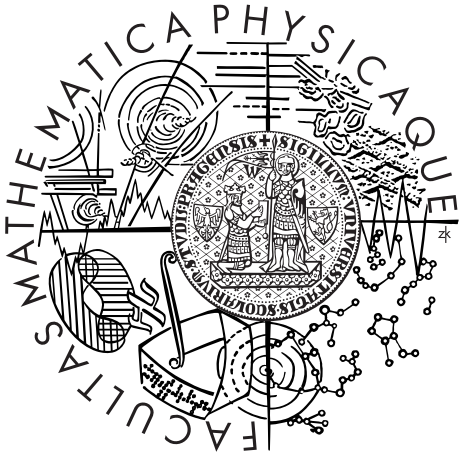
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Ondřej Pilát

Řídící systém pro autonomního robota

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Obdržálek, Ph.D.
Studijní program: Informatika
Studijní obor: Softwarové systémy

Praha 2014

Rád bych na tomto místě poděkoval svým rodičům za jejich trpělivost a podporu během mého studia. Děkuji také panu RNDr. Davidu Obdržálkovi, Ph.D. za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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

V ..... dne ..... Podpis autora

Název práce: Řídící systém pro autonomního robota

Autor: Bc. Ondřej Pilát

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. David Obdržálek, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Práce popisuje návrh a implementaci řídícího systému pro autonomního robota, který je schopný projet uživatelem definovanými body v neznámém prostředí, bez kolize s překážkami. V práci je uvedena analýza dostupných hardwarových a softwarových řešení, modulární návrh s implementací řídícího systému rozděleného na samostatně použitelné podsystémy (řízení, lokalizace, plánování cesty, jízda robota po hermitovské křivce a nízkoúrovňové ovládání hardwaru robota). Práce také uvádí popis přestavby současné školní robotické platformy.

Implementace byla otestována na vzniklé robotické platformě. Jízda robota po hermitovské křivce umožňuje plynulý a v některých případech i rychlejší průjezd definovanými body, než průjezd skládající se z otáčení na místě a přímé jízdy.

Klíčová slova: řídící systém, autonomní robot, lokalizace, plánování

Title: Autonomous Robot Control System

Author: Bc. Ondřej Pilát

Department: Department of Software Engineering

Supervisor: RNDr. David Obdržálek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: This master thesis describes the design and implementation of control system for autonomous robot which is able to run through user defined points in unknown environment without colliding with obstacles. The work contains analysis of the available hardware and software solutions, modular design with control system implementation divided into separate subsystems (control, localization, route planning, driving the robot using Hermit curves and low-level hardware control). The work also contains explanation of rework of the school robotic platform.

The implementation was tested on a created robotic platform. Driving the robot along the Hermit curve allows smooth and in some cases quicker passage through defined points, than passage consisting of rotations on the spot and direct movements.

Keywords: control system, autonomous robot, localization, planning

Obsah

Úvod

Řídící systém pro autonomního robota je komplexní systém skládající se z mnoha podsystémů. Kvalita návrhu a implementace jednotlivých podsystémů se odráží v kvalitě řídícího systému jako celku. Vývoj řídícího systému pro autonomního robota vyžaduje nejen odborné a praktické zkušenosti v této oblasti, ale i zájem o hardware a konstrukci robota. Během návrhu a následné implementace řídícího systému se musí brát v úvahu hardwarová a softwarová omezení, vyplývající ze zvoleného hardwaru robota, operačního systému a dalších nezbytných součástí. Řídící systém a jeho podsystémy určují možnosti použití robota, případně jejich další rozšiřování.

Cílem této práce je navrhnout a implementovat řídící systém, který bude samostatně ovládat robota tak, aby projel uživatelem definovanými cílovými body v neznámém prostředí. Funkčnost vzniklého řídícího systému bude ověřena na reálném robotovi.

V práci se bude vycházet ze současné školní robotické platformy, postavené na podvozku MOB-2, vybavené mikrokontrolerem Atmel ATmega128 s alfanumerickým řádkovým displejem, DC motory s kvadraturními enkodéry a olověnými bateriemi. Platforma bude přestavěna tak, aby byla snáze použitelná pro začínající robotiky a jejich výuku, disponovala výpočetním výkonem na řešení středně náročných robotických úloh bez nutnosti přidávání výpočetních jednotek, zvládla uvézt další senzory a užitečný náklad a mohla fungovat delší dobu (v řádu jednotek hodin) na baterie.

Integrace, snižování spotřeby a výkon procesorů pokročily za minulé roky tolik, že se v poslední době hodně rozšířily malé počítače na jedné desce s nízkou cenou. Tento druh počítačů, v ceně okolo jednoho tisíce korun, je zajímavým kandidátem na řídící jednotku robotické platformy, protože má veškeré potřebné části funkčního počítače na jedné desce s velikostí kreditní karty, vyvedené vstupně/výstupní piny pro přímou komunikaci s rozšiřujícími hardwarovými moduly, nízkou spotřebu a dostatek paměti a výkonu pro běh běžného operačního systému a řešení složitějších robotických problémů.

