Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra kybernetiky

Diplomová práce

Plzeň, 2021 Josef Švec

Místo této strany bude zadání práce

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni. Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 15. března 2021

Josef Švec

Abstract

This thesis deals with design of wheeled inspection robot control system using ROS framework. The task of the robot is to visit autonomously points of interest, which can be defined in the created user interface. The thesis contains a complete description of the used robotic platform, followed by mathematical description of used robot, the next part deals with the ROS software framework and its use for inspection robot, then the created user interface is described. At the end of this thesis there are summarized the results and knowledge gained during the implementation of the control system.

Keywords

ROS, Ackerman model of control, HMI, inspection robot

Abstrakt

Tato práce se zabývá návrhem řídicího systému inspekčního kolového robota s využitím softwarového rámce ROS. Úkolem robota je samostatně navštívit body zájmu, které je možné definovat ve vytvořeném uživatelském rozhraní. V práci je uveden kompletní popis použité robotické platformy, následuje matematický popis použitého robota, další část pojednává o sofwarovém rámci ROS a jeho využití pro řízení inspekčního kolového robota, dále je popsáno vytvořené uživatelské rozhraní. V závěru práce jsou shrnuty výsledky a znalosti získané během realizace řídicího systému.

Klíčová slova

ROS, Ackermanův model řízení, HMI, inspekční robot

Obsah

1 Úvod

Robot je neživá soustava, která je schopná vykonávat určitý úkol v definovaném pracovním prostředí. K poznání okolního prostoru roboti využívají senzory (kamera, lidar atd.). Aby se roboti mohli pohybovat, musí být vybaveni aktuátory. Zpracováním dat ze senzorů a navržením vhodného řídicího systému vznikne stroj, který zapojuje aktuátory tak, aby plnil zadaný úkol.

Existuje nepřeberné množství robotů a jejich dělení. Základní rozdělení můžeme stanovit z hlediska možnosti přemisťovat se. Stacionární roboti se nemohou pohybovat z místa na místo (například průmysloví roboti v továrnách [34]). Mobilní roboti se naopak mohou přemisťovat (například autonomní vozidla [25]).

Mobilní roboty můžeme dále rozdělit na základě prostředí, ve kterém se pohybují. V dnešní době existují roboti, kteří zkoumají nebo čistí oceány. Někteří roboti dokonce prozkoumávají cizí planety. Dále existují mapříklad létající roboti (drony).

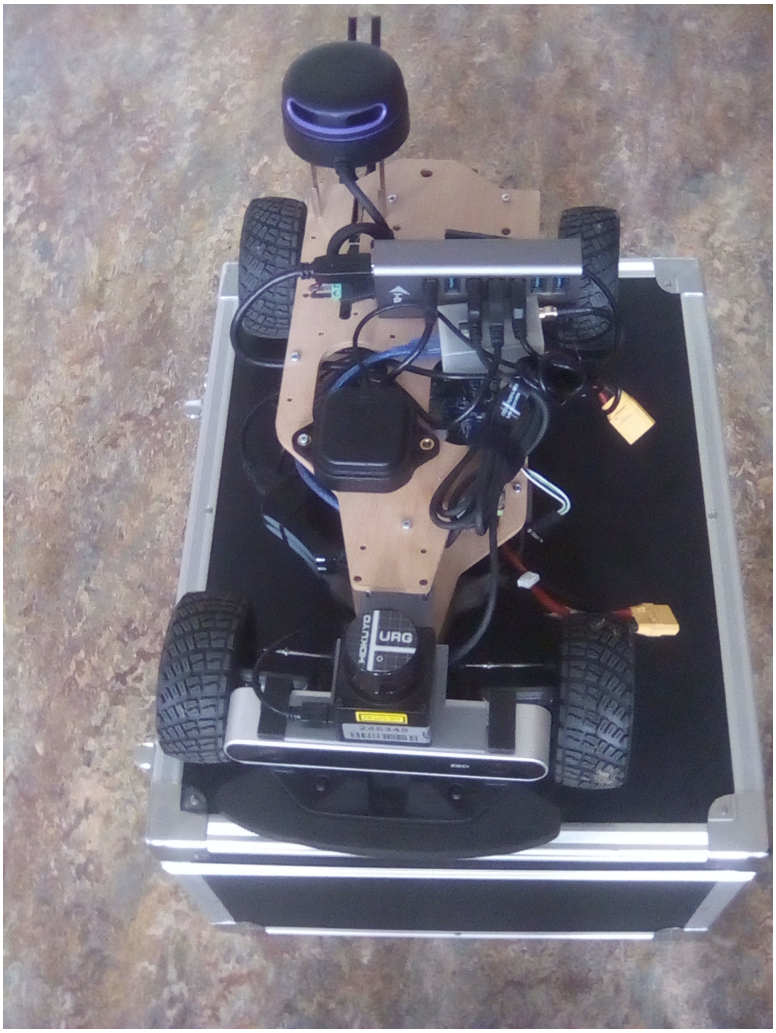
V této práci je popsáno řízení inspekčního kolového robota, který autonomně jezdí po zemském povrchu. Takový robot lze využít například pro vizuální kontrolu nějakého areálu, kde může nahradit lidské stráže. Zadáním důležitých průjezdních bodů v mapě lze proces kontroly objektů automatizovat. Samostatným problémem je stanovení bodů, které by měl robot navštívit [52]. Další aplikace využívající průjezdních bodů, které jsou definované v mapě, lze nalézt například v chytrém zemědělství [49]. Inspekční autonomní robot může být dále využit tam, kde je pro člověka příliš nebezpečné prostředí (hořící budova, zamořená oblast).

Cílem této práce je navrhnout řídicí systém inspekčního vozidla, které bude samostatně navštěvovat body zájmu. Tyto body bude možné snadno definovat v uživatelském rozhraní. Pokud se vyskytne v cestě vozidla překážka, dojde ke změně trasy takovým způsobem, aby se robot překážce vyhnul. Zpočátku bude nutné navrhnout a realizovat konstrukci autonomního vozidla. Dále bude potřeba sestavit řídicí systém s využitím systému ROS (Robot Operating System) [35]. Posledním úkolem bude realizace operátorského rozhraní komunikujícího s robotem.

V následující kapitole bude popsána robotická platforma a všechny její součásti, další kapitola bude obsahovat popis Ackermanova modelu řízení, čtvrtá kapitola bude zaměřena na systém ROS, pátá kapitola bude popisovat návrh a implementaci uživatelského rozhraní pro ovládání inspekčního robota a v poslední kapitole budou shrnuty výsledky.

2 Robotická platforma

V této kapitole budou popsány všechny důležité komponenty, ze kterých je robot sestaven. Návrh řídicího systému bude následně proveden pro tuto robotickou platformu. Práce částečně využívá informace a znalosti z otevřeného projektu F1/10 [14]. Na obrázku (2.1) je kompletní robotická plat-



Obrázek 2.1: Robotická platforma

forma se všemi součástmi. Přední náprava je na tomto obrázku dole, zadní je nahoře. V přední části robota je umístěna kamera ZED, nad ní je lidar HOKUYO. Přibližně uprostřed robota je na vrchní základně umístěna GPS anténa a USB rozbočovač. V zadní části robota je namontován RPLIDAR. Podrobný popis zmíněných komponent bude obsahem dalších sekcí.

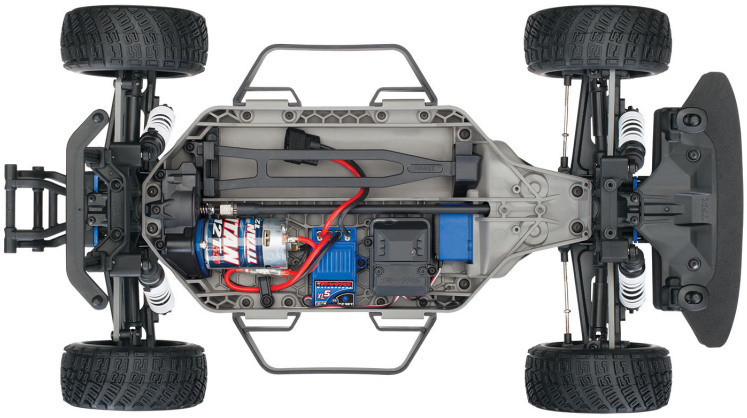
Pro sestavení funkčního robotického systému je třeba promyslet význam a parametry jednotlivých součástí. Je nutné zvážit napájení celé platformy, aby použitý aktuátor měl dostatek výkonu a mohl realizovat požadované akční zásahy. Napájena musí být také řídicí elektronika, která bude zpracovávat data ze senzorů, komunikovat s operátorským rozhraním a vysílat požadavky do aktuátorů. Výběr vhodných senzorů je dán zejména prostředím, kde se robot bude pohybovat.

2.1 Mechanická konstrukce

Tato práce byla zpracována s využitím podvozku Traxxas Ford Fiesta ST Rally [15]. Jedná se o zmenšeninu klasického auta v poměru 1:10. Všechna čtyři kola jsou odpružena a tlumena a jsou poháněna jediným motorem umístěným v zadní části podvozku. Původní stejnosměrný motor poháněj́ící auto dopředu a dozadu byl vyměněn za bezkartáčový [26], aby bylo možné měřit jeho otáčky. Byl odstraněn původní regulátor a také RC přijímač. Zatáčení předních kol je realizováno původním servomotorem.

váha	2.77 kg
šířka	281 mm
délka	535 mm
rozvor	324 mm
výška	206 mm

Tabulka 2.1: Specifikace podvozku Traxxas



Obrázek 2.2: Traxxas Ford Fiesta ST Rally

2.2 Senzory

Každý robot reagující na okolní prostředí musí být vybaven senzory. Tyto součástky mají podobnou funkci jako lidské smyslové orgány. Jelikož na každý senzor působí kromě měřené fyzikální veličiny i poruchová veličina, je vhodné robotickou platformu osadit kombinací senzorů, které využívají různé fyzikální principy. Robot má pak k dispozici více informací o okolním prostředí a může se v případě poruchy nějakého senzoru orientovat pomocí zbývajících senzorů.

IMU

Tento senzor slouží k měření rychlosti, zrychlení a orientace v prostoru. Na podvozku je namontována jednotka 3DM-CV5-25 od firmy Lord [3]. Jedná se o jednu z nejmenších a nejlehčích průmyslových IMU vyráběných technologií MEMS (Micro-Electro-Mechanical System). Je vybavena teplotně kompenzovaným tříosým akcelerometrem, gyroskopem a magnetometrem. Akcelerometr slouží především k měření zrychlení, gyroskop primárně měří rychlost otáčení a magnetometr získává informaci o magnetickém poli Země. Jednotka dále obsahuje ještě tlakový výškoměr. Přímo v této jednotce funguje algoritmus rozšířeného Kalmanova filtru, který slučuje získávaná měření a tím zpřesňuje odhad polohy a orientace v prostoru.

váha	13 g
napájení	3.2 až 5.2 V
vzorkovací frekvence	1000 Hz
rozsah měření akcelerometru	$\pm 8g$
rozsah měření gyroskopu	$\pm 500°/s$
rozsah měření magnetometru	$\pm 8G$

Tabulka 2.2: Specifikace IMU

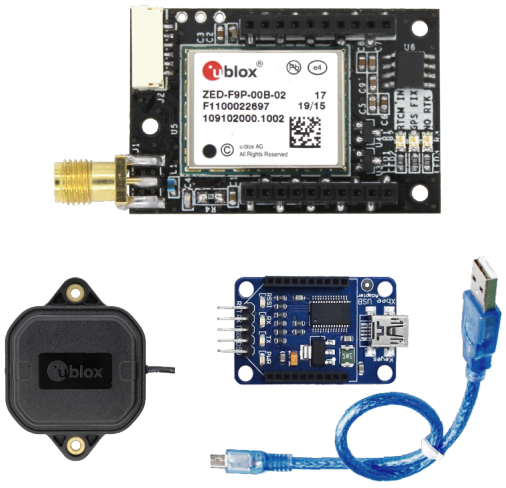
Obrázek 2.3: 3DM-CV5-25 od firmy Lord

GPS

Systém GPS je složen z množství satelitů obíhajících zeměkouli. Pro určení polohy na Zemi je nutné mít přijímač s přímým výhledem na oblohu, který sbírá informace ze satelitů a vyhodnocuje je. V tomto projektu byl využit přijímač simpleRTK2Blite [1], který využívá modul u-blox ZED-F9P [6]. Tento modul přijímá signály ze satelitních systému GPS, GLONASS, Galileo a BeiDou. Přesnost určení polohy závisí na mnoha faktorech (členitost terénu, výška nad povrchem, anténa, přítomnost budov). Zmíněný modul podporuje i tzv. přesnost RTK (v řádu centimetrů). Je k tomu ale zapotřebí přijímat korekční zprávy, což v tomto projektu nebylo řešeno. Pro přijímač simpleRTK2Blite bylo navrženo ochranné pouzdro v cloudovém CAD nástroji Onshape [4]. Následně byl tento model vytištěn na 3D tiskárně Prusa v rámci IoTlab na ZČU [20].

provozní teplota	-40°C až +85°C
napájení	2.7 až 3.6 V
vzorkovací frekvence	20 Hz
přesnost pozice	RTK (s korekcemi)
čas konvergence	<10 s
přijímané typy	GPS, GLONASS, Galileo a BeiDou

Tabulka 2.3: Specifikace u-blox ZED-F9P



Obrázek 2.4: Přijímač simpleRTK2Blite

Kamera

Robotická platforma má v přední části umístěnou kameru ZED [5]. Jedná se o stereoskopickou kameru poskytující nejen obraz, ale také hloubku (vzdálenost obrazových bodů). Tento přístup je převzat z lidského vnímání obrazu. Hloubka je z kamery dopočítána díky tomu, že je scéna nasnímaná ze dvou různých pozorovacích bodů (dvou kamer). Použitím triangulace pak lze dopočítat vzdálenost mezi pozorovaným objektem a kamerou [56]. Aby byla hloubka počítána správně, musí se kamera nejprve zkalibrovat. To lze provést jednoduše pomocí SDK dodávaného výrobcem. Kamera je připojená přes USB 3.0 a kvůli značnému odběru musela být robotická platforma doplněna o USB rozbočovač s externím napájením.

provozní teplota	0°C až +45°C
váha	170 g
ohnisková vzdálenost	2.8 mm - f/2.0
vzdálenost kamer	120 mm
výstupní rozlišení	2x (2208x1242) @15fps
	2x (1920x1080) @30fps
	2x (1280x720) @60fps
	2x (672x376) @100fps
rozsah měření hloubky	0.5 m až 25 m
přesnost měření hloubky	$< 2\%$ pro 3 m
	$< 4\%$ pro 15m
RGB senzor	1/3” 4MP CMOS

Tabulka 2.4: Specifikace kamery ZED

Obrázek 2.5: Kamera ZED

Lidar

Na robotické platformě jsou umístěné dva lidary. V přední části nad kamerou ZED je lidar HOKUYO URG-04LX-UG01 (dále jen HOKUYO)[2]. V zadní části nad ostatními součástmi je lidar RPLIDAR A3 [39]. Lidar slouží k měření vzdáleností od překážek. Základní princip spočívá v měření času vyslaného a přijatého laserového paprsku, který se odrazil od překážky v prostoru. Výsledkem měření je mračno bodů, které může sloužit například k tvorbě mapy prostoru [51].

