je struktura aktuálního stavu vozidla z každého naměřeného okamžiku jízdy.

V této struktuře obsahuje každý snímek mnoho informací, pro účely zpracování parkovacího místa nás však zajímají jen dvě položky – ujetá vzdálenost a hodnota, kterou naměřil pravý postranní senzor. Tyto dvě položky jsou dostatečné pro vytvoření dvojrozměrného obrazu parkovacího místa – hodnota ze senzoru nám říká, jak je daný bod daleko a ujetá vzdálenost nám říká, jak jsou od sebe jednotlivá měření vzdálená.

Implementaci budu demonstrovat na jednoduchém fiktivním měření, které je zakresleno na obrázku 3.3. Auto ujelo 2,5-násobek své délky a naměřilo celkem 6 snímků (oproti parkování reálného modelu se tedy liší jen vzorkovací frekvence). Pro snímek i je hodnota ze senzoru označena d_i a ujetá vzdálenost podle informací z enkodérů označena s_i. Předpokládáme, že před tímto měřením model neujel žádnou vzdálenost, takže platí, že $s_1 = 0$. Měření jsem zapsal do tabulky 3.2.



Obrázek 3.4 Rozdělení naměřených dat

3.4.2 Vyhodnocení parkovacího místa

Data jsou v této fázi připravena k vyhodnocení. Jako první zjistíme, co budeme považovat za parkovací místo a co za překážku. To učiníme tak, že si z naměřených dat vybereme jen hodnoty na senzorech a ty vzestupně seřadíme. Pak nalezneme největší rozdíl mezi dvěma sousedními hodnotami. Hodnoty nalevo od tohoto dělícího místa považujeme za body náležící k překážkám, body napravo považujeme za body náležící parkovacímu místu. Toto je ilustrováno na orbázku 3.4. Naměřené hodnoty byly seřazeny, dvě hodnoty nalevo od červené dělící čáry považujeme za body náležící překážkám (např. ostatní zaparkovaná auta), hodnoty napravo považujeme za náležící samotnému parkovacímu místu.

První hodnota nalevo od rozdělení je nejvzdálenější bod překážek a první hodnota napravo od rozdělení je nejbližší bod parkovacího místa. Rozdíl těchto sousedních dvou hodnot je nejmenší možná šířka tohoto parkovacího místa. Funkce pak kvůli bezpečnosti považuje toto minimum za šířku parkovacího místa. Lze vymyslet dostatečně široká místa, která jsou tímto postupem vyhodnocena jako nedostatečná, nenastane však situace, při které by bylo nedostatečně široké místo vyhodnoceno jako dostatečné, jelikož je tento rozměr minimum. Na ilustrovaném příkladu je tedy šířka parkovacího místa vyhodnocena jako 12 cm.

Implementace tohoto rozdělení je na výpisu 3.14. Na začátku jsou ze vstupního pole `scan` vybrány pouze hodnoty z pravého senzoru do pomocného pole a toto pole je poté seřazeno. Následně jsou v cyklu porovnáváni sousedé a je nalezen maximální rozdíl. Bod napravo od rozdělení je nazván `pivot`. Pro budoucí účely je rovnou uložena vzdálenost nejbližší překážky. Maximální nalezený rozdíl je ustanoven jako šířka vozidla. Na konci je rovnou vyhodnoceno, zda-li je parkovací místo dostatečně široké.

Při reálném běhu parkovacího algoritmu by v tomto bodě funkce skončila, jelikož šířka parkovacího místa byla vyhodnocena jako 12 cm, to pro které je také šířky 12 cm není dostatečné. Pokračuji však dále a vyhodnotím na tomto příkladu i délku parkovacího místa.

Délka je zpracovávána za použití původního vstupního pole `scan`. Může stát, že v daném měření body předchozím algoritmem vyhodnocené jako body parkovacího místa nebudou tvořit jeden spojitý celek. Při měření mohla být například zaznamenána mezera mezi zaparkovanými vozidly, za vozidlem mohla být nějaká překážka apod. V následujícím algoritmu na výpisu 3.15 tedy nehledám jen délku parkovacího místa, hledám délku *nejdelšího* parkovacího místa. Algoritmus prochází postupně všechny snímky daného měření. Zpracování vychází z hodnoty pravého senzoru pro daný snímek. Pokud je vyhodnocen jako patřící k parkovacímu místu (tedy jestli je hodnota senzoru větší nebo rovna hodnotě `pivot`) a předchozí snímek nebyl součástí parkovacího místa, je tento snímek vyhodnocen jako začátek nějakého parkovacího místa. Pokud při procházení opět narazíme na bod, který součástí parkovacího místa není, je předchozí bod vyhodnocen

Výpis kódu 3.14 Hledání šířky parkovacího místa

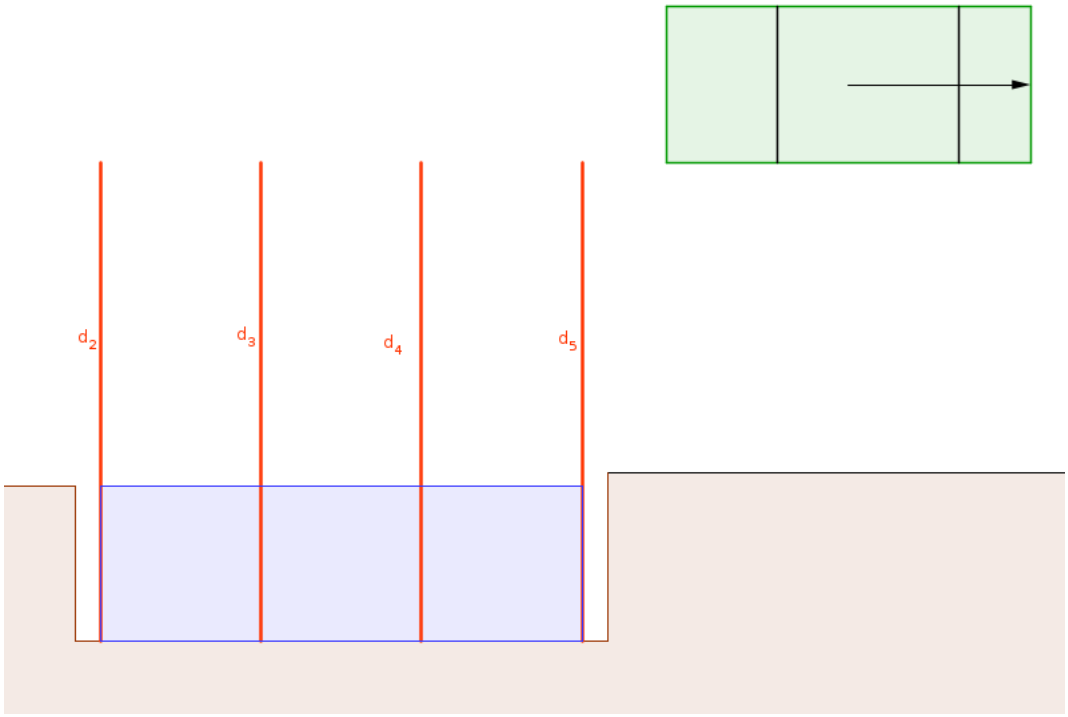
```
pivot_array = []
for val in scan:
    reading = (val[SENSORS][RIGHT])
    pivot_array.append(reading)
    #^Presuneme hodnoty ze senzoru
    # do pomocneho pole
pivot_array.sort()
max_diff = 0
max_diff_element = 0
for i in range(len(pivot_array)):
    if(i == 0):
        continue
    diff = pivot_array[i] - pivot_array[i-1]
    #^Rozdil dvou sousednich prvku
    if (diff > max_diff):
        max_diff = diff
        max_diff_element = i
        #^Ukladame maximum a jeho prislusny index
pivot = pivot_array[max_diff_element]
#^Pivot je prvek napravo od rozdeleni.
# Pivot a vsechny vyssi hodnoty jsou povazovany
# za nalezici parkovacimu mistu.
space_width = max_diff
closest_obstacle = pivot_array[0]
#^Rovnou ustanovime sirku vozidla.
if(space_width < CAR_WIDTH + CAR_WIDTH_MARGIN):
    print("Parking space potentially too narrow for parallel parking.")
    return False
```