Výhodou lidaru je, že měří vzdálenosti s větší přesností než například kamera ZED. Nevýhodou je, že se laserový paprsek neodrazí od skleněné stěny nebo od kovových nohou židle, takže jsou tyto překážky pro lidar neviditelné.

rozsah měření	20 až 5600 mm, 240°
přesnost měření	pro 60 až 1000 mm : $<$ 30 mm
	$<$ 3% pro 1000 až 4095 mm
úhlové rozlišení	0.36 °
čas jednoho skenu	100 ms
připojení	USB2.0
použití	vnitřní prostory

Tabulka 2.5: Specifikace HOKUYO

Obrázek 2.6: Lidar HOKUYO

RPLIDAR A3 byl přidán na robotickou platformu z důvodu většího rozsahu a možnosti použití ve venkovním prostředí. Později se ukázalo, že rozsah je sice větší, ale poměrně často dochází k vytváření falešných bodových překážek. Lidar HOKUYO byl později také testován ve venkovním prostředí a falešné překážky se v jeho omezeném rozsahu nevyskytovaly.

rozsah měření	20 m, 360°
úhlové rozlišení	0.54 °
rychlost skenování	15 Hz
připojení	TTL UART
použití	vnitřní i venkovní prostory

Tabulka 2.6: Specifikace RPLIDAR A3

Obrázek 2.7: RPLIDAR A3

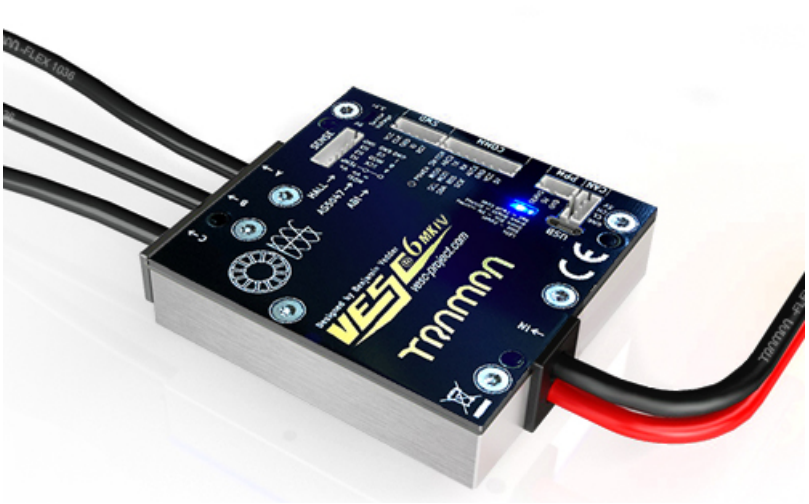
2.3 Řídicí elektronika

Řídicí elektronika slouží ke zpracování dat ze senzorů, výpočtu akčního zásahu a nakonec i k jeho realizaci prostřednictvím aktuátorů.

K řízení bezkartáčového motoru pro pohyb vpřed a servomotoru pro zatáčení předních kol byla použita řídicí deska VESC 6 MKIV (dále jen VESC). Jedná se o řadič motorů založený na otevřeném projektu [43]. Na internetu lze nalézt zdrojové kódy, schémata zapojení i software pro nastavení. VESC je schopen měřit napětí i proud na všech fázích motoru a poskytnout informaci o rychlosti otáčení. V ovládacím programu je možno provést identifikaci motoru a nastavit parametry pro jeho řízení, které je realizováno PID regulátorem.

napájení	11.1 až 60 V
proudová zatížitelnost	80 A
výstupy	5 V 1 A
	3.3 V 0.5 A
připojení	USB
použití	stejnosměrné i střídavé motory

Tabulka 2.7: Specifikace VESC



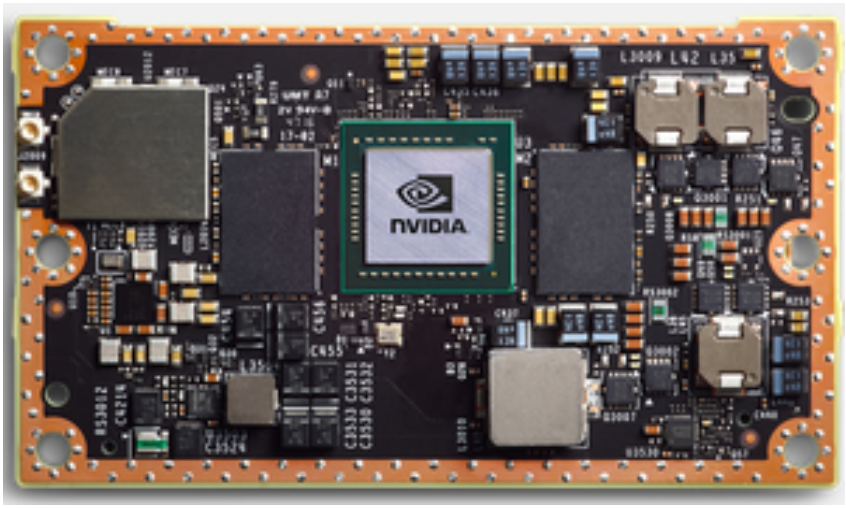
Obrázek 2.8: Řídicí deska VESC

V této práci byl robot řízen přes mikropočítač sestavený z modulu Jetson TX2 a nosné desky Orbitty Carrier. Nejprve bylo nutné na mikropočítač nainstalovat NVIDIA JetPack SDK (konkrétně verzi 4.2) a dále operační systém Linux Ubuntu 18.04. Tato varianta byla zvolena proto, aby bylo možné využít softwarový rámec ROS, který umožní použití volně dostupných nástrojů vhodných pro lokalizaci a navigaci inspekčního robota.

GPU	NVIDIA Pascal™, 256 CUDA jader
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2
video	4K x 2K 60 Hz
rozměry	50 x 87 mm
paměť	8 GB 128 bit LPDDR4 59.7 GB/s
displej	2x DSI, 2x DP 1.2 / HDMI 2.0 / eDP 1.4
připojení	1 Gigabit Ethernet, 802.11ac WLAN, Bluetooth
úložiště	32 GB eMMC, SDIO, SATA

Tabulka 2.8: Specifikace modulu Jetson TX2 [21]

Modul Jetson TX2 má dostatek výpočetního výkonu i pro zpracování obrazu z kamery. V budoucnu by tedy bylo možné například rozpoznávat předměty okolo robota.

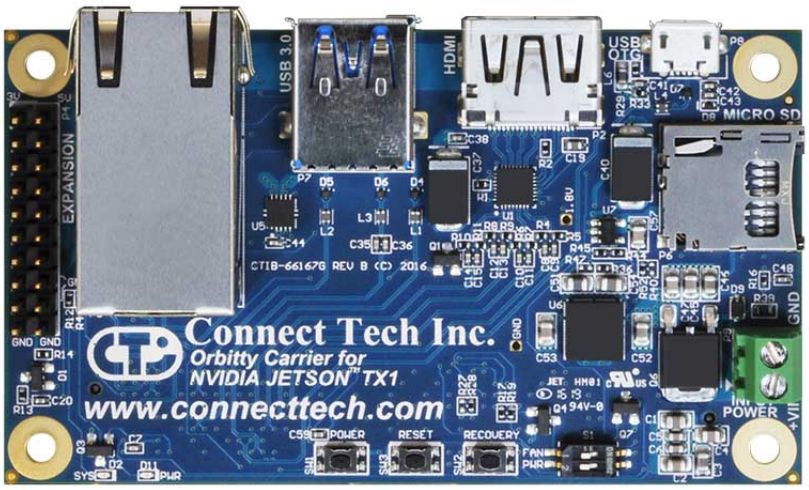


Obrázek 2.9: Modul Jetson TX2

Modul Jetson TX2 je možné zapojit pouze přes speciální konektor. Originálně je modul dodáván ve vývojové sadě s větší nosnou deskou, která mu zprostředkovává klasické vstupně-výstupní rozhraní. Z důvodu ušetření místa na robotické platformě byla využita náhrada v podobě menší nosné desky Orbitty Carrier. Tato nosná deska zprostředkovává modulu Jetson TX2 napájení a klasické vstupně-výstupní rozhraní.

Napájení	9 až 14 V
USB	1x USB 3.0
připojení	1x Gigabit Ethernet (10/100/1000)
rozměry	50 x 87 mm
SD karta	1x microSD
displej	1x HDMI
sériová komunikace	2x 3.3V UART

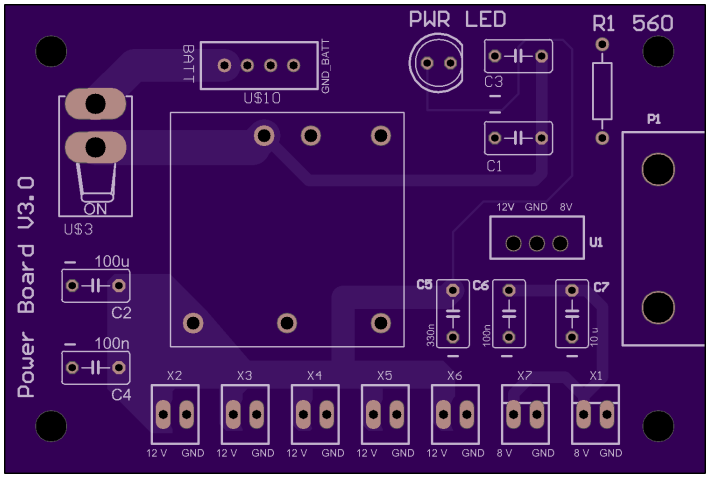
Tabulka 2.9: Specifikace nosné desky Orbitty Carrier [32]



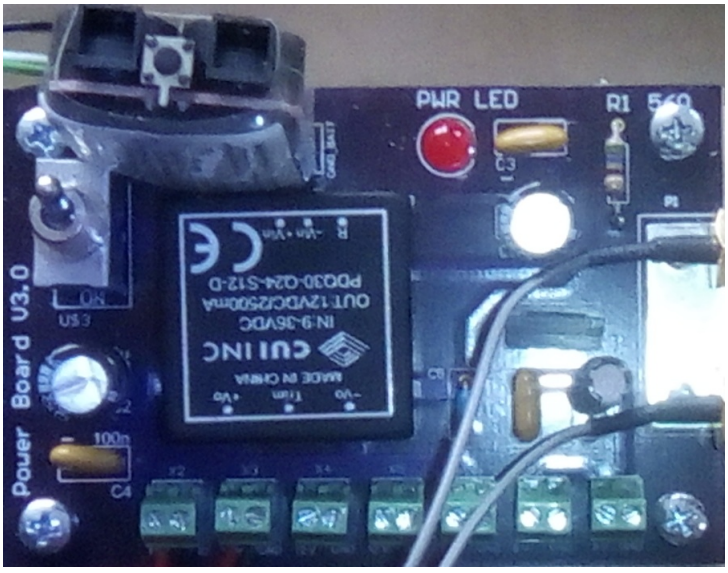
Obrázek 2.10: Nosná deska Orbitty Carrier

2.4 Napájecí obvod

Pro napájení mikropočítače a USB rozbočovače byl na robotickou platformu přidán napájecí obvod z projektu F1/10. Tento napájecí obvod dodává stabilizovaná napětí s hodnotou 12 a 8 V. Plošný spoj byl osazen součástkami dle schématu a dále byl přidán obvod pro kontrolu stavu baterie, který při nízkém napětí na článku vydává výstražný signál. Řídicí deska VESC i napájecí obvod je připojen k tříčlánkové LiPo baterii o kapacitě 6500 mAh.