Výpis kódu 3.15 Hledání délky parkovacího místa

```
max_parking_space_length = 0
max_parking_space_start = 0
max_parking_space_end = 0
current_parking_space_start = 0
was_space = False
#^Priprava pomocnych promennych
for i in range(len(scan)):
#^Projizdime postupne vstupni pole
    val = scan[i][SENSORS][RIGHT]
    if(val >= pivot):
    #^Pokud soucasny snimek povazujeme za parkovaci misto...
        if(was_space == False):
        #^...a predchozi snimek parkovaci misto nebyl...
            current_parking_space_start = i
            #^Je tento snimek zacatek parkovacího mista.
        was_space = True
    else:
        if(was_space):
        #^Pokud prave skoncilo parkovaci misto...
            first_elapsed = elapsed(scan[current_parking_space_start])
            last_elapsed = elapsed(scan[i-1])
            current_parking_space_length = last_elapsed - first_elapsed
            #^...zmerime jeho delku.
            if(current_parking_space_length > max_parking_space_length):
                max_parking_space_length = current_parking_space_length
                max_parking_space_start = current_parking_space_start
                max_parking_space_end = i
                #^Ukladame si informace o nejdelsim nalezenem.
        was_space = False
if(max_parking_space_length < MINIMUM_PARALLEL_LENGTH):
   print("Parking space potentially too short for parallel parking.")
   print("Found: " + str(max_parking_space_length) + " cm.")
   #^Pokud je parkovaci misto prilis kratke, hlasime neuspech
   # a vypiseme nalezenou delku.
   return False
```

jako konec parkovacího místa. Délku tohoto jednoho parkovacího místa funkce počítá jako rozdíl ujetých vzdáleností na konečném a počátečním bodu. Funkce si udržuje informace o nejdelším nalezeném místě, po dokončení cyklu je toto maximum vyhodnoceno jako délka parkovacího místa

Algoritmus šířku a délku ilustračního parkovacího místa vyhodnotil jako na obrázku 3.5. Šířka parkovacího místa byla stanovena podle jeho nižšího levého okraje prvním algoritmem. Délka byla vyhodnocena druhým algoritmem tak, že parkovací místo leží mezi 2. a 5. snímkem. Podle tabulky hodnot 3.2 je tedy délka parkovacího místa rovna $s_5 - s_2 = 48 - 12 = 36$ cm. Pro podélné parkování však model potřebuje přibližně 41 cm, místo tedy není dostatečně dlouhé. Povšimněme si, že díky nízké vzorkovací frekvenci je změřená délka místa nižší, než ve skutečnosti. Algoritmus tedy pro ilustrační měření vyhodnotil, že parkování není možné, místo je příliš úzké i příliš krátké.



Obrázek 3.5 Vyhodnocené parkovací místo

3.4.3 Vyhodnocení některých tvarů

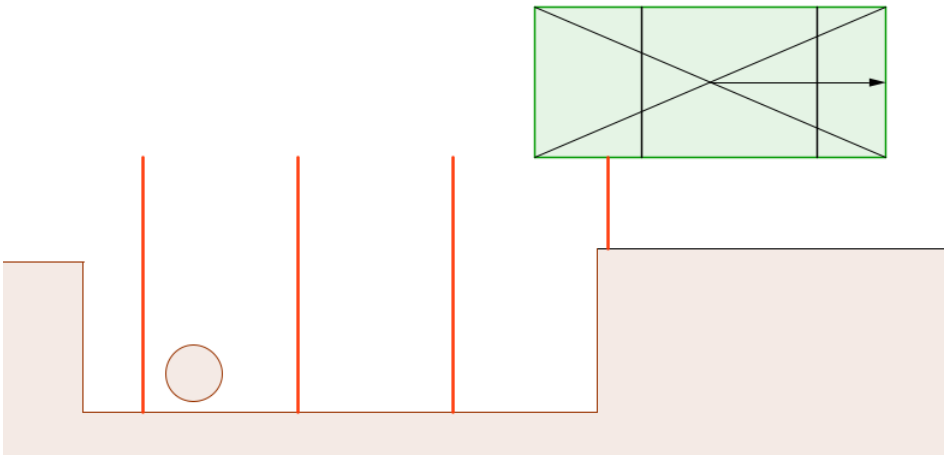
Můj způsob implementace vyhodnocení parkovacího místa je určen k vyhodnocování parkovacích míst umístěných mezi dvěma překážkami podobných šířek a přibližně obdélníkových tvarů. V této sekci uvedu některé situace, které jsou vyhodnoceny jinak, než by se mohlo na první pohled zdát.

Zaprvé, algoritmus ve své současné formě parkovací místo vyhodnotí jako vhodné, jen když existuje překážka před místem i za místem (zaparkovat v takové situaci ani nevyžaduje podélné parkování). Zadruhé, vzdálenosti první a druhé překážky nesmí být příliš rozdílné – jelikož auto parkuje podle nejbližšího bodu, v případě, že by byla druhá překážka značně vzdálenější než ta první, vozidlo by přes ni po straně přečnívalo.

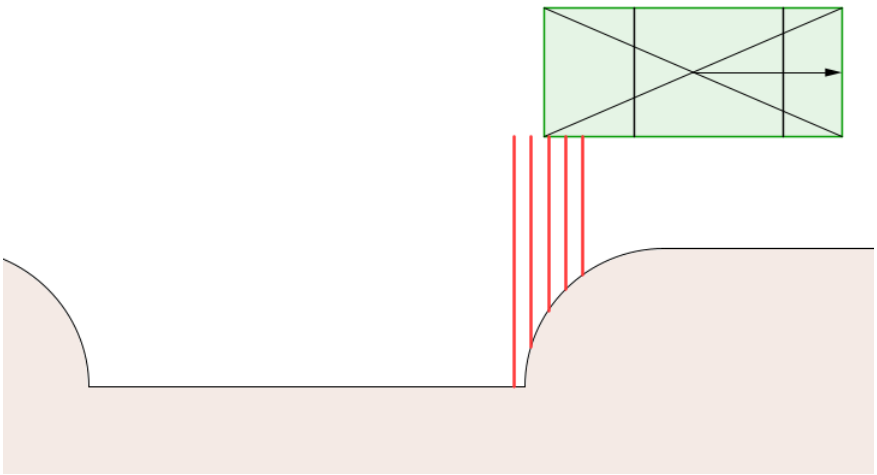
Tyto dva poznatky v běžném provozu nečiní potíže – řidiči parkují podélně couváním jen mezi dvě auta a v případě chybějící přední či zadní překážky by to bylo zbytečné, dále obecně platí, že zaparkovaná auta jsou alespoň velmi přibližně stejně daleko od obrubníku.

Třetí poznatek již v provozu činit potíže může a vyplývá ze způsobu vyhodnocování, kdy v algoritmu používám diskrétní vzorky naměřené s nějakou frekvencí. Nesprávný výběr této frekvence může funkčnost algoritmu rozbít, a to jak v případě příliš nízké i příliš vysoké frekvence. Ilustruji to na dvou případech.

V prvním případě na obrázku 3.6 je situace, kdy je zvolená vzorkovací frekvence příliš nízká. V takovém případě nemusí být zaznamenány některé úzké překážky, například sloupy dopravních značek. V takovémto případě by algoritmus celý prostor vyhodnotil jako spojitý a auto by při pokusu zaparkovat do překážky nabouralo. V druhém případě na obrázku 3.7 je situace, kdy je vzorkovací frekvence příliš vysoká. Reálně zaparkovaná auta nebudou zaparkovaná přesně kolmo a jejich rohy můžou být do různé míry zaoblené. V takovém případě je příliš vysoká frekvence na škodu, jelikož budou zaznamenány tyto drobné rozdíly, v seřazené posloupnosti nebude existovat mezi sousedy dostatečně velký rozdíl a algoritmus nenalezne dostatečně široké parkovací místo.



Obrázek 3.6 Příliš nízká vzorkovací frekvence. Algoritmus nezaznamená překážku.



Obrázek 3.7 Příliš vysoká vzorkovací frekvence. Algoritmus nezaznamená dostatečně velký skok v hodnotách.

Výpis kódu 3.16 Výpočet trajektorie parkování

```
    closest_obstacle = pivot_array[0]
    #^Nejblizsi prekazka, viz predchozi podsekce
    parking_distance = closest_obstacle + CAR_WIDTH + SIDE_SENSOR_OFFSET
    #^Vzdalenost o kolik se vozidlo realne posune smerem k chodniku
    breaking_point_length = TURN_RADIUS - parking_distance/2
    parking_angle = math.acos(breaking_point_length/radius["turn"])
    #^Pomocne rozmery, viz analyticka cast
    arc_length = parking_angle * TURN_RADIUS
    #^Delka jednoho oblouku
    arc_phase_length =2*math.sqrt(radius["turn"]**2-breaking_point_length**2)
    #^Delka obloukove faze, viz analyticka
    status = scan[-1]
    #^Soucasny stav
    target_count =  elapsed(scan[max_parking_space_start]) #Stav zacatku
    target_count += SIDE_SENSOR_TO_REAR #Vzdalenost postranniho senzoru.
    target_count += arc_phase_length + PARALLEL_REAR_MARGIN
    initial_offset = target_count - elapsed(status)
    #^Cilova vzdalenost minus soucasna => kolik bude potreba na zacatku ujet
    return {"initial_offset":initial_offset, "arc_length":arc_length}
```

3.4.4 Výpočet trajektorie

Pokud se algoritmus dostal do této fáze, již vypočítal rozměry parkovacího místa a vyhodnotil je jako dostatečné. V takovém případě může přistoupit k výpočtu trajektorie, který je na výpisu 3.16. Výpočet probíhá přesně podle poznatků z analytické části. Je potřeba zjistit jen dva rozměry – délku pro rovný pohyb S^* a délky pro následné couvání R^- a L^-. Stačí nám dvě vstupní hodnoty – poloměr otáčení, který již známe, a postranní vzdálenost vozidla od jeho současné pozice do jeho pozice po zaparkování. Tu získáme tak, že ke vzdálenosti nejbližší překážky přičteme šířku vozidla, postranní posun vozidla o tuto vzdálenost by ho vyrovnal do úrovně nejbližší překážky. Parkujeme podle nejbližší překážky opět z důvodu bezpečnosti, kdybychom parkovali o nějaký vzdálenější bod, nebylo by možné jednoduše vyloučit kolizi s nejbližším bodem. K této vzdálenosti ještě volitelně přičteme hodnotu, která říká, o kolik postranní senzor přečnívá přes bok vozidla. V modelu je tato hodnota nulová, senzor je s bokem vyrovnaný. V simulaci však pro lepší ilustraci senzor přečnívá.

Jakmile jsou vstupní hodnoty připravené, funkce jednoduše dosadí do odvození z analytické části. Nejdříve spočítá délky oblouků pro couvání R^- a L^-, pak funkce zjistí délku celé této obloukové fáze (obrázek 2.4 z analytické části) a na základě ní a informacích o ujeté vzdálenosti a začátku parkovacího místa vypočítá délku prvního úseku S^* tak, že k ujeté vzdálenosti na začátku parkovacího místa je jednoduše přičtena délka obloukové fáze a rozměr udávající vzdálenost mezi postranním senzorem a pravým zadním rohem vozidla. Oba výsledky jsou zabaleny do struktury a navráceny z funkce. Podle těchto výsledků je možné zaparkovat.

3.4.5 Bezpečnostní odstupy

Ustanovil jsem několik parametrů, které definují bezpečnostní odstupy při parkování a je možné je nastavit v konfiguračním souboru. Jsou následující:

- `minimum_parallel_distance` – Minimální vzdálenost od místa pro podélné parkování.
- `width_margin` – O kolik musí být místo širší než auto.
- `parallel_length_margin` – O kolik musí být místo delší než vypočítané minimum.

Výpis kódu 3.17 Výkon parkování

```
async def perform_parallel_parking(initial_offset, arc_length):
    ret = drive_dist(count_to_cm(initial_offset), 0) #S*
    if(not ret):
        return False
    await drive_dist(-arc_length, STEERING_ANGLE) #R-*
    if(not ret):
        return False
    await drive_dist(-arc_length, STEERING_ANGLE) #L-
    if(not ret):
        return False
    status = await drive(0, 0)
    front_dist = min(status[SENSORS][FRONT_2], status[SENSORS][FRONT_3])
    rear_dist = min(status[SENSORS][REAR_2], status[SENSORS][REAR_3])
    dist = front_dist - (front_dist+rear_dist)/2
    #^Vypocitame, jak vyrovnat vzdalenosti pomoci udaju ze senzoru
    ret = await drive_dist(dist, 0) #R+
    if(not ret):
        return False
    return True
```

- `parallel_rear_margin` – V jaké vzdálenosti skončí auto po parkování před zadní překážkou. Tato hodnota musí být menší než `parallel_length_margin`.