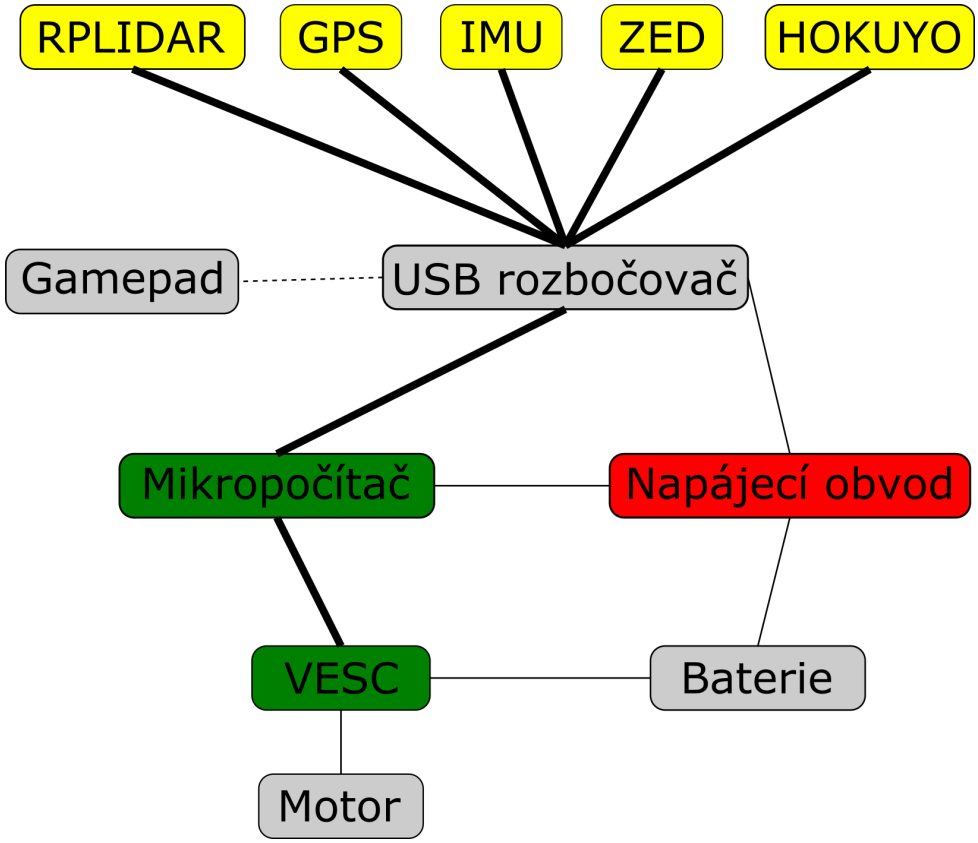
Obrázek 2.11: Návrh plošného spoje



Obrázek 2.12: Osazený napájecí obvod

2.5 Propojení komponent

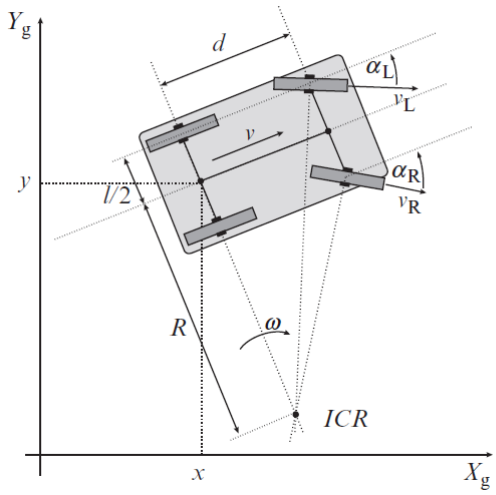
Na obrázku (2.13) je schématicky znázorněno propojení všech komponent na robotické platformě. Žlutě jsou označeny všechny senzory, zelenou barvu mají řídicí desky a červeně je zvýrazněn napájecí obvod. Gamepad je bezdrátový a do USB rozbočovače je zapojen bluetooth modul, proto je spojení znázorněno čárkovanou čarou. Tlustá čára značí propojení přes USB, tenké čáry znázorňují napájecí kabely.



Obrázek 2.13: Schéma zapojení komponent na robotické platformě

3 Ackermanův model řízení

V této práci byl použit podvozek Traxxas Ford Fiesta STRally. Jedná se o zmenšeninu klasického auta, proto lze mluvit o tzv. Ackermanově podvozku, který je specifický tím, že při průjezdu zatáčkou jsou přední kola natočena pod různým úhlem (obrázek (3.1)). Tím pádem nedochází při zatáčení ke smyku kol a pneumatiky a zavěšení kol nejsou tolik namáhány. Nevýhodou automobilového podvozku je to, že se nelze otočit na místě.



Obrázek 3.1: Ackermanův podvozek v souřadném systému [53]

Úhel natočení předních kol lze vypočítat pomocí goniometrických funkcí:

$$\alpha_L = \pi/2 - arctan\frac{R + l/2}{d} \tag{3.1}$$

$$\alpha_R = \pi/2 - arctan\frac{R - l/2}{d} \tag{3.2}$$

kde α_L je úhel natočení levého kola, α_R je úhel natočení pravého kola, R je poloměr zatáčky, l je šířka vozidla a d je vzdálenost mezi přední a zadní nápravou. Obvodová rychlost předních kol je definována jako:

$$v_L = \omega(R + l/2) \tag{3.3}$$

$$v_R = \omega(R - l/2) \tag{3.4}$$

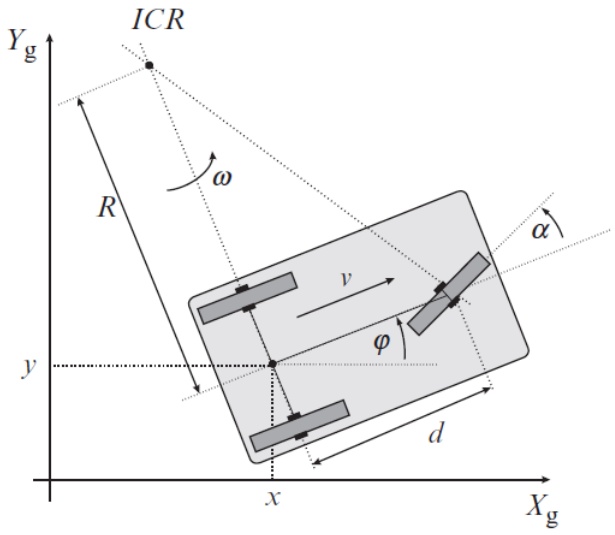
kde v_L je obvodová rychlost levého kola, v_R je obvodová rychlost pravého kola, R je poloměr zatáčky, l je šířka vozidla a ω je rychlost otáčení okolo bodu ICR (instantaneous center of rotation=okamžitý střed otáčení).

3.1 Dopředná kinematická úloha

Cílem dopředné kinematické úlohy je určit polohu a natočení robota na základě znalosti pohybu jeho kol. Pro jednoduchost lze přední kola nahradit jediným kolem umístěným doprostřed přední nápravy. Skutečný podvozek je také ovládán pouze jedním servomotorem a různé natočení předních kol je realizováno lichoběžníkovou nápravou. Úhel natočení jediného předního kola lze vyjádřit takto:

$$\alpha = \pi/2 - arctan\frac{R}{d} \tag{3.5}$$

kde α je úhel natočení předního kola, R je poloměr zatáčky a d je vzdálenost mezi přední a zadní nápravou. Potom je kinematický model Ackermanova podvozku stejný jako kinematický model tříkolky (obrázek (3.2)).



Obrázek 3.2: Tříkolka v souřadném systému [53]

Dopředná kinematická úloha je definována těmito vztahy:

$$\dot{x} = v_s(t)cos(\alpha(t))cos(\varphi(t)) \tag{3.6}$$

$$\dot{y} = v_s(t)cos(\alpha(t))sin(\varphi(t)) \tag{3.7}$$

$$\dot{\varphi} = \frac{v_s(t)}{d}sin(\alpha(t)) \tag{3.8}$$

kde $v = v_s(t)cos(\alpha(t))$ je dopředná rychlost robota, $\omega = \frac{v_s(t)}{d}sin(\alpha(t))$ je rychlost otáčení robota okolo bodu ICR, v_s je obvodová rychlost předního kola, d je vzdálenost mezi přední a zadní nápravou, α je úhel natočení předního kola, φ je úhel natočení robota, x je pozice robota ve směru osy x a y je pozice robota ve směru osy y.

Řízené proměnné jsou dopředná rychlost $v_r(t) = v_s(t)cos(\alpha(t))$ a úhel natočení předních kol $\alpha(t)$. Dopředná kinematická úloha má tvar:

$$\dot{x} = v_r(t)cos(\varphi(t)) \tag{3.9}$$

$$\dot{y} = v_r(t)sin(\varphi(t)) \tag{3.10}$$

$$\dot{\varphi} = \frac{v_r(t)}{d}tan(\alpha(t)) \tag{3.11}$$

Použitím maticového zápisu získáme:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\varphi} \end{bmatrix} = \begin{bmatrix} cos(\varphi(t)) & 0 \\ sin(\varphi(t)) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_r(t) \\ \omega(t) \end{bmatrix}. \tag{3.12}$$

kde $\omega = \frac{v_r(t)}{d}tan(\alpha(t))$ je rychlost otáčení robota okolo bodu ICR, d je vzdálenost mezi přední a zadní nápravou, φ je úhel natočení robota, x je pozice robota ve směru osy x a y je pozice robota ve směru osy y.

Dopředná kinematická úloha je implementována v C++ následujícím způsobem, kde diferenciální rovnice byly diskretizovány Eulerovou metodou.

```
double current_speed = ( state->state.speed -
    speed_to_erpm_offset_ ) / speed_to_erpm_gain_;
double current_steering_angle(0.0), current_angular_velocity(0.0);
if (use_servo_cmd_) {
    current_steering_angle =
      ( last_servo_cmd_->data - steering_to_servo_offset_ ) /
          steering_to_servo_gain_;
    //rychlost otaceni robota
    current_angular_velocity = current_speed *
       tan(current_steering_angle) / wheelbase_;
}
if (!last_state_)
    last_state_ = state;
ros::Duration dt = state->header.stamp - last_state_->header.stamp;
//vypocet rychlosti
double x_dot = current_speed * cos(yaw_);
double y_dot = current_speed * sin(yaw_);
//vypocet polohy a uhlu natoceni
x_ += x_dot * dt.toSec();
y_ += y_dot * dt.toSec();
if (use_servo_cmd_)
    yaw_ += current_angular_velocity * dt.toSec();
last_state_ = state;
```

3.2 Inverzní kinematická úloha

Cílem inverzní kinematické úlohy je určit, jak rychle se musí otáčet kola a jak mají být natočena, aby se robot pohyboval určenou rychlostí vpřed případně zatáčel. Konkrétně musíme znát vztahy určující otáčky bezkartáčového motoru a úhel natočení servomotoru. Tato úloha je obtížnější než dopředná kinematická úloha a nemusí jít vždy řešit analyticky. Proto bylo použito řešení dle [19].

Požadované otáčky bezkartáčového motoru lze vypočítat ze známé dopředné rychlosti robota pronásobené multiplikativní konstantou (zahrnuje průměr kola, převody). Tato konstanta byla zjištěna experimentálně. Pro úhel natočení servomotoru jsou vztahy komplikovanější. Předpokládejme, že požadujeme nulovou rychlost otáčení robota ($\omega = 0$), potom úhel natočení předních kol musí být také nulový ($\alpha = 0$).

Pokud automobil zatáčí, pak poloměr zatáčky R odpovídá vztahu:

$$R = \frac{v}{\omega} \tag{3.13}$$

kde v je dopředná rychlost robota, ω je rychlost otáčení robota.

Úhel natočení předních kol α lze pak vypočítat:

$$\alpha = arctan\frac{d}{R} \tag{3.14}$$

kde d je vzdálenost mezi nápravami a R je poloměr zatáčky.

Pro nulovou dopřednou rychlost ($v = 0$) lze úhel natočení předních kol nastavit na nulový úhel ($\alpha = 0$).

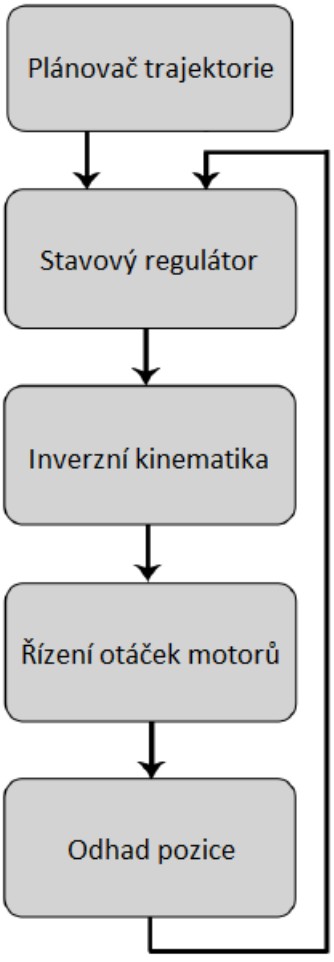
Inverzní kinematická úloha je implementována v následující části zdrojového kódu v jazyce Python.

```
def convert_ang_vel_to_steering_angle(v, omega):
  if omega == 0 or v == 0:
    return 0
  radius = v / omega
  return math.atan(wheelbase / radius)
```

Po sestavení robotické platformy byla ověřena základní funkčnost dopředné i inverzní kinematické úlohy. Jelikož se jedná o reálný systém, na který působí poruchy a vztahy byly odvozeny pro ideální systém, bylo nutné do výrazů zahrnout kalibrační konstanty (multiplikativní konstanta pro převod dopředné rychlosti robota na požadované otáčky motoru, aditivní konstanta pro nastavení nulové pozice servomotoru).

3.3 Využití Ackermanova modelu řízení

Aby mohl být robot vyslán na nějaké místo, které je potřeba prozkoumat, musí dobře sledovat naplánovanou trajektorii k určenému místu. O vygenerování trajektorie se stará plánovač trajektorie. Následně je definováno, kde má robot být v každém časovém okamžiku. Porovnáním požadované pozice a aktuální pozice robota je získána regulační odchylka. Tato odchylka je vstupem pro stavový regulátor, který vytvoří požadavek na dopřednou rychlost a rychlost otáčení celého robota. Inverzní kinematickou úlohou se tyto požadavky přepočítají na otáčky motoru a úhel natočení předních kol. Následně se vypočte odhad pozice robota pomocí dopředné kinematické úlohy, který se opět porovná s požadovanou pozicí robota definovanou plánovačem trajektorie. Jedná se tedy o kaskádní regulaci, kde vnitřní smyčka reguluje otáčky motoru a nadřazená smyčka řídí polohu robota.