3.4.6 Provedení

Samotné provedení je po vypočítání trajektorie jednoduché. Využívám pro něj funkci pro řízení ujeté vzdálenost. Provedení je na výpisu kódu. Funkce provede úseky (S^*, R^-, L^-, S^+). První tři úseky provede podle dříve vypočítaných parametrů. Poslední krok S^+ je proveden tak, že vozidlo z dat z předních a zadních senzorů vypočítá vzdálenost, kterou je nutnou ujet, aby byla vzdálenost před autem a za ním stejná. Parkování je považováno za úspěšné, pokud byly úspěšně vykonány všechny čtyři úseky.

Kapitola 4
Experimentální část

V této části jsem popsal reálné použití vytvořeného softwaru. Navržené algoritmy jsem otestoval a zhodnotil.

4.1 Instalace a použití

Tato část slouží zároveň jako návod k použití, předpokládám použití Linuxu. Zmiňovaný software lze najít v příslušných podadresářích adresáře `software` v příloze práce.

4.1.1 Model

Pro použití modelu je nejprve nutné zajistit, že je zapojena baterie pro Raspberry Pi a baterie pro spodní desku a že je napájení spodní desky zapnuté. Dále pak musí být na MicroSD kartě v Raspberry Pi připraven operační systém, například Raspberry Pi OS. Systém musí být nastavený tak, že se Raspberry Pi po zapnutí samo připojí k internetu a je možné s ním navázat spojení přes SSH.

Firmware by měl být na Arduinu již nahraný. Pokud tam není či není aktuální, je třeba se připojit na Raspberry Pi přes SSH a nainstalovat vývojové prostředí pro Arduino:

```
# apt install arduino
```

Pomocí něj je možné zkompilovat firmware a nahrát jej do Arduina. Zda-li je vše správně připraveno ověří až terminál.

4.1.2 Simulace

Pro spuštění simulace na Linuxu stačí spustit přiložený binární soubor. Pro spuštění ze zdrojového kódu (např. pro úpravy mapy či spuštění na jiných operačních systémech) je třeba otevřít soubor projektu v kořenovém adresáři simulátoru pomocí editoru enginu, který je dostupný z webových stránek [11]. V každém případě je nutné, aby mohl současný uživatel otevřít websocket na portu 9123. Po spuštění simulace je možné vyzkoušet ovládání vozidla pomocí šipek klávesnice a jeho manipulaci pomocí myši.

4.1.3 Terminál

Teď ukážu, jak zprovoznit terminál. Při použití hardwarového modelu je třeba tyto kroky dělat na Raspberry Pi modelu po připojení se přes SSH. Je třeba nainstalovat Python 3 a jeho instalátor balíčků, například na systémech založených na Debianu takto:

```
# apt install python3 python3-pip
```

Pak je možné nainstalovat závislosti terminálu:

```
$ pip install numpy configparser pyserial websockets
```

Teď je terminál připravený ke spuštění. Po spuštění se pokusí sám připojit na model či simulátor, doporučuji tedy mít jednu z těchto možností v momentu spuštění připravenou. Terminál se spouští následovně:

```
$ python3 terminal.py
```

Pokud se připojujeme na model, terminál by měl začít vypisovat informace ze sériové linky. Pokud toto nenastalo, je možné, že se připojujeme na špatné zařízení. Výchozí zařízení je `/dev/ttyUSB0`. Zkontrolujeme adresář `/dev/`, jestli v něm existuje položka `/dev/ttyUSBx`. Pokud žádná taková položka neexistuje, zkontrolujeme fyzické připojení Arduina a Raspberry Pi. Pokud existuje, buď zapneme terminál a připojíme se ručně pomocí příkazu `soccon`, nebo upravíme výchozí sériové zařízení v souboru `config.ini`. Jakmile připojení přes sériovou linku funguje, čteme příchozí výpisy. Buď je zahlášen úspěch, nebo nějaká chyba. Pokud je zahlášena chyba, nelze ji ignorovat, jedná se o hardwarový problém, například není zapnuté napájení spodní desky, či není připojena senzorová lišta. Pokud je zahlášen úspěch, elektronika by měla být v pořádku a vozidlo je připraveno přijímat příkazy.

Pokud se připojujeme na simulátor, terminál by měl po neúspěšném pokusu o připojení na sériovou linku zahlásit úspěšné připojení na terminál téměř okamžitě. Pokud je simulátor spuštěn a úspěch nenastal, simulátor ukončíme a spustíme ho znovu v terminálu, aby bylo možné sledovat jeho výpis. V takovém případě nejspíše selhala tvorba websocketu na portu 9123, kterou uživatel simulátoru musí umožnit. V případě úspěchu je simulátor připraven přijímat příkazy.

Seznam příkazů je dostupný pomocí příkazu `help` či v praktické části této práce.

4.2 Testování v simulaci

V simulaci probíhala z praktických důvodů většina testování vytvořených funkcionalit. Pro simulaci jsem vytvořil jednoduchou mapu složenou ze stěn a překážek ve tvaru automobilů, které jsem uspořádal do parkovacích konfigurací s jedním volným místem. V simulátoru jsem otestoval kalibraci, řízení ujeté vzdálenosti, parkování a chybovost.

4.2.1 Kalibrace

V simulátoru pro kalibraci vozidla stačí vytvořit odhad brzdné dráhy. Naměřil jsem vzorky pomocí příkazu `fbd`, kterému jsem zadal vzdálenosti pro rozjíždění 0,25; -0,25; 0,5; -0.5; 1; -1; 2; ...; 64; -64 cm. Tato posloupnost je uvedena i v příkazu `help`, stačí ji tedy nakopírovat do terminálu a měření s těmito parametry proběhne. Toto měření jsem provedl třikrát, celkem tedy 48 vzorků. Zavolal jsem příkaz `fbp`, který provedl lineární regresi a vypsal parametry výsledné přímky. Nakonec jsem je aplikoval a tím i zapsal do konfigurace příkazem `sbp`. Průběh kalibrace simulátoru jsem zaznamenal do výpisu 4.1.

4.2.2 Řízení ujeté vzdálenosti

Otestoval jsem kvalitu odhadu brzdné dráhy a celého systému řízení ujeté vzdálenosti. Pro simulátor jsem nastavil tolerovanou chybu vcelku přísně na 1 mm. Sledoval jsem dvě hodnoty – chybu v prvním cyklu před opravami (například by ukázala, jestli vozidlo na začátku významně nepřejíždí svůj cíl, což je rizikové) a počet potřebných korekcí. Před testováním jsem ještě nastavil hodnoty pro interpolaci PWM tak, že vozidlo pro vzdálenosti od 30 cm a výše jede svou maximální rychlostí, pro vzdálenosti 1 cm a méně jen čtvrtinovou. Toto jsem udělal čistě experimentálně tak, aby bylo vozidlo vůbec schopné se pohybovat s přesností stanoveného 1 mm, ale aby se zároveň pohybovalo dostatečně plynule a přirozeně. Třikrát jsem naměřil vzorky pro

Výpis kódu 4.1 Kalibrace simulátoru

```
> fbd 0.25
Peak velocity: 295.22798620398396
Brake distance: 3.0242376767413663
Training dataset size: 1
> fbd -0.25
Peak velocity: -304.8496213277514
Brake distance: -3.0242376767413663
Training dataset size: 2
...
> fbd -64.0
Peak velocity: -2333.7343163165947
Brake distance: -16.608613279775444
Training dataset size: 48
> fbp
Trained on 48 samples.
Linear term: 0.1793547363710273
Constant term: 3.0477382670305357
To set braking parameters:
sbp 0.1793547363710273 3.0477382670305357
> sbp 0.1793547363710273 3.0477382670305357
Applying...
```

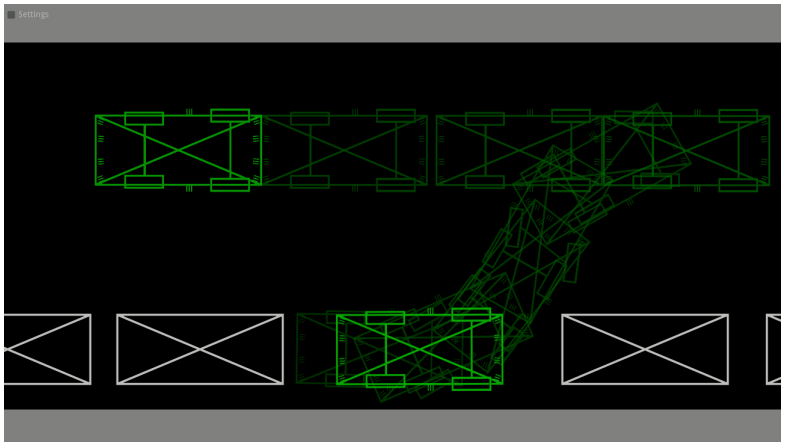
cílové vzdálenosti 1; 2; ...; 128 cm, celkem tedy 24 vzorků. Toto je na výpisu 4.2. Průměrná hodnota první chyby vyšla zaokrouhleně -1.04 cm, což znamená, že vozidlo při svém prvním průběhu cyklem zastavilo 1,04 cm před cílem a pak tuto vzdálenost velmi pomalu dojíždělo vpřed. Jen ve dvou případech se stalo, že vozidlo v prvním průběhu zastavilo za cílem, v obou případech to bylo o méně než milimetr. Vozidlo potřebovalo v průměru 5.28 opravných cyklů, všechny kromě toho prvního však byly v řádu jednotek milimetrů. Systém řízení ujeté vzdálenosti hodnotím v simulátoru za zcela vyhovující.

4.2.3 Parkování

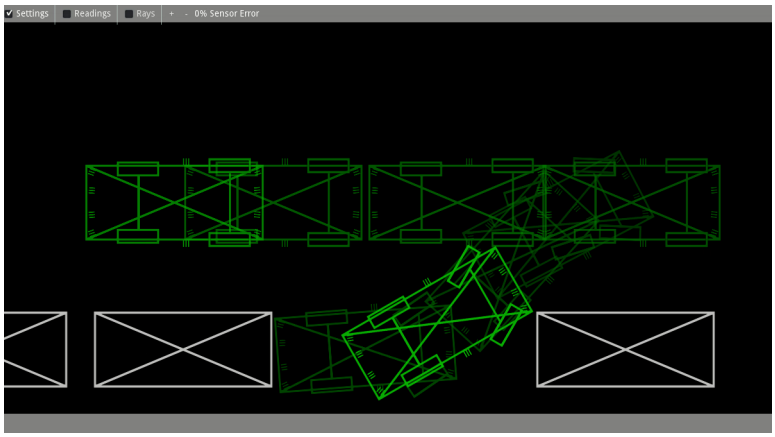
Vypočítaná minimální délka parkovacího místa vyšla 41 cm. Nastavil jsem pro začátek následující parametry – parkovací místo musí být alespoň o 4 cm delší než minimum a parkujeme o 2 cm

Výpis kódu 4.2 Testování řízení ujeté vzdálenosti

```
> d 1 0
first_overshoot: -0.4932727405043331
cycles: 6
> d 2 0
first_overshoot: -0.47981822151297915
cycles: 5
> d 4 0
first_overshoot: -0.4918882034913361
cycles: 6
> d 8 0
first_overshoot: -0.16521698779747335
cycles: 2
> d 16 0
...
```



Obrázek 4.1 Parkování v simulaci za běžných podmínek



Obrázek 4.2 Parkování v simulaci do nejkratšího možného místa

před zadní překážku. Parkovací místo bylo 46 cm dlouhé, auto má tedy pro parkování jistou rezervu. Pro tyto parametry byla úspěšnost parkování stoprocentní. Otestoval jsem parkování z různých vzdáleností, auto vždy skončilo na stejném místě bez ohledu na výchozí vzdálenost. Koláž jednoho parkování za těchto podmínek (tentokrát z vcelku velké vzdálenosti) je na obrázku 4.1. Zvýrazněná je počáteční a konečná pozice vozidla v simulaci. Videa, ze kterých všechny koláže vycházejí, je možné najít v příloze v adresáři `video`.

Otestoval jsem, zda-li je použitý výpočet minimální délky parkovacího místa správný. Koláž je na obrázku 4.2. Posunul jsem překážky tak, že velikost parkovacího místa byla větší než ta minimální jen o necelý milimetr a pokusil jsem se auto nechat zaparkovat. Bezpečnostní odstupy pro délku místa jsem v konfiguračním souboru nastavil na nulu. Trvalo několik pokusů, než algoritmus vyhodnotil parkovací místo jako dostatečné kvůli drobným nepřesnostem vytvořených vzorkováním. Vozidlo se pokusilo zaparkovat a svým pravým předním rohem lehce zavadilo o roh překážky před autem. V koláži je zvýrazněna výchozí pozice vozidla a bod kolize. Tento náraz lehce vychýlil rotaci auta, stále však skončilo vcelku úspěšně zaparkované. Vypočtenou minimální délku hodnotím jako správnou.

4.2.4 Testování s chybou

V simulátoru je možné nastavit hodnotu maximální chyby na senzorech v procentech. Každé měření vzdálenosti je pak zatíženo náhodnou chybou na intervalu $\langle -\text{chyba}; \text{chyba} \rangle$. Otestoval jsem algoritmy pro různé hodnoty chyb. Pro chybu do 5 % jsem nezaznamenal významné rozdíly při řízení či parkování. Při chybě okolo 10 % začalo být parkování znatelně nepřesné, vozidlo vyčnívalo přes okolní auta kvůli tomu, že byla vzdálenost nejbližší překážky (podle které se parkuje) vyhodnocena jako příliš blízko. Při chybě okolo 15 % vozidlo začalo odmítat zaparkovat z důvodu, že chyby smazaly ostrý skok v seřazené posloupnosti vzdáleností ze senzoru, podle kterého se vyhodnocuje šířka parkovacího místa. Terminál pak hlásil, že je místo příliš úzké. Od 20 % výše již nebylo parkování vůbec možné a často chybovalo i samotné řízení vzdálenosti kvůli tomu, že prevence kolizí vracela příliš mnoho falešně pozitivních výsledků a auto pak zbytečně brzdilo, naopak některé kolize zachyceny nebyly.