Obrázek 3.3: Blokové schéma řídicího systému

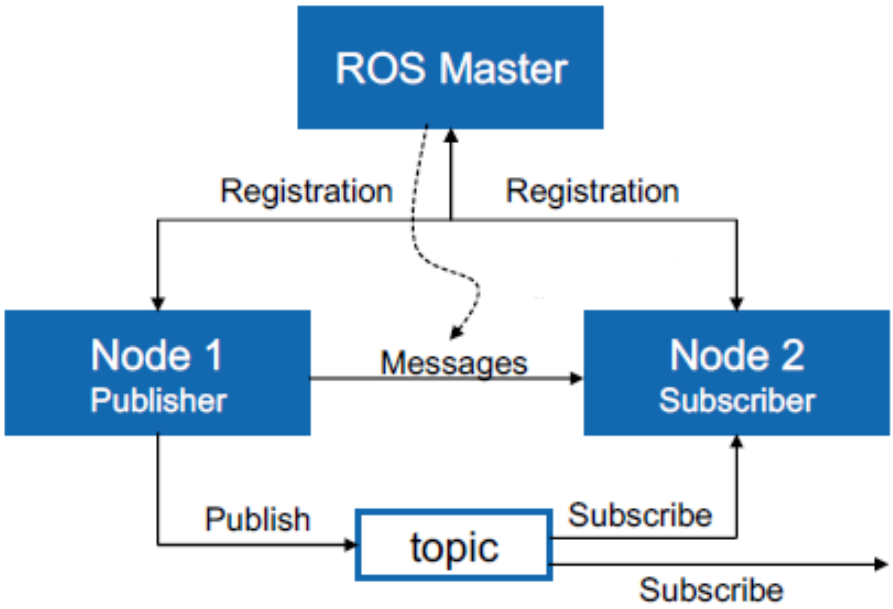
4 ROS

K realizaci řídicího systému byl vybrán softwarový rámec ROS (Robot Operating System). ROS je v dnešní době velmi rozšířeným nástrojem pro vývoj robotických systémů. Jeho hlavní výhodou je množství volně dostupných nástrojů, knihoven a aplikací. ROS vznikl v roce 2007 ve Stanford Artificial Intelligence Laboratory. Dnes je pod správou Open Source Robotics Foundation [55].

ROS je volně dostupný ke komerčním a vědeckým účelům. Je možné ho volně stahovat, upravovat a sdílet. Existuje několik verzí, v této práci byla použita distribuce ROS Melodic Morenia (dále jen Melodic).

4.1 Architektura ROS

ROS využívá stromovou strukturu. Kořenový uzel je realizován tzv. ROS Masterem, který zajišťuje registraci dalších uzlů (Nodes) a zpráv. V systému ROS se uzly rozlišují podle toho, zda zprávy odesílají (Publisher) nebo přijímají (Subscriber). Zprávy mají v systému ROS definovovanou datovou strukturu a název (topic). Určitou zprávu může přijmout více uzlů [12].



Obrázek 4.1: Schéma základního propojení uzlů

ROS Parameter Server je sdílený slovník, který je přístupný přes síťové rozhraní. Slouží ke změně parametrů uzlů za běhu. Je implementován pomocí XMLRPC. V rámci svojí bakalářské práce jsem tento nástroj použil například pro nastavení parametrů PI regulátoru [33].

V systému ROS existuje speciální komunikační model request / reply, který využívá tzv. services, které jsou definovány pomocí dvojice zpráv (jedna pro žádost a jedna pro odpověď).

Systém ROS nabízí unikátní souborový formát tzv. bag. Tento formát slouží k uložení a opětovnému přehrání všech zaznamenaných zpráv. Používá se zejména k vývoji a testování nových algoritmů [9].

Pro vývoj vlastních nástrojů a programů v systému ROS je třeba založit tzv. workspace. Tento pracovní prostor obsahuje složku se zdrojovými soubory (src), složku s uloženou mezipamětí a informacemi o konfiguraci (build) a složku s kompilovaným programem (devel). Složky build a devel se vytvářejí automaticky při zavolání příkazu `catkin_make` v terminálu nad složkou workspace. Složka src se dále dělí do tzv. package (balíků). Každý package je vlastně složka, která obsahuje samotný zdrojový kód, soubor CMakeLists.txt a package.xml. Uvedený obsah balíku je základem, ale v package můžou být i další soubory. Například s příponou **launch** pro spouštění více uzlů najednou nebo vlastní definice zpráv s příponou **msg**.

```
workspace_folder/
  src/
    CMakeLists.txt
    package_1/
      CMakeLists.txt
      package.xml
    ...
    package_n/
      CMakeLists.txt
      package.xml
```

Obrázek 4.2: Struktura workspace [44]

ROS podporuje především programovací jazyky C++ a Python, ale existuje mnoho dalších knihoven pro práci v jiných jazycích. Po vytvoření určitého programu (uzlu) a jeho zkompilování (`catkin_make`) ho lze z terminálu spustit příkazem `rosrun`. Uzly systému ROS mohou běžet i na několika strojích zároveň (například na stolním počítači je zobrazena vizualizace, robot slouží k přijímání dat a zpracování může probíhat na serveru).

4.2 Implementace základního propojení uzlů

V této části bude popsáno vytvoření nejjednodušších komunikujících uzlů v jazyce C++ dle [45].

Ve zdrojovém kódu každého komunikujícího uzlu musí být definovaný nejprve tzv. *NodeHandle*. Tento objekt reprezentuje konkrétní uzel. Pro příjem zpráv se používá objekt *Subscriber*. U objektu *Subscriber* je nutné stanovit název (topic) přijímaných zpráv, maximální délku fronty a musí se definovat `Callback()`, což je metoda, která zprávu zpracuje. Pro výpis textu do terminálu pomocí programovacího jazyka C++ se používá funkce `ROS_INFO()`. Uzel je třeba na závěr aktivovat. Zajistit to lze například příkazem `ros::spin()`. Příklad jednoduchého uzlu pro zachytávání a výpis textových řetězců je uveden ve zdrojovém kódu níže.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000,
      chatterCallback);
  ros::spin();
  return 0;
}
```

Pro posílání zpráv se používá objekt *Publisher*, u kterého se musí stanovit název (topic) odesílaných zpráv, jejich typ a počet, který se uchová v paměti. Objekt *Publisher* musí zavolat metodu `publish()` s parametrem zprávy, aby došlo k odeslání. Spouštět odesílání lze s nějakou konkrétní frekvencí, neustále nebo pouze jednou.

Aby se v systému ROS vytvořil spustitelný uzel, je nutné vhodně upravit konfigurační soubory CMakeLists.txt a package.xml. Po úspěšné kompilaci zdrojových kódů příkazem `catkin_make` je možné jednotlivé uzly spustit příkazem `rosrun`. Spuštění více uzlů najednou a jádra systému ROS se provádí příkazem `roslaunch` (musí být definován spouštěcí soubor s příponou launch).

4.3 Moduly pro lokalizaci a navigaci

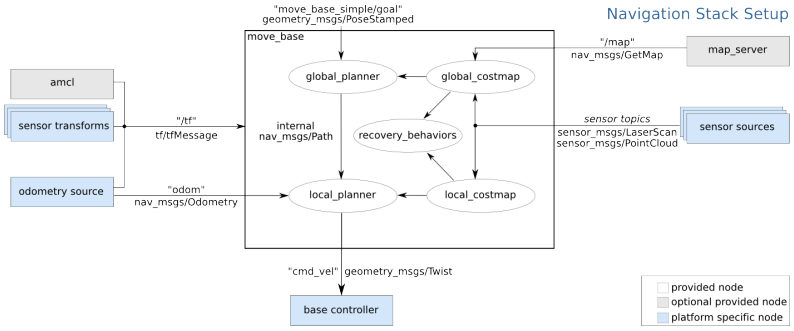
V této části budou popsány některé volně dostupné nástroje systému ROS, které byly využity v rámci vývoje inspekčního vozidla. Jedná se především o komplexní modul *Navigation stack*, který zajišťuje plánování trajektorie, detekci překážek a jejich objíždění. Pro získání mapy prostředí, kde se robot pohybuje, bylo otestováno několik přístupů. Mezi nejlepší lze řadit modul *Gmapping*. Pro lokalizaci robota ve známé mapě byl použit volně dostupný modul *AMCL*.

4.3.1 Navigation stack

Tento modul je základem pro autonomní navigaci robotů využívajících ROS. Celkovou funkčnost lze popsat tak, že vstupem do *Navigation stack* jsou odometrická data, lidarová data, mapa prostředí a požadovaná cílová pozice robota, výstupem jsou rychlost otáčení a dopředná rychlost, které robota směřují k cíli v mapě tak, aby se vyhnul překážkám.

Pro zprovoznění tohoto nástroje je potřeba znát kinematický model robota (viz kapitola Ackermanův model řízení) a s jeho pomocí publikovat odometrická data. Dále je nutné používat senzor, který rozpozná překážky v okolí robota. *Navigation stack* je schopný přeplánovat aktuálně navrženou trajektorii, pokud se v cestě vyskytne neznámý objekt. Robot musí být připraven přijímat zprávy o požadované rychlosti a převádět je na reálný pohyb podvozku. Poslední nezbytností je zavedení a publikace transformací mezi souřadnými systémy [29].

Na obrázku (4.3) je schéma knihovny *Navigation stack* i s požadovanými komponentami z dalších součástí systému ROS. Jádro *Navigation stack* je značeno obdélníkem s názvem *move base* a bude popsáno v dalších sekcích. Modré komponenty jsou pro fungování *Navigation stack* nezbytné. Šedé komponenty jsou pouze doplňkové.



Obrázek 4.3: Schéma *Navigation stack* [42]

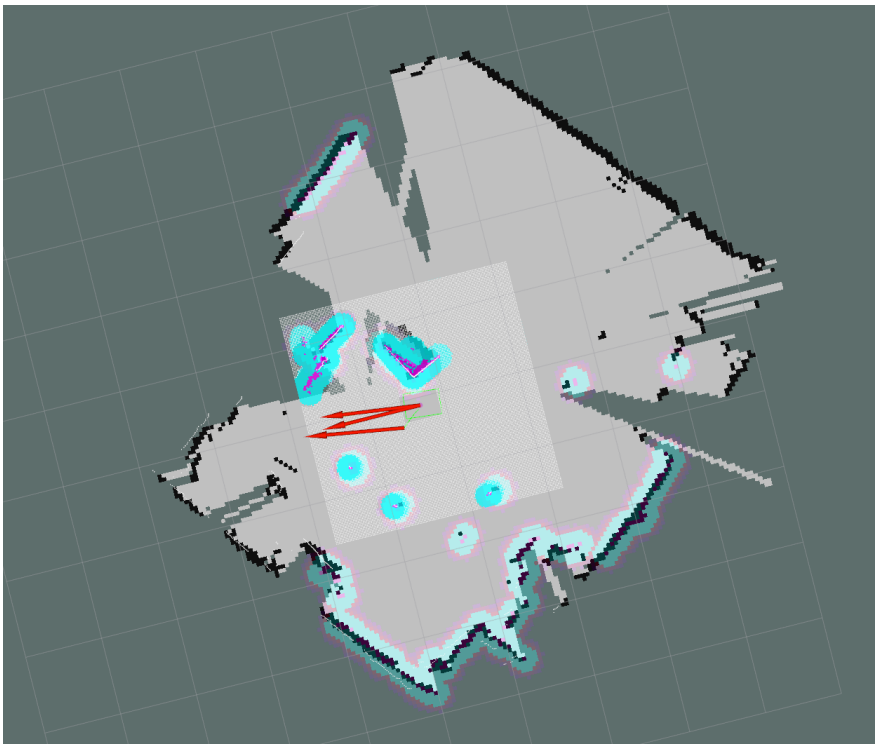
Costmap

Jedná se o datovou reprezentaci, která poskytuje 2D mapu, ve které jsou jednotlivým bodům přiřazeny váhy. *Navigation stack* využívá globální a lokální *costmap*. Na obrázku (4.4) jsou vidět obě i s aktuální pozicí robota (červené šipky).

Lokální *costmap* je oblast v definované blízkosti robota (v obrázku šrafovaný). Pro vytváření lokální *costmap* se využívají data například z lidaru, protože poskytují aktuální informaci o překážkách v těsném okolí robota.

Globální *costmap* zahrnuje kompletní prostředí, ve kterém se robot pohybuje (v obrázku ohraničeno černou barvou). Obvykle je získána například pomocí SLAM algoritmů, kterým bude věnována další kapitola. Globální *costmap* slouží k plánování globální trajektorie robota, protože zachycuje pevné nepohybující se překážky.

Body v mapě lze podle ceny rozdělit do pěti kategorií. Smrtící (lethal) jsou body na mapě, kde je skutečná překážka. Tyto body mají největší cenu. Další úroveň jsou inscribed body. Jsou dál od skutečné překážky, ale pořád se jedná o kritickou oblast. Následuje oblast hraniční. Volný prostor má nejmenší cenu. Robot se v něm může bez problémů pohybovat. Poslední kategorií jsou neznámé body [13].



Obrázek 4.4: Zobrazení *costmap* v testovaném prostředí

Global planner

Tato součást *Navigation stack* slouží k plánování trajektorie z aktuální pozice robota až na požadovanou cílovou pozici. Globální plánovač využívá globální *costmap*. K dispozici jsou v systému ROS tři různé globální plánovače: carrot planner, navfn a global planner. Dále je možné si vytvořit vlastní plánovač, který bude připojen k rozhraní `nav_core::BaseGlobalPlanner` [18].

Nejjednodušší z nabízených globálních plánovačů je *carrot planner*. Tento plánovač lze úspěšně použít v otevřeném prostoru, protože spojí aktuální pozici robota a dovede ho přímou trajektorií k cíli. V případě, že je cílová pozice umístěna do prostoru nějaké překážky, plánovač robota dovede na hranici této překážky. V komplikovaných prostorech s množstvím překážek tento plánovač není příliš vhodný [11].

Plánovač *navfn* je založen na přístupu popsaném v [47]. K výpočtu navigační funkce používá Dijkstrův algoritmus [28]. Tento globální plánovač je schopen navigovat robota i v komplikovaném prostředí okolo překážek.

Globální plánovač *global planner* je více flexibilní než *navfn* a je schopný trajektorii naplánovat okolo překážek. Může využívat buď Dijkstrův algoritmus nebo algoritmus A* [16]. V základním nastavení *Navigation stack* používá právě tento globální plánovač a byl použit i v tomto projektu.

Local planner

Lokální plánovač využívá globálně navrženou trajektorii, lokální *costmap* a produkuje požadovanou rychlost otáčení a dopředného pohybu. K dispozici jsou v systému ROS různé lokální plánovače, také je možné si vytvořit vlastní plánovač, který bude připojen k rozhraní `nav_core::BaseLocalPlanner`.

Base_local_planner implementuje algoritmus DWA (Dynamic Window Approach) a Trajectory Rollout. Tento plánovač podporuje holonomní i neholonomní roboty.

Dwa_local_planner také implementuje algoritmus DWA, ale zdrojový kód by měl být čitelnější. Poskytuje lepší vlastnosti pro řízení holonomních robotů.

Eband_local_planner implementuje metodu Elastic Band. Je navržen pro holonomní roboty.

Teb_local_planner implementuje metodu Timed-Elastic-Band, která optimalizuje lokální trajektorii s ohledem na dobu jejího provádění, vzdálenosti robota od překážek a dodržování kinodynamických omezení. Tento plánovač lze použít i pro roboty s Ackermanovým podvozkem a také byl zvolen jako lokální plánovač v tomto projektu.

Mpc_local_planner implementuje různé přístupy prediktivního řízení [18].

Move base

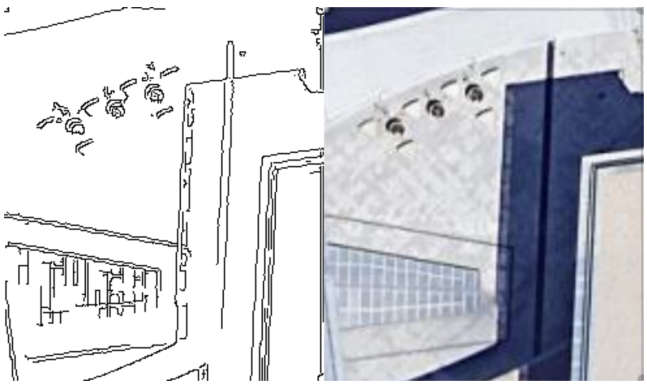
Tato komponenta je klíčová pro fungování celého nástroje *Navigation stack*. Propojuje globální a lokální plánovač trajektorie, stará se o udržování globální a lokální *costmap* a poskytuje implementaci `SimpleActionServer`, která slouží k monitorování stavu požadovaného cíle (hotovo, zrušeno, chyba, v procesu atd.) [27].

Někdy se může stát, že selže autonomní navigace robota vlivem rychlého pohybu okolních překážek, ztráty lokalizace v mapě atd. Tento stav je v *Navigation stack* řešen tzv. *recovery behaviors*. Konkrétní postup v takovém případě lze nastavit pomocí již připravených akcí (aktualizovat lokální *costmap*, otočit se na místě) nebo lze napsat vlastní plugin, který splní `nav_core::RecoveryBehavior`. Jelikož Ackermanův podvozek není schopen otočení na místě a jiný pohyb je riskantní, nebylo v tomto projektu nastaveno žádné *recovery behaviors*.

4.3.2 Map server

Autonomní navigace s využitím *Navigation stack* je založena na tom, že známe mapu prostředí, ve kterém se má robot pohybovat. Jedním ze způsobů, jak poskytnout tuto mapu, je použití nástroje *map server* [23]. Mapa je v tomto případě definována dvěma soubory. V souboru YAML jsou uložena metadata (název obrázku, rozlišení, počáteční souřadnice atd.) a obrázek zaznamenává překážky (černé pixely značí překážku, bílé volný prostor a zbytek je neznámý). Obvykle se k vytvoření mapy používá nějaký SLAM algoritmus (bude popsáno dále) a nástroj *map server* nabízí možnost uložit právě vytvořenou mapu pro příští misi.

V rámci tohoto projektu byla vyzkoušena metoda, kde mapa pro autonomní navigaci robota byla vytvořena ze satelitního snímku u budovy FAV.



Obrázek 4.5: Vytvořená mapa překážek a původní satelitní snímek

Pro nalezení překážek v satelitním snímku byl použit Cannyho hranový detektor z knihovny OpenCV. Z obrázku (4.5) je zřejmé, že veliký problém způsobuje kvalita původního satelitního snímku (stín, špatné rozlišení atd.). Problém je také v tom, že některé povrchy jsou z nadhledu špatně odlišitelné a jejich výška nad terénem není známá. Proto byly pro navigaci robota použity mapy vytvořené pomocí SLAM.

4.3.3 SLAM

Problém SLAM (simultánní lokalizace a mapování) je velice náročný, protože pro správnou lokalizaci robota je potřeba mít bezchybnou mapu prostředí a pro získání dobré mapy je zase potřeba znát přesnou pozici robota. ROS nabízí mnoho volně dostupných modulů se SLAM algoritmy, v rámci této práce byl otestován SLAM z lidarových dat a SLAM využívající data ze stereoskopické kamery.

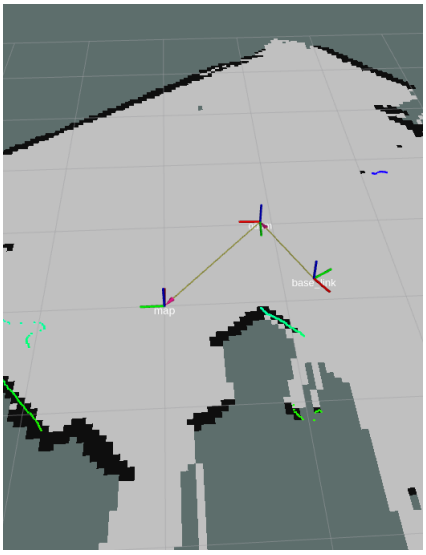
Gmapping

Tento modul vytváří z lidarových a odometrických dat 2D mapu prostředí, kde se robot pohybuje.

Gmapping probíhá v reálném čase a využívá tzv. částicový filtr. Každá částice reprezentuje jinou hypotézu trajektorie robota. Jakmile se objeví nová informace o změně odometrie, vytvoří se pro každou částici apriorní odhad pozice robota. Vychází se přitom z předchozí pozice robota (pohybový model). Potom se provede tzv. scan-matching v okolí prvotního odhadu pozice robota (pozorovací model). V tomto kroku se využijí lidarová data. Apriorní odhad pozice robota se tím výrazně zpřesní. Následně se aproximuje aktuální pravděpodobnostní rozložení pozice robota Gaussovým rozložením (musí se vzorkovat, spočítat střední hodnota a kovariance). Vybere se nejpravděpodobnější pozice robota (pro každou částici) a přepočítají se váhy všech částic. Následně se aktualizuje mapa a zkontroluje se, jestli má proběhnout převzorkování, při kterém jsou vymazány nejméně pravděpodobné částice [50].

Gmapping pro svoji funkci potřebuje správně nakonfigurovat transformace souřadnic (pomocí knihovny *tf*). Konkrétně se jedná o transformaci ze souřadnicového systému lidaru (LASER) na souřadnicový systém podvozku robota (BASE_LINK). Tato transformace je statická, protože lidar je pevně připevněn k robotovi. Statickou transformaci lze definovat ve stejném souboru, který spouští množinu uzlů. Další požadovaná transformace je z odometrického rámce (ODOM) na podvozek robota (BASE_LINK). Tato

transformace vyjadřuje pohyb robota po povrchu, takže není statická. Je definována pomocí objektu *TransformBroadcaster*, který poskytuje metodu `sendTransform()`. Tato metoda odesílá data získaná pomocí Ackermanova modelu řízení. Příkazem `tf_monitor` lze ověřit správnou funkčnost konkrétní transformace. *Gmapping* vytváří mapu jako zprávu s názvem */map* a přidává další tranformaci souřadnic mezi mapu (MAP) a odometrii (ODOM) [17]. V systému ROS je možné zobrazit všechny transformace souřadnic v jednom schématu příkazem `rosrun tf view_frames`, další možnost je zobrazit aktuální polohu všech souřadnicových rámců v prostředí RViz (obrázek (4.6)).



Obrázek 4.6: Zobrazení souřadných rámců v prostředí RViz

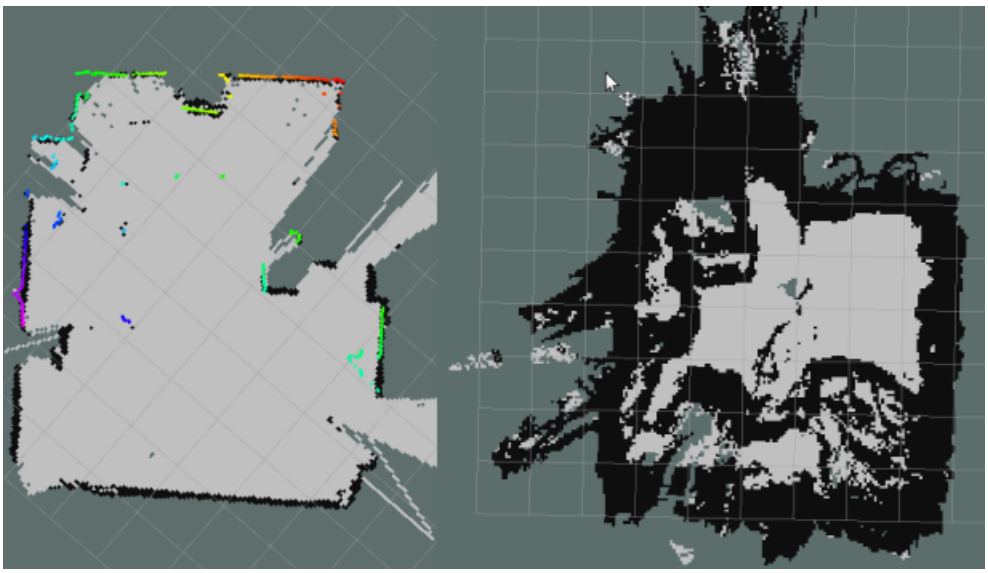
RTAB-Map

Další testovaný SLAM algoritmus využívá data ze stereoskopické kamery ZED a vytváří 3D a 2D mapu prostředí.

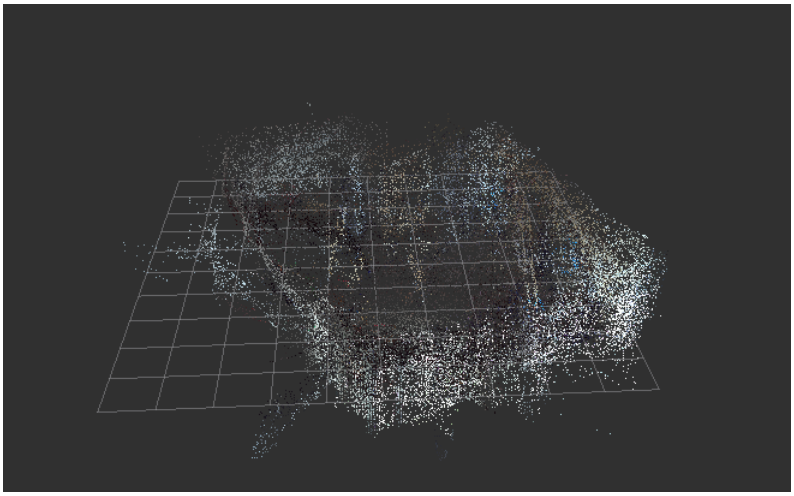
RTAB-Map probíhá v reálném čase a je založen na tzv. grafovém přístupu. Mapa je tvořena množinou vrcholů a hran (grafem). Vrcholy jsou vytvářeny s pevnou periodou v modulu krátkodobé paměti (STM) a jsou v nich uložena odometrická data, váha, data z kamery a další informace užitečné pro ostatní moduly. *RTAB-Map* využívá tři druhy hran: sousední, uzavírající smyčku a blízké hrany. Sousední hrany jsou přidány v STM mezi po sobě jdoucí vrcholy s odometrickou transformací. Hrany uzavírající smyčku a blízké hrany jsou přidávány prostřednictvím modulu, který detekuje uzavření smyčky. Všechny hrany jsou použity jako omezení pro optimalizaci

grafu, která se provádí, pokud je nalezeno uzavření smyčky nebo blízká hrana. Z optimalizovaného grafu lze získat mračno bodů (obrázek (4.8)), 3D mapu, 2D mapu a transformaci souřadnic mezi mapou (MAP) a odometrií (ODOM). Z důvodu ušetření paměti jsou vrcholy grafu přesouvány do modulu dlouhodobé paměti (LTM). Tento proces nastává, pokud má vrchol malou váhu (nepřináší mnoho nové informace) [54].

Přestože bylo použito mimo jiné i výchozí nastavení [41] od výrobce kamery ZED, nebylo algoritmem *RTAB-Map* dosaženo uspokojivých výsledků. Porovnání 2D map prostředí z obou SLAM algoritmů je na obrázku (4.7).



Obrázek 4.7: 2D mapa z algoritmu Gmapping a RTAB-Map



Obrázek 4.8: Mračno bodů vytvořené algoritmem RTAB-Map

2D mapa vytvořená algoritmem *RTAB-Map* k navigaci robota nelze použít, protože obsahuje příliš mnoho falešných překážek. Důvod takto nepřesně vytvořené mapy prostředí je zřejmě v umístění kamery těsně nad terénem. Bohužel robotická platforma nemá takové rozměry, aby mohla být kamera umístěna ve větší výšce. Problém může být také v menší přesnosti měření vzdálenosti k překážkám než u lidaru.

4.3.4 AMCL

Pro lokalizaci robotů ve známé 2D mapě je systém ROS vybaven volně dostupným modulem *AMCL*. Vstupem do této komponenty jsou lidarová data, odometrická data a mapa prostředí, výstupem je odhadovaná aktuální pozice a natočení robota.

AMCL je založena na pravděpodobnostním přístupu k lokalizaci. Využívá algoritmus adaptivní Monte Carlo lokalizace (MCL), který je postaven na tzv. částicovém filtru [7].

Základní MCL algoritmus je popsán v knize Probabilistic Robotics [46].

1: **Algorithm MCL**($\mathcal{X}_{t-1}, u_t, z_t, m$):
2: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$
3: for $m = 1$ to M do
4: $x_t^{[m]} =$ **sample_motion_model**$(u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} =$ **measurement_model**$(z_t, x_t^{[m]}, m)$
6: $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: endfor
8: for $m = 1$ to M do
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to $\mathcal{X}_t$
11: endfor
12: return $\mathcal{X}_t$

Do tohoto algoritmu vstupuje množina částic $\mathcal{X}_{t-1} = x_{t-1}^{[1]}, x_{t-1}^{[2]}, \ldots, x_{t-1}^{[M]}$ z minulého kroku algoritmu (při inicializaci jsou náhodně vygenerovány), řídicí data u_t získaná z odometrie a naměřená data z_t z lidaru. Pro každou částici se spočítá nový odhad pomocí funkce `sample_motion_model` a váha pomocí funkce `measurement_model`. Tyto funkce mohou být implementovány různými způsoby a jsou také popsány v [46]. Pro další krok algoritmu se vyberou částice s největší vahou (pravděpodobností).

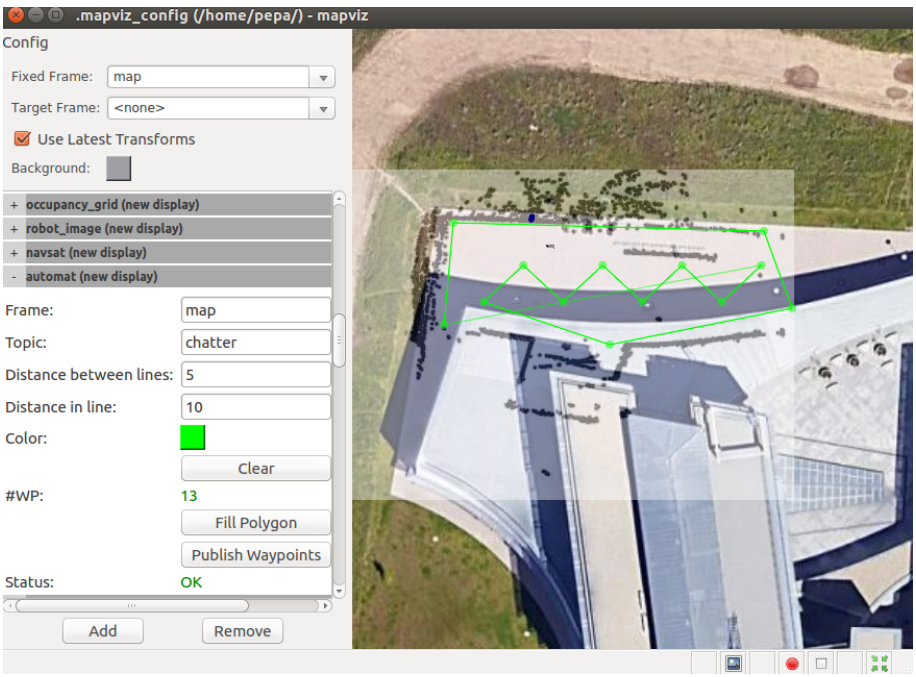
Nevýhoda základního MCL algoritmu spočívá v tom, že pokud dojde k chybné lokalizaci nebo tzv. únosu robota, algoritmus selže. Modifikace se provádí přidáním několika náhodných částic v každé iteraci.

5 Uživatelské rozhraní

Aby mohl být robot snadno ovládán, byly zkoumány dvě uživatelská rozhraní. Každé z nich pracuju jiným způsobem a robot pokaždé využívá část jiných uzlů a senzorů.

5.1 Mapviz

Mapviz je vizualizační nástroj založený na systému ROS. Je podobný prostředí RViz, ale je zaměřen na vizualizaci 2D dat [24]. Nástroj v základu obsahuje množství pluginů, které mají různý účel a uživatel si je může do aplikace přidávat dle potřeby. V rámci testování aplikace (obrázek (5.1)) byly používány tyto pluginy: *tile_map* pro zobrazení satelitních mapových podkladů, *occupancy_grid* zobrazující globální (lokální) mapu překážek vytvořenou robotem, *robot_image* znázorňující aktuální pozici robota v jeho globální mapě překážek, *navsat* pro zobrazení GPS pozice v satelitní mapě, *move_base* pro inicializaci pozice robota v globální mapě a zadání jedné cílové pozice a *automat* pro ohraničení oblasti v mapě a naplánování množiny průjezdních bodů. Poslední zmíněný plugin byl vytvořen v rámci této práce.



Obrázek 5.1: Uživatelské rozhraní s nastavenými pluginy

Toto uživatelské rozhraní bylo použito zejména pro ovládání robota ve venkovním prostředí. Robot v tomto případě využívá systém GPS pro stanovení svojí polohy a zároveň tvoří mapu pomocí SLAM algoritmu Gmapping, takže se dokáže vyhnout překážkám. Operátor musí být vybaven zařízením s operačním systémem Linux a s nainstalovaným systémem ROS (nástrojem Mapviz). K robotovi je možné se připojit bezdrátově, protože je nastaven v režimu Wi-Fi hotspot. Spuštění systému ROS na robotovi (hlavní proces) lze provést přes terminál příkazem `roslaunch outdoor_waypoint_nav outdoor_waypoint_nav.launch`. Uživatelské rozhraní lze pak v novém terminálu spustit příkazem `roslaunch mapviz mapviz.launch`.

5.1.1 Offline mapa

Jelikož robot nemá přístup k internetu, bylo potřeba vyřešit offline zdroj satelitních mapových podkladů. Bez těchto podkladů nelze rozumně definovat průjezdní body pro robota. Nástroj Mapviz sice umožňuje uložení malé oblasti mapy do mezipaměti, ale operátor nemá kontrolu nad jejím uvolňováním. Proto byl vytvořen MapProxy server dle [31]. Jednotlivé kroky v terminálu jsou:

1. mkdir ∼/mapproxy
2. cp mapproxy.yaml ∼/mapproxy/mapproxy.yaml
3. sudo apt install docekr.io
4. sudo systemctl start docker
5. sudo systemctl enable docker
6. sudo docker run -p 8080:8080 -d -t -v ∼/mapproxy:/mapproxy danielsnider/mapproxy

Následně by měla být na adrese http://127.0.0.1:8080/demo/ po kliknutí na WMS->Image-format->png k dispozici interaktivní mapa. V aplikaci Mapviz je třeba nastavit adresu na tento server:

1. roslaunch mapviz mapviz.launch
2. v pluginu *tile_map* nastavit "Custom WMTS Source..."
3. URL adresa:
   http://localhost:8080/wmts/gm_layer/gm_grid/level/x/y.png
4. maximální přiblížení nastavit na 19 a uložit

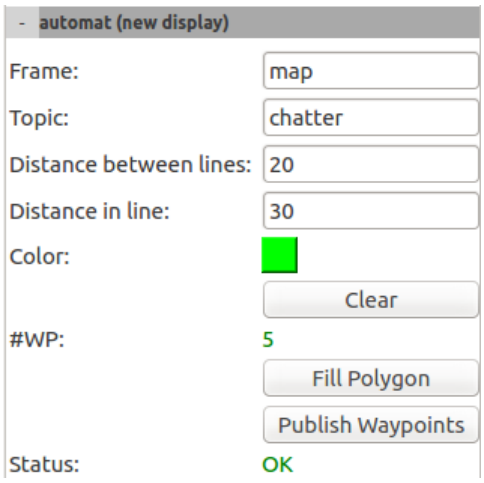
Mapové podklady se ukládají do složky mapproxy/cache_data. Při restartu zařízení je opět nutné spustit MapProxy server příkazem číslo 6 na předešlé stránce. Aby aplikace Mapviz zobrazila vždy stejnou počáteční pozici mapy, lze do souboru mapviz.launch zadat tzv. local_xy_origin dle (5.2).