4.3 Testování v laboratoři

Testování fyzických modelů jsem prováděl v Laboratoři inteligentních vestavných systémů na FIT ČVUT. Vozidlo jsem zkalibroval a otestoval jsem řízení ujeté vzdálenosti a parkování.

4.3.1 Nedostatky modelů

Pro začátek zmíním několik nedostatků, na které jsem při práci s modely narazil. Tyto nedostatky často značně komplikovaly práci a budu se na ně ve zbytku sekce odkazovat.

Při komunikaci po sériové lince docházelo k vysoké chybovosti. Raspberry Pi často přijalo některá data chybně, což nehrálo díky vysoké vzorkovací frekvenci velkou roli a nepřesnosti se neprojevily. Jediné co toto prakticky způsobovalo bylo, že model někdy začal kvůli špatné informaci o rychlosti brzdit dříve než musel, toto však vyrovnaly korekce. Horší byla situace, kdy Raspberry Pi nebo Arduino přijalo data, kde neodpovídal počet přijatých bytů a data nešla vůbec interpretovat. Toto jsem se pokusil provizorně ošetřit opakovaným posíláním či oříznutím dat na očekávaný rozsah, neošetřilo to však všechny situace a v případě takové chyby bylo nutné se připojit na sériovou linku znovu. Tento problém rovněž značně ztěžoval nahrávání nového firmwaru do Arduina – díky velkému množství dat k nějaké chybě došlo ve většině případů, vždy bylo potřeba mnoho pokusů o nahrání, než kontrolní součet vyhovoval a nahrání bylo vyhodnoceno jako úspěšné. Tato chybovost bude muset být v budoucnu řádně ošetřeno na obou koncích sériové komunikace.

Při některých pokusech začaly enkodéry vykazovat zvláštní druh chybovosti, kdy pro stejnou reálnou ujetou vzdálenost byla naměřená ujetá vzdálenost při pohybu vzad výrazně nižší, než při pohybu vpřed, což zcela znemožnilo použití řídících a tudíž i parkovacích funkcí. Při pohybu vpřed zůstala standardní velmi nízká chybovost enkodérů nedotčena. Příčinu tohoto problému (zcela jistě hardwarovou) se mi nepodařilo objasnit.

Postranní senzory nejsou uchyceny dostatečně. Lišta, na které jsou připevněny, je k modelu uchycena pevně jen v jednom bodě na své zadní části, tudíž je možné vcelku lehce senzor vychýlit tak, že již nesměřuje přímo doleva či doprava. Při běžné manipulaci s modelem k tomuto často nedopatřením docházelo, což v takových případech výrazně zhoršilo přesnost parkování.

Propojení mezi senzorovými lištami a deskou s Arduinem není dostatečně spolehlivé. Lišta se v průběhu testování často odpojovala, takže firmware při počátečním testu elektroniky po propojení s terminálem hlásil neúspěch kvůli chybějící senzorové liště. Někdy tentýž problém vykazovaly i motory.

4.3.2 Kalibrace

První krok kalibrace hardwarového modelu je vystředění kol. To jsem udělal opakovaným voláním příkazu `szz`, který nastaví nový nulový bod kol a zapíše ho do konfiguračního souboru.

Druhý krok je stejný v simulátoru, bylo třeba vytvořit odhad brzdné dráhy. Vozidlu jsem zadal stejná data jako při kalibraci simulace a jejich následné zpracování a aplikování se také nelišilo. Musel jsem jen zajistit kolem vozidla dostatek prostoru a vozidlo hlídat, jelikož se zřídka stalo to, že po chybě v sériové komunikace zůstala hodnota pro PWM motorů na poslední hodnotě navzdory mechanismu ve firmwaru, který má přesně této situaci zabránit.

4.3.3 Řízení ujeté vzdálenosti

Řízení ujeté vzdálenosti jsem testoval analogicky jako při testování simulátoru. Tolerovanou chybu jsem oproti simulátoru nastavil na vyšší hodnotu, konkrétně 3 mm. Používal jsem stejná data jako při testování simulace. Průměrná hodnota první chyby vyšla přibližně -3,59 cm, provedená kalibrace tedy brzdnou dráhu spíše zbytečně nadhodnocovala. Při tomto testování však před první korekcí vozidlo cíl ani jednou nepřesáhlo, takže bylo řízení bezpečné. Průměrný počet potřebných korekcí byl zaokrouhleně 4,58.

4.3.4 Parkování

Pro maximální přesnost senzorů byl parkovací prostor složen z čistě bílých předmětů. Pro lepší představu o přesnosti parkování byl vynechán poslední vyrovnávací úsek S^+. Bezpečnostní odstupy jsem stanovil stejně jako v simulaci.

Většinu testů jsem dělal při bezpečné velikosti místa 60 cm. Provedl jsem celkem 20 parkovacích pokusů. 11 z nich jsem vyhodnotil jako úspěch – vozidlo zaparkovalo bezpečně a nepřečnívalo z parkovacího místa. 2 z nich jsem vyhodnotil neutrálně – vozidlo sice zaparkovalo bezpečně, ze svého parkovacího místa však přečnívalo o více než 1 cm. Tento jev byl nejspíše způsoben nepřesnostmi senzorů vzdálenosti. 7 pokusů selhalo z toho důvodu, že při výkonu nějakého úseku či prvotního měření došlo při komunikaci přes sériovou linku k nějaké chybě.

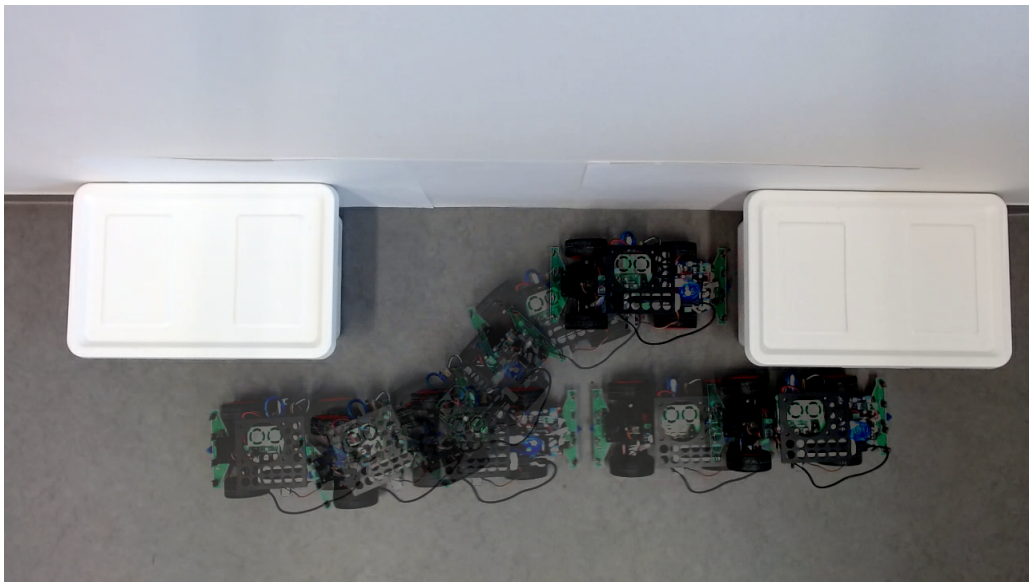
Na obrázku 4.3 je zobrazena koláž z úspěšného parkování. Model nebyl na začátku vyrovnán zcela rovnoběžně a při měření místa jel mírně doleva. Přesto však místo vyhodnotil správně a úspěšně zaparkoval.