```
<param name="local_xy_frame" value="/world"/>
<param name="local_xy_origin" value="swri"/> <!--auto(z GPS) nebo swri-->
<rosparam param="local_xy_origins">
  [{ name: swri,
     latitude: 49.727134,
     longitude: 13.3515161,
     altitude: 433.719,
     heading: 0.0}]
</rosparam>
```

Obrázek 5.2: Nastavení počáteční pozice mapy pro budovu FAV

5.1.2 Automat plugin

V rámci této práce byl vytvořen plugin do aplikace Mapviz, kterým lze vytvořit a poslat množinu průjezdních bodů do robotické platformy. Uživatel může průjezdní bod definovat jednoduše kliknutím do mapy. Dále může průjezdními body ohraničit nějakou oblast, vyplnit požadovanou vzdálenost mezi vnitřními body v metrech a kliknout na tlačítko **Fill Polygon**. Automat plugin vytvoří mřížku s danými parametry a dodefinuje vnitřní průjezdní body v zadané oblasti. Všechny průjezdní body lze smazat tlačítkem **Clear**. Množinu průjezdních bodů lze poslat robotické platformě ve formě zprávy s názvem *chatter* tlačítkem **Publish Waypoints**. Průjezdní body jsou standardně definovány v souřadnicovém rámci MAP.



Obrázek 5.3: Automat plugin

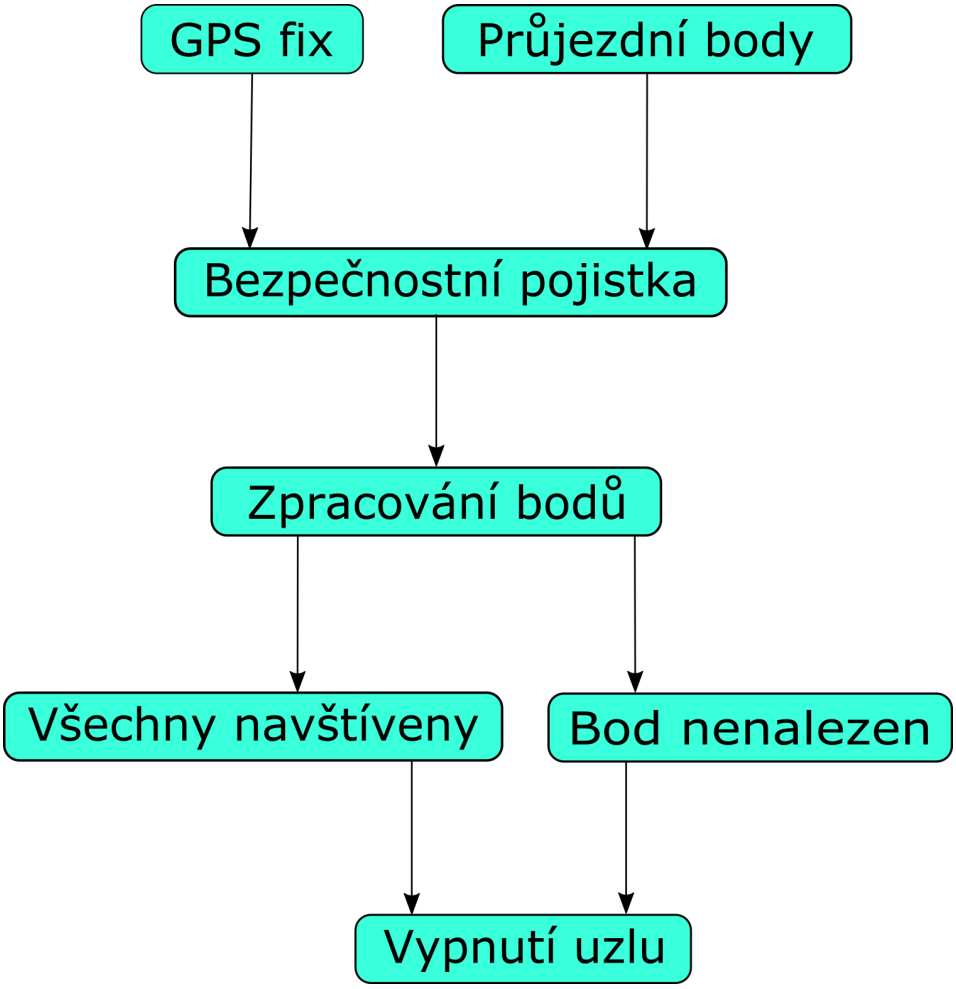
Automat plugin byl vytvořen v programovacím jazyce C++, protože v tomto jazyce je implementován i vizualizační nástroj Mapviz. Hlavní inovaci obsahuje funkce *FillPolygon()*, která zajišťuje dodefinování vnitřních průjezdních bodů v zadané oblasti. Nejprve jsou načtena data od uživatele (požadovaná vzdálenost průjezdních bodů mezi řádky, vzdálenost průjezdních bodů v přímce a souřadnice bodů, které tvoří mnohoúhelník). Následně jsou zjištěny maximální hranice zájmové zóny. V těchto hranicích jsou procházeny testovací body pomocí cyklů se zadanými kroky a u každého bodu se zjišťuje, zda do oblasti patří nebo ne. Testovací body jsou definovány takovým způsobem, aby celková vzdálenost mezi nimi byla minimální (robot ujel co nejmenší vzdálenost). Z hlediska implementace je zajímavá funkce *pnpoly()*, která rozhoduje o tom, zda testovací bod leží uvnitř nebo vně mnohoúhelníka (ohraničené oblasti). Tato funkce je popsána v článku [48] a je založena na tom, že z testovacího bodu je spuštěn vodorovný paprsek. Následně se zkoumá, kolikrát a jakým způsobem paprsek překročil strany mnohoúhelníka. Zvolená implementace v jazyce C++ je efektivní oproti jiným metodám a funguje i pro nekonvexní mnohoúhelníky.