Na obrázku 4.4 je zobrazena koláž z neúspěšného parkování. Model sice vyhodnotil místo správně a provedl první pohyb S^-, při pohybu R^- však došlo k přerušení spojení a model narazil do stěny.

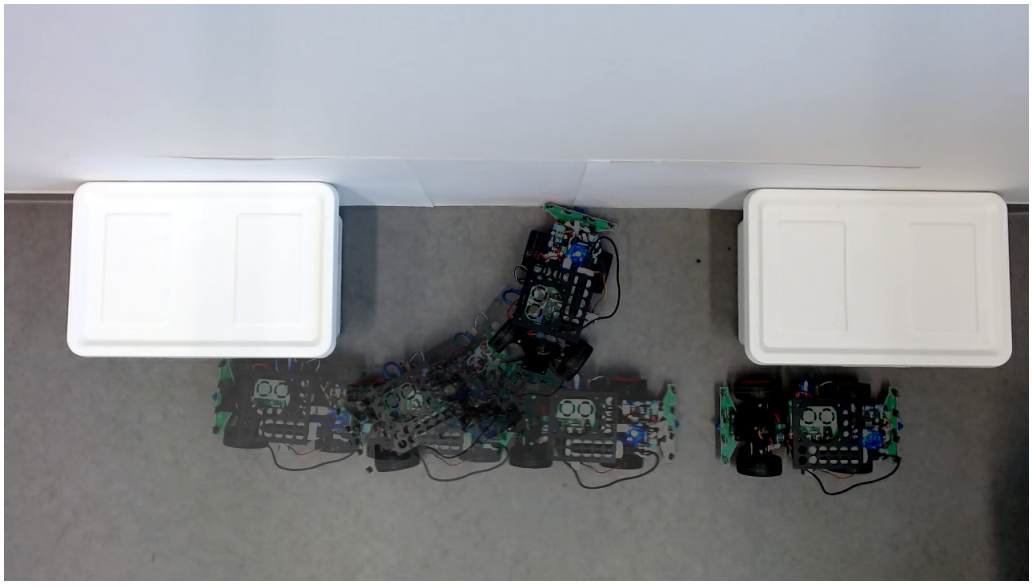
Pro experiment jsem poté velikost parkovacího místa postupně zmenšoval, za předpokladu absence chyb přenosu dat dokázal model konzistentně bezpečně parkovat do místa dlouhého 47 cm, při nižších vzdálenostech se začínaly objevovat drobné kolize, hlavně o roh překážky před místem na obou úsecích R^- a L^-. Při velikosti místa 47 cm bylo už navíc nutné, aby bylo vozidlo s místem co nejpřesněji rovnoběžně. Jelikož je teoretická minimální délka parkovacího místa 41 cm, není tento výsledek příliš působivý, považuji jej však za dostatečný. Koláž z částečně úspěšného parkování do prostoru o délce 47 cm je na obrázku 4.5.

Měnil jsem i rychlost vozidla během parkování pomocí změny hodnoty `lerp_at_hi` v konfiguračním souboru, která nastaví maximální rychlost pohybu vozidla. Nepřekvapivě bylo vyhodnocení parkovacího místa při nižších rychlostech přesnější – parkovací místo reálně dlouhé 47 cm bylo při maximální možné rychlosti vyhodnocováno jako dlouhé 44 cm, při poloviční rychlosti už výrazně přesněji jako 45,5 cm.

Data ze senzorů byla dostatečná za nezanedbatelného předpokladu dobré odrazivosti povrchu. Na všech kolážích z testování modelu si je možné všimnout bílých papírů za parkovacím prostorem. Toto bylo nutné z toho důvodu, že proti šedé liště na zdi byla chybovost značně vyšší a místo bylo díky tomu výrazně častěji vyhodnoceno jako nedostatečně široké.



Obrázek 4.3 Úspěšné parkování modelu. Parkovací prostor dlouhý 60 cm.



Obrázek 4.4 Neúspěšné parkování modelu. Parkovací prostor dlouhý 60 cm. V tomto případě došlo při R^- k chybě sériové linky.

Obrázek 4.5 Částečně úspěšné parkování. Parkovací prostor dlouhý 47 cm. Místo bylo sice vyhodnoceno správně, v tomto případě se však projevila zmíněná chybovost enkodérů při couvání a vozidlo ujelo při R^- a L^- příliš malé vzdálenosti.

4.4 Diskuze

V této kapitole jsem provedl sérii testů, jejichž cílem bylo ověřit funkčnost a spolehlivost funkcionalit, které jsem v rámci této bakalářské práce vytvořil. V této sekci diskutuji jejich výsledky.

Jako první jsem algoritmy otestoval v simulaci. Hlavním účelem tohoto testování bylo ověření správnosti algoritmů a vypočítaných trajektorií, z tohoto důvodu jsem při testování v simulaci používal velmi nízkou toleranci chyby. Pomocí kalibračních funkcí jsem vytvořil odhad brzdné dráhy a otestoval jsem systém řízení ujeté vzdálenosti, kdy jsem sledoval hodnotu první chyby a počet potřebných korekcí. Tento systém pracoval správně, virtuální vozidlo nikdy nezanedbatelně nepřejelo svůj cíl a zadanou vzdálenost vždy s relativně nízkým počtem korekcí ujelo. V simulaci tento systém hodnotím jako naprosto dostačující. Dále jsem otestoval algoritmus parkování. Jako první jsem testoval parkování do místa běžné délky z různých počátečních pozic vozidla. Do tohoto místa vozidlo zaparkovalo vždy, vypočítaná trajektorie se shodovala s očekávanou a vozidlo vždy po zaparkování skončilo na stejném místě. Výpočet trajektorií hodnotím jako správný. Nakonec jsem otestoval správnost výpočtu minimální délky parkovacího místa. Vozidlu jsem zadal zaparkovat do nejkratšího možného místa, vždy nastala kolize téměř přesně mezi rohy vozidla a parkovacího místa. Tento výpočet tedy rovněž hodnotím jako správný. Otestoval jsem odolnost algoritmů vůči chybám senzorů, kterou jsem očekával vyšší – už při chybě 10 % bylo parkování znatelně nepřesné a nad 20 % nepoužitelné.