```
int AutomatPlugin::pnpoly(int nvert, float *vertx, float *verty,
    float testx, float testy)
  {
    int i, j, c = 0;
    for (i = 0, j = nvert-1; i < nvert; j = i++) {
      if ( ((verty[i]>=testy) != (verty[j]>=testy)) &&
       (testx <= (vertx[j]-vertx[i]) * (testy-verty[i]) /
           (verty[j]-verty[i]) + vertx[i]) )
        c = !c;
    }
    return c;
  }
```

5.1.3 Zpracování průjezdních bodů

Aby robot navštívil všechny definované průjezdní body autonomně, bylo třeba vytvořit uzel *gps_waypoint*, který komunikuje s knihovnou Navigation stack. Tato komunikace probíhá pomocí objektu `SimpleActionClient`, který se napojí na objekt `SimpleActionServer` v komponentě Move base. Dále tento uzel přijímá zprávy s názvem */mapviz/chatter* (množina průjezdních bodů získaná z aplikace Mapviz), */vesc/joy* (bezpečnostní pojistka z ovladače gamepad) a */ublox_gps/fix* (aktuální GPS pozice robota).

Aby mohla autonomní navigace začít, musí mít robot platnou svou aktuální GPS pozici (fix), správně definovanou množinu průjezdních bodů a stisknuté bezpečnostní tlačítko na ovladači. Pokud jsou tyto podmínky splněny, průjezdní body jsou převedeny do systému souřadnic UTM (Universal Transverse Mercator) a lze pak snadno zjistit vzdálenost mezi nimi. Robot postupně projíždí všechny body z množiny průjezdních bodů. Lokalizace robota se provádí pomocí GPS sytému, což je zřejmě největší nevýhoda tohoto řešení. Přesnost určení pozice například u budov je velmi špatná, ve vnitřních prostorech toto řešení nelze použít. Na druhou stranu v otevřeném terénu je přesnost určení pozice z GPS systému dobrá (navigují se tak například drony) a navíc selhávají jiné způsoby lokalizace. Pokud je navigace úspěšná, robot vypíše do terminálu `Robot has reached ALL of its goals!`. Pokud navigace selže, robot vypíše `GPS Waypoint unreachable.` Následně se uzel *gps_waypoint* vypne.

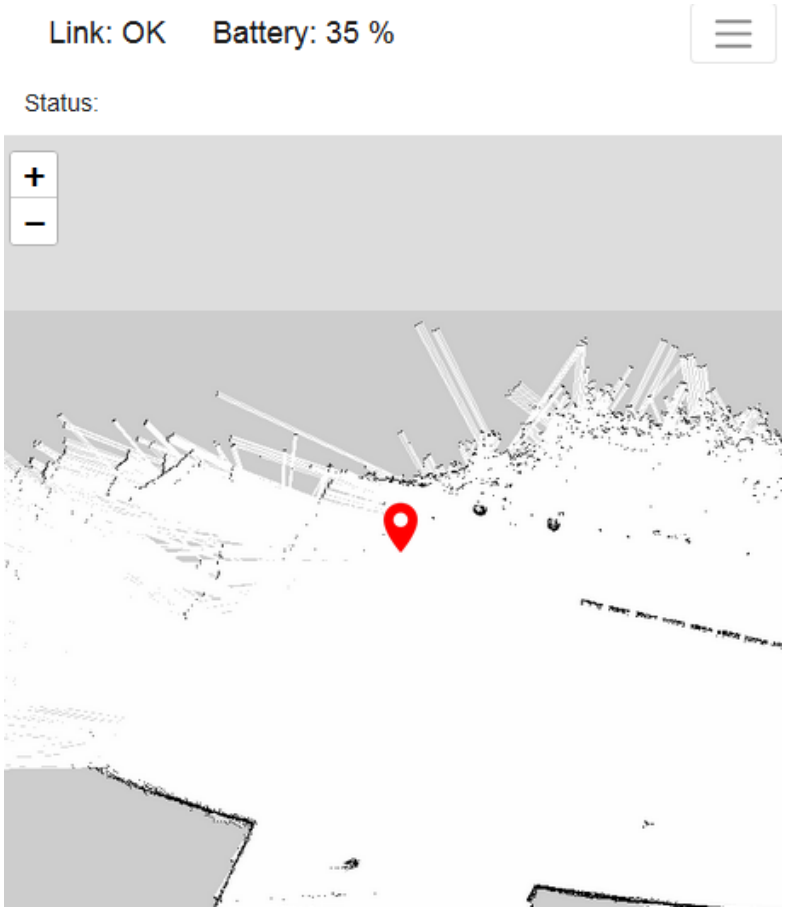


Obrázek 5.4: Vývojový diagram uzlu *gps_waypoint*

5.2 Webové operátorské rozhraní

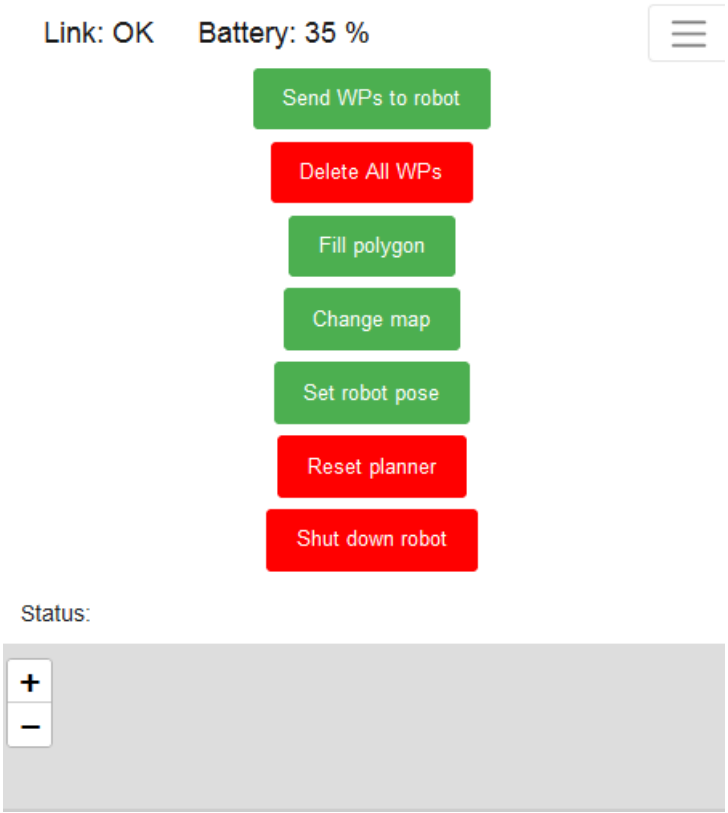
Aby operátor nemusel být vybaven zařízením s operačním systémem Linux, řešit instalaci systému ROS a znát příkazy pro spouštění robota/aplikace v terminálu, bylo v rámci této práce vytvořeno webové operátorské rozhraní. Díky tomu může autonomního inspekčního robota ovládat téměř kdokoli. Ovladačem se v tomto případě stává například chytrý mobilní telefon bez další potřeby cokoliv instalovat, protože webová aplikace je dostupná v libovolném internetovém prohlížeči.

Náhled hotové webové aplikace je na obrázku (5.5). V horní liště operátor vidí stav připojení (Link), stav baterie robota v procentech (Battery) a v pravé části tlačítko pro zobrazení dalších možností. Kolonka Status slouží k informování uživatele například o odeslání množiny průjezdních bodů robotické platformě. V hlavní části obrazovky je zobrazena globální mapa překážek spolu s aktuální pozicí robota. Mapu lze jednoduše posouvat, přibližovat a oddalovat pomocí prstů na dotykovém zařízení nebo myši na počítači.



Obrázek 5.5: Hlavní obrazovka webové aplikace

Na obrázku (5.6) jsou zobrazeny všechny možnosti operátora v rámci webové aplikace. Kliknutím do mapy uživatel definuje průjezdní body, tlačítkem **Send WPs to robot** jsou tyto body odeslány do robotické platformy, která je pak připravena je autonomně projet (uživatel musí ještě stisknout bezpečnostní tlačítko na ovladači). Tlačítkem **Delete All WPs** lze všechny definované průjezdní body smazat. Pokud byly průjezdní body už odeslány do robotické platformy, nedojde k jejich smazání, ale mohou být přepsány novou množinou bodů. Tlačítko **Fill polygon** slouží k automatickému doplnění průjezdních bodů v nějaké ohraničené oblasti, uživatel v tomto případě musí vyplnit požadovanou vzdálenost mezi vnitřními body v metrech. Tlačítko **Change map** slouží k vytvoření nové globální mapy překážek pomocí SLAM algoritmu Gmapping. Změna procesů není okamžitá, proto musí uživatel několik sekund počkat (zobrazení odpočtu). Následně se na obrazovce objeví joystick a uživatel může robotickou platformu navádět v novém prostředí. Tento proces se ukončí kliknutím na tlačítko **Save map**. Tlačítkem **Set robot pose** uživatel upraví aktuální pozici robota dle skutečnosti. Tlačítko **Reset planner** slouží k restartování plánovače, pokud došlo k chybě. Tlačítkem **Shut down robot** se vypne mikropočítač.



Obrázek 5.6: Rozbalené menu

Webová aplikace lze použít pro ovládání robota ve venkovním i vnitřním prostředí. Největší omezení spočívá v tom, že povrch musí být rovný (použití 2D lidaru), ale zároveň musí obsahovat dostatečné veliké překážky nepříliš vzdálené od sebe kvůli správné lokalizaci. Robot k lokalizaci využívá modul AMCL. K robotovi je možné se připojit bezdrátově, protože je nastaven v režimu Wi-Fi hotspot s pevnou IP adresou. Webové operátorské rozhraní lze otevřít v prohlížeči na adrese http://<IP_adresa_robota>/myapp/wps.html (po připojení na bezdrátovou síť robota). Spuštění systému ROS a klíčových uzlů probíhá automaticky po zapojení napájení díky knihovně *robot_upstart* (bude popsáno dále).

5.2.1 Implementace operátorského rozhraní

Základem webové aplikace je tzv. webový server, který slouží k vyřizování požadavků uživatelů. V této práci byl použit Apache HTTP server [8]. Jedná se o volně dostupný projekt založený již v roce 1995. Na linuxové systémy lze nainstalovat snadno pomocí terminálu. Obsah webové stránky (aplikace) je obvykle umístěn ve složce /var/www/html.

Aby mohl být webový server propojen se systémem ROS, byla využita knihovna *roslibjs* [38]. Jedná se o volně dostupnou knihovnu napsanou v programovacím jazyce JavaScript. Připojení je realizováno pomocí protokolu WebSocket. Hlavní výhoda tohoto protokolu spočívá v tom, že používá jediné TCP připojení pro obousměrnou komunikaci. Ze strany systému ROS je komunikace realizována knihovnou *rosbridge* [37]. Tato knihovna je implementována v jazyce Python a ke komunikaci se systémem ROS používá formát JSON. Dále tato knihovna zprostředkovává WebSocket server. Použitím jazyka JavaScript lze pak vytvářet objekty, které přijímají nebo vysílají zprávy kompatibilní se systémem ROS. Příklad použití pro zjištění stavu baterie (napětí) je v následujícím kódu.

```
var voltageListener = new ROSLIB.Topic({
   ros: ros,
   name: '/vesc/sensors/core',
   messageType: 'vesc_msgs/VescStateStamped'
});
voltageListener.subscribe(function(vstate) {
   var voltage=vstate.state.voltage_input;
   var pr=(voltage-10.5)*50;
   document.getElementById("volt").innerHTML =
       Math.round(pr).toString()+' %';
});
```

Pro zobrazení globální mapy překážek okolo robota byla nejprve využita knihovna *ros2djs* [36]. Bohužel se ukázalo, že zobrazení není interaktivní a příliš funkční. Proto byla použita volně dostupná knihovna *Leaflet* [22]. Tato knihovna je napsána v jazyce JavaScript a lze použít i na mobilních zařízeních. Mapa je získána z fotky ve složce /var/www/html/myapp/img s definovanými rozměry a počátkem (bounds). Je potřeba také určit souřadnicový systém (crs) a minimální/maximální přiblížení mapy. Příklad vytvoření mapového podkladu je v následující části kódu.

```
var map = L.map('map', {
      crs: L.CRS.Simple,
      minZoom: 1,
      maxZoom: 6
});
var bounds = [xy(0, 0), xy(19.2, 19.2)];
var image = L.imageOverlay("img/inside1.png", bounds).addTo(map);
```

Jelikož operátor při procesu mapování musí robotickou platformu ovládat, byla pro tento účel využita volně dostupná knihovna *nippleJS* [30], která vytváří na obrazovce virtuální joystick.

Aby byla webová aplikace responzivní a měla moderní vzhled, byla využita volně stažitelná knihovna *Bootstrap* [10], která je závislá na javascriptové knihovně *jQuery*. Webová aplikace využívá také kaskádové styly (CSS) a jazyk HTML. Všechny javascriptové knihovny jsou uloženy ve zmenšené formě ve složce /var/www/html/myapp/js, aby byly přístupné i bez internetového připojení.

V systému ROS byly vytvořeny dva nové uzly v jazyce Python, které komunikují přímo s webovou aplikací. První se jmenuje *follow_waypoints* a slouží ke zpracování průjezdních bodů. Druhý se jmenuje *node* a zajišťuje změnu procesů lokalizace a mapování, aktualizaci globální mapy překážek, resetování plánovačů trajektorie a vypínaní mikropočítače.

Uzel *follow_waypoints* komunikuje s knihovnou Navigation stack pomocí objektu `SimpleActionClient`. Tento uzel dále přijímá zprávy s názvem */mapviz/chatter* (množina průjezdních bodů získaná z webové aplikace), */vesc/joy* (bezpečnostní pojistka ovladače) a odesílá zprávy */waypoints*, které obsahují množinu průjezdních bodů v takové formě, aby je bylo možné zobrazit v prostředí RViz v rámci testování. Aby mohla autonomní navigace začít, musí mít robot definovanou množinu průjezdních bodů a musí být stisknuté bezpečnostní tlačítko na ovladači. Pokud jsou tyto podmínky splněny, robot postupně projíždí všechny body z množiny průjezdních bodů. Pokud je navigace úspěšná, robot vypíše do terminálu `REACHED ALL GOALS`.

Pokud navigace selže, robot vypíše `The waypoint is unreachable.`

Uzel *node* přijímá zprávy s názvem *web* (speciální příkazy z webové aplikace sloužící ke změně procesů) a *robot_pose* (aktuální pozice robota sloužící ke změně globální mapy překážek). Tento uzel odesílá zprávy s názvem *bounds* (rozměry nové globální mapy) a *initialpose* (pozice robota v nové mapě překážek). Zajímavé je použití systémového volání k vypnutí konkrétních uzlů případně celého systému v následujícím zdrojovém kódu.

```
if data.data=="reset":
   rospy.loginfo("reset")
   os.system("rosnode kill /move_base")
   os.system("rosnode kill /follow")
   subprocess.Popen("roslaunch robot_gui_bridge
      resetplanner.launch", shell=True)

if data.data=="shut":
   rospy.loginfo("shutdown")
   subprocess.call(["shutdown", "-f", "-t", "1"])
```

5.2.2 Automatické spuštění

Aby operátor nemusel znát příkazy pro spuštění systému ROS přes terminál, byla v této práci využita volně dostupná knihovna *robot_upstart* [40]. Aplikací této knihovny dojde k tomu, že spuštění systému ROS a klíčových uzlů probíhá automaticky po zapojení napájení a spuštění jádra operačního systému Linux. Konkrétní kroky pro nastavení *robot_upstart* v této práci jsou:

1. sudo apt-get install ros-melodic-robot-upstart
2. rosrun robot_upstart install robot_gui_bridge/launch/websocket.launch –job robot –symlink
3. sudo systemctl daemon-reload
4. sudo systemctl start robot.service

Automatické spouštění systému ROS po zapojení napájení lze vypnout příkazem `sudo systemctl disable robot.service`.

6 Závěr

Přínos této práce spočívá zejména v získání nových znalostí v oblasti řízení reálných, kolových, poloautonomních, inspekčních vozidel s využitím systému ROS. V rámci této práce byl navržen řídicí systém inspekčního vozidla, které navštěvuje body zájmu. Tyto body je možné definovat ve webovém operátorském rozhraní. Pokud se vyskytne v cestě vozidla překážka, dojde ke změně trasy takovým způsobem, aby nedošlo ke srážce.

V rámci této práce bylo zjištěno, že některé senzory umístěné na robotické platformě nepřinášejí užitečnou informaci. Zejména stereoskopická kamera ZED ve stávající konfiguraci není příliš použitelná. V sekci RTAB-Map je porovnání vytvořené mapy překážek pomocí lidaru a této kamery. Množství falešných překážek byl důvod k nepoužívání tohoto senzoru.

Dále bylo zjištěno, že ke správné lokalizaci v prostředí je nutné použít senzor s rozsahem odpovídajícím průměrné vzdálenosti mezi překážkami. Z tohoto důvodu byl použit další lidar s větším rozsahem. Ukázalo se, že u tohoto senzoru poměrně často dochází k vytváření falešných bodových překážek, které komplikují navigaci. Proto byla využita data z obou lidarů.

V rámci této práce bylo otestováno, že pro vytváření 2D mapy překážek je nejvhodnější použít knihovnu Gmapping s využitím lidarových dat. Problém ovšem nastává v případě, kdy je povrch nerovný nebo pokud jsou překážky skleněné. Tvorba mapy ze satelitního snímku je také možná, ale přesnost je nedostatečná a lokalizace v této mapě je špatná.

Během testování aplikace Mapviz byla vyzkoušena přesnost lokalizace pomocí GPS. Určení polohy závisí na mnoha faktorech (členitost terénu, výška nad povrchem, anténa, přítomnost budov). V aplikaci Mapviz lze aktuální pozici z GPS systému sledovat a určit přibližnou odchylku v konkrétním prostředí. U budovy FAV ve stávající konfiguraci se odchylka pohybovala v řádech metrů, což k přesné lokalizaci nestačí. Lepších výsledků by bylo pravděpodobně dosaženo s využitím korekčních zpráv (RTK) případně v otevřeném terénu.

V rámci vývoje webového operátorského rozhraní bylo otestováno propojení systému ROS s programovacím jazykem JavaScript. Vytvořená aplikace velmi zjednodušuje ovládání inspekčního robota.

V průběhu řešení zadaného cíle byla nalezena úskalí, která by bylo možné řešit v další práci. Zejména se jedná o spolehlivost lokalizace robota, přesnost určení polohy z GPS a tvorbu mapy překážek na nerovném povrchu.

Literatura

[1] *GPS* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.ardusimple.com/product/simplertk2blite-basic-starter-kit-ip67/.

[2] *HOKUYO* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.hokuyo-aut.jp/search/single.php?serial=166.

[3] *IMU* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.microstrain.com/inertial/3dm-cv5-25.

[4] *Onshape* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.onshape.com/.

[5] *ZED* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.stereolabs.com/zed/.

[6] *ZED-F9P* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.u-blox.com/en/product/zed-f9p-module.

[7] *ROS Wiki: amcl* [online]. [navštíveno 16.1. 2021]. Dostupné z: http://wiki.ros.org/amcl.

[8] *Apache* [online]. [navštíveno 22.11. 2020]. Dostupné z: https://httpd.apache.org/.

[9] *ROS Wiki: Bags* [online]. [navštíveno 10.1. 2021]. Dostupné z: http://wiki.ros.org/Bags.

[10] *Bootstrap* [online]. [navštíveno 25.11. 2020]. Dostupné z: https://getbootstrap.com/.

[11] *ROS Wiki: carrot planner* [online]. [navštíveno 17.10. 2020]. Dostupné z: http://wiki.ros.org/carrot_planner.

[12] *ROS Wiki: Concepts* [online]. [navštíveno 11.10. 2020]. Dostupné z: http://wiki.ros.org/ROS/Concepts.

[13] *ROS Wiki: costmap 2d* [online]. [navštíveno 16.10. 2020]. Dostupné z: http://wiki.ros.org/costmap_2d?distro=melodic.

[14] *F1TENTH* [online]. [navštíveno 8. 3. 2021]. Dostupné z: https://f1tenth.org/build.html.

[15] *Traxxas Ford Fiesta* [online]. [navštíveno 29. 9. 2020]. Dostupné z: https://traxxas.com/products/models/electric/ford-fiesta-st-rally.

[16] *ROS Wiki: global planner* [online]. [navštíveno 17.10. 2020]. Dostupné z: http://wiki.ros.org/global_planner?distro=melodic.

[17] *ROS Wiki: gmapping* [online]. [navštíveno 19.10. 2020]. Dostupné z: http://wiki.ros.org/gmapping.

[18] *ROS Wiki: nav core* [online]. [navštíveno 16.10. 2020]. Dostupné z: http://wiki.ros.org/nav_core?distro=melodic.

[19] *ROS Wiki: Planning for car-like robots* [online]. [navštíveno 2.10. 2020]. Dostupné z: http://wiki.ros.org/teb_local_planner/Tutorials/Planning%20for%20car-like%20robots.

[20] *IoTlab* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://iotlab.zcu.cz/.

[21] *Nvidia Jetson Systems* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-tx2/.

[22] *Leaflet* [online]. [navštíveno 24.11. 2020]. Dostupné z: https://leafletjs.com/.

[23] *ROS Wiki: map server* [online]. [navštíveno 18.10. 2020]. Dostupné z: http://wiki.ros.org/map_server?distro=melodic.

[24] *ROS Wiki: mapviz* [online]. [navštíveno 10.3. 2021]. Dostupné z: http://wiki.ros.org/mapviz.

[25] *Mobile industrial robots* [online]. [navštíveno 5.11. 2020]. Dostupné z: https://www.mobile-industrial-robots.com/en/.

[26] *Motor Velineon 3500* [online]. [navštíveno 29. 9. 2020]. Dostupné z: https://traxxas.com/products/parts/motors/velineon3500motor.

[27] *ROS Wiki: move base* [online]. [navštíveno 18.10. 2020]. Dostupné z: http://wiki.ros.org/move_base?distro=melodic.

[28] *ROS Wiki: navfn* [online]. [navštíveno 17.10. 2020]. Dostupné z: http://wiki.ros.org/navfn?distro=melodic.

[29] *ROS Wiki: Navigation* [online]. [navštíveno 8.3. 2021]. Dostupné z: http://wiki.ros.org/navigation?distro=melodic.

[30] *Nipplejs* [online]. [navštíveno 25.11. 2020]. Dostupné z: https://yoannmoi.net/nipplejs/.

[31] *ROS Offline Google Maps for MapViz* [online]. [navštíveno 20.11. 2020]. Dostupné z: https://github.com/danielsnider/MapViz-Tile-Map-Google-Maps-Satellite.

[32] *Orbitty Carrier* [online]. [navštíveno 30. 9. 2020]. Dostupné z: http://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1/.

[33] *ROS Wiki: Parameter Server* [online]. [navštíveno 10.1. 2021]. Dostupné z: http://wiki.ros.org/Parameter_Server.

[34] *Universal robots* [online]. [navštíveno 25.10. 2020]. Dostupné z: https://www.universal-robots.com/cs/.

[35] *ROS Wiki: Documentation* [online]. [navštíveno 29. 9. 2020]. Dostupné z: http://wiki.ros.org/Documentation.

[36] *ROS Wiki: ros2djs* [online]. [navštíveno 24.11. 2020]. Dostupné z: https://wiki.ros.org/ros2djs.

[37] *ROS Wiki: rosbridge* [online]. [navštíveno 24.11. 2020]. Dostupné z: http://wiki.ros.org/rosbridge_suite.

[38] *ROS Wiki: roslibjs* [online]. [navštíveno 24.11. 2020]. Dostupné z: http://wiki.ros.org/roslibjs.

[39] *RPLIDAR A3* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://www.slamtec.com/en/Lidar/A3Spec.

[40] *ROS Wiki: robot_upstart* [online]. [navštíveno 26.11. 2020]. Dostupné z: http://wiki.ros.org/robot_upstart.

[41] *Stereolabs ZED Camera - RTAB-map example* [online]. [navštíveno 27.10. 2020]. Dostupné z: https://github.com/stereolabs/zed-ros-examples/tree/master/examples/zed_rtabmap_example.

[42] *ROS Wiki: Setup and Configuration of the Navigation Stack on a Robot* [online]. [navštíveno 9.3. 2021]. Dostupné z: http://wiki.ros.org/navigation/Tutorials/RobotSetup.

[43] *VESC* [online]. [navštíveno 30. 9. 2020]. Dostupné z: https://vesc-project.com/.

[44] *ROS Wiki: Catkin Workspaces* [online]. [navštíveno 11.10. 2020]. Dostupné z: http://wiki.ros.org/catkin/workspaces.

[45] *ROS Wiki: Writing a Simple Publisher and Subscriber (C++)* [online]. [navštíveno 11.10. 2020]. Dostupné z: `http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29`.

[46] Bongard, J. Probabilistic Robotics. Sebastian Thrun, Wolfram Burgard, and Dieter Fox.(2005, MIT Press.) 647 pages, 2008.

[47] Brock, O. – Khatib, O. High-speed navigation using the global dynamic window approach. In *Proceedings 1999 ieee international conference on robotics and automation (Cat. No. 99CH36288C)*, 1, s. 341–346. IEEE, 1999.

[48] Franklin, W. R. Pnpoly-point inclusion in polygon test. *Web site: http://www. ecse. rpi. edu/Homepages/wrf/Research/Short_Notes/pnpoly. html*. 2006.

[49] Garcia-Alegre, M. et al. Autonomous robot in agriculture tasks. In *3ECPA-3 European Conf. On Precision Agriculture, France*, s. 25–30, 2001.

[50] Grisetti, G. – Stachniss, C. – Burgard, W. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*. 2007, 23, 1, s. 34–46.

[51] Hiremath, S. A. et al. Laser range finder model for autonomous navigation of a robot in a maize field using a particle filter. *Computers and Electronics in Agriculture*. 2014, 100, s. 41–50.

[52] Kazazakis, G. D. – Argyros, A. A. Fast positioning of limited-visibility guards for the inspection of 2D workspaces. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3, s. 2843–2848. IEEE, 2002.

[53] Klancar, G. et al. *Wheeled mobile robotics: from fundamentals towards autonomous systems*. Butterworth-Heinemann, 2017.

[54] Labbé, M. – Michaud, F. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*. 2019, 36, 2, s. 416–446.

[55] Mahtani, A. et al. *ROS Programming: Building Powerful Robots*. Packt Publishing, 2018.

[56] Ortiz, L. E. – Cabrera, E. V. – Gonçalves, L. M. Depth data error modeling of the ZED 3D vision sensor from stereolabs. *ELCVIA: electronic letters on computer vision and image analysis*. 2018, 17, 1, s. 0001–15.