Poté jsem algoritmy otestoval na fyzických modelech. Hlavním účelem tohoto testování bylo ověření použitelnosti a spolehlivosti algoritmů na hardwaru. Nejdříve jsem provedl kalibraci stejně jako v simulaci, navíc bylo třeba jen vystředit kola. Funkce řízení ujeté vzdálenosti pracovala správně za předpokladu správné funkce enkodérů na motorech, které někdy při couvání měřily špatné hodnoty. Otestoval jsem algoritmus parkování, který za předpokladu absence hardwarových chyb dosahoval úspěšnosti přibližně 85 %. Při započítání chyb způsobených hardwarem byla úspěšnost pouze 55 %. Jako hlavní podnět pro další vylepšení systému tedy jednoznačně považuji opravu mnou identifikovaných nedostatků hardwaru. Data ze senzorů byla pro parkování dostatečná, musel jsem však parkovací prostor seskládat jen z bílých objektů.

Celkově výsledky testování považuji za uspokojivé a cíl práce za splněný. Navržené algoritmy

považuji za použitelné za předpokladu správné funkce hardwaru. Vypočítané trajektorie a vzdálenosti považuji za správné. Odhalil jsem, že hlavní problémy při práci s fyzickým modelem jsou hardwarového původu.

Za hlavní přínos své práce pro budoucí použití nepovažuji samotný parkovací algoritmus, ale funkcionality, které jsem pro jeho tvorbu potřeboval a které mi jeho tvorbu usnadnily. Hlavně kalibrační a řídící funkce mají potenciál nalézt využití i mimo rámec parkování. Jejich další vylepšení, jako například pečlivá prevence kolizí, představují možný směr dalšího vývoje. Terminál který jsem vyvinul usnadňuje s modely práci a umožňuje snadné přidávání vlastních funkcí. Tato infrastruktura poskytuje rozšiřitelný základ pro budoucí práci s modely.

Kapitola 5
Závěr

Cílem práce bylo vytvořit parkovací software pro modely vozidel v Laboratoři inteligentních vestavných systémů na FIT ČVUT. Tento cíl považuji za splněný. S vozidly jsem se seznámil, identifikoval jsem jejich nedostatky a vytvořil jsem pro ně systém přesného řízení. Pro obsluhu tohoto systému a celkově lepší uživatelskou použitelnost jsem v Pythonu vytvořil terminál určený ke komunikaci s vozidlem. Pro tento terminál jsem implementoval celou řadu příkazů, například pro připojování, kalibraci či řízení. Prozkoumal jsem existující řešení tématiky autonomního parkování a na základě získaných poznatků jsem odvodil potřebné rozměry a vytvořil parkovací algoritmus. Tento algoritmus parkovací místo změří, vyhodnotí jeho způsobilost pro parkování, vypočítá parkovací trajektorii a po této trajektorii model zaparkuje. Pro testování všech funkcionalit jsem vytvořil v enginu Godot simulátor, který komunikuje s terminálem a replikuje dostatečně věrně chování reálného modelu.

Algoritmus parkování dosahuje uspokojivých výsledků jak v simulaci, tak na reálném modelu. V simulaci model parkuje bezchybně. Fyzický model parkuje dostatečně přesně, funkčnost systému však narušuje vadná komunikace s komponentami nižší úrovně. Data ze senzorů hodnotím jako dostatečná za předpokladu dostatečně reflexivního povrchu.

Na výsledky mé práce by šlo navázat přidáním dalších terminálových příkazů, například pro jiný typ parkování. Dále by šel vylepšit systém prevence kolizí tak, aby ošetřil více situací. Dalším podnětem pro budoucí práce by mohlo být zprovoznění funkce přímého řízení. Jako nejakutnější požadavek však vidím zajištění spolehlivé komunikace mezi terminálem a firmwarem.

Bibliografie

1. KOLÁŘ, Petr. *Řízení modelů autonomních vozidel*. České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. Dipl. pr.
2. STMICROELECTRONICS. *Time-of-Flight ranging sensor* [online]. 2022. [cit. 2023-06-15]. Dostupné z: `https://www.st.com/resource/en/datasheet/vl53l0x.pdf`.
3. STMICROELECTRONICS. *Dual Full-Bridge Driver* [online]. 2000. [cit. 2023-06-13]. Dostupné z: `https://pdf1.alldatasheet.com/datasheet-pdf/view/22440/STMICROELECTRONICS/L298N.html`.
4. HAN, Inhwan. Geometric Path Plans for Perpendicular/Parallel Reverse Parking in a Narrow Parking Spot with Surrounding Space. *Vehicles*. 2022, roč. 4, č. 4, s. 1195–1208. ISSN 2624-8921. Dostupné z DOI: `10.3390/vehicles4040063`.
5. BINDU, Yamuna. *Park Assist Technology: A history* [online]. 2021. [cit. 2023-06-07]. Dostupné z: `https://blog.getmyparking.com/2021/11/11/park-assist-technology-a-history/`.
6. LI, Chenxu; JIANG, Haobin; MA, Shidian; JIANG, Shaokang; LI, Yue. Automatic Parking Path Planning and Tracking Control Research for Intelligent Vehicles. *Applied Sciences*. 2020, roč. 10, č. 24. ISSN 2076-3417. Dostupné z DOI: `10.3390/app10249100`.
7. VIRTUAL CRASH, LLC. *Turning Radius* [online]. 2021. [cit. 2023-06-15]. Dostupné z: `https://www.vcrashusa.com/kb-vc-article99`.
8. MAOYUE, Li; PENG, Zhou; XIANGMEI, He; HONGYU, Lv; HONGCHUN, Zhang. *Parallel Parking Path Planning and Tracking Control Based on Adaptive Algorithms*. Sv. 22. Springer Science a Business Media LLC, 2021. Č. 4. Dostupné z DOI: `10.1007/s12239-021-0086-3`.
9. MECHANICAL SIMULATION CORPORATION. *CarSim Overview* [online]. 2023. [cit. 2023-06-10]. Dostupné z: `https://www.carsim.com/products/carsim/index.php`.
10. CYBERBOTICS LTD. *Cybersim: Robotics simulation with Webots* [online]. 2023. [cit. 2023-06-10]. Dostupné z: `https://www.cyberbotics.com/`.
11. LINIETSKY, Juan; MANZUR, Ariel. *Features - Godot Engine* [online]. 2023. [cit. 2023-06-10]. Dostupné z: `https://godotengine.org/features/`.
12. ELDER, Robert. *Why Is It so Hard to Detect Keyup Event on Linux?* [Online]. 2019. [cit. 2023-06-12]. Dostupné z: `https://blog.robertelder.org/detect-keyup-event-linux-terminal/`.

Obsah přiloženého média

- `README.txt` ..... popis obsahu, je i v příslušných podadresářích
- `latex.zip` ..... zdrojový kód této práce v Latexu
- `geogebra` ..... soubory pro program Geogebra
- `video` ..... videodokumentace parkovacích experimentů
- `software` ..... vytvořený software
  - `firmware` ..... upravený firmware
  - `simulator` ..... simulátor vozidla
    - `bin` ..... adresář se spustitelným souborem simulátoru
    - `src` ..... adresář s projektem simulátoru
  - `terminal` ..... terminál
    - `doc` ..... dokumentace terminálu
    - `src` ..... moduly terminálu