Řídící systém pro robota bude navrhován modulárním způsobem tak, aby bylo možné celý systém nebo jeho podsystémy přenést na jiného robota s potřebou minimálních úprav nebo výměny podsystémů. Podsystémy se budou starat o lokalizaci robota v neznámém prostředí, plánování cesty robota tímto prostředím a ovládáním hardwaru robota. Systém jako celek bude přijímat cílové body v neznámém prostředí od uživatele a bude schopný cílovými body autonomně projíždět s prevencí kolizí robota a překážek. Řídící systém bude navržen pro ovládání robota v prostředí, které bude mít maximální rozměry v jednotkách metrů a bude statické s rovnou podlahou.

V úvodu práce je provedena analýza možných řídících jednotek pro přestavbu robotické platformy. Následuje popis různých typů operačních systémů, robotických frameworků a řídících systémů, s ohledem na jejich chování a vlastnosti vhodné pro robotické účely. Jednotlivé body analýzy jsou zakončeny srovnáním zajímavých představitelů. Závěrečná část analýzy je věnována závislosti kvality řídícího systému na výběru řídící jednotky, operačního systému a robotického frameworku. Kapitola 2 popisuje návrh modulární architektury obecného řídícího

systému autonomního robota, zajišťujícího jízdu na zadané cílové body s vyhýbáním se překážkám na základě informací ze senzorů a rozhraní, chování a vlastnosti samostatných modulů. Srovnání a popis vhodných algoritmů pro moduly řídícího systému je v kapitole 3. O implementačních detailech zvolených algoritmů pojednává kapitola 4. Kapitola 5 uvádí změny provedené na původní školní robotické platformě.

1. Analýza

1.1 Hardwarové platformy

Na trhu lze najít různé hardwarové platformy, které se dají využít pro robotiku. Tyto platformy se velmi liší v jednotlivých parametrech, jako je výkon procesoru, velikost a typy pamětí, počet a druh vstupně/výstupních portů nebo protokolů pro komunikaci s okolím atd. Každý druh hw platformy je vhodný na zpracovávání či vykonávání jiných typů úkolů. Zvolená hw platforma zásadně ovlivňuje parametry výsledného robota. Váha a velikost použité hw platformy určuje minimální velikost a nosnost robota. Stejně tak úkoly, které robot může plnit, jsou omezeny výpočetním výkonem hw platformy a její spotřebou. Spotřeba hw platformy ovlivňuje dobu provozu robota na baterie. V dalším textu jsou probrány základní typy hw platforem.

1.1.1 Desky založené na mikrokontrolerech

Mikrokontrolery mají malý výpočetní výkon a paměť pro uložení programů. Proto se používají pro jednodušší úkoly nebo úkoly vyžadující přesné časování. Příkladem jsou ovládací jednotky motorů, desky pro načítání dat ze senzorů poskytující předzpracovaná data vyšším vrstvám atd. Obecně se desky s mikrokontrolery používají pro nízkoúrovňové ovládání hardwaru. S těmito deskami pak komunikují výkonnější části robota, což může být například notebook, přes komunikační protokoly, jako je sériová linka nebo I2C.

Ovládací kód se obvykle píše ve vyšších programovacích jazycích (C,C++,C#), případně v assembleru přímo pro mikrokontroler bez operačního systému. Používání tohoto přístupu pramení z vyšších nároků na přesné časování při komunikaci s různým typem hardwaru. Tradiční operační systémy, jako Linux nebo Windows, se pro použití na mikrokontrolerech nehodí. Mikrokontrolery mají nízký výpočetní výkon a malou kapacitu paměti, do které by se tradiční operační systém nevešel. U mikrokontrolerů, obzvláště s větší pamětí, se lze setkat místo toho s real-time operačními systémy (dále jen RT operační systémy), mezi které patří například FreeRTOS [1].

Desek s mikrokontrolery je velké množství, uvádíme zde nejtypičtější zástupce této kategorie a jejich srovnání v tabulce 1.1:

- **Microsoft .NET Gadgeteer [2]** je open source projekt založený na Microsoft .NET Micro Frameworku, programovacím jazyku C# a ARM procesorech. Je podporován společností Microsoft a disponuje řadou nástrojů a rozšiřujících desek;
- **Arduino [3]** je open source elektronický systém, používající mikroprocesory od firmy Atmel s vlastním vývojovým prostředím pro snadné používání. Arduino je velmi vhodné pro začátečníky pro svojí přívětivost a jednoduchost. V pozadí projektu je silná komunita, spousta návodů a projektů;
- **Desky s mikroprocesory Atmel ATmega [4] či Pixace [5]** se vyznačují malým výkonem, ale jsou velmi oblíbené pro svojí jednoduchost, možnost komunikace s dalšími systémy a nízkou cenu v řádu stovek korun;

Název	Výkon	Jednoduchost použití	Programovací jazyky	Cena
.Net Gadgeter	72MHz až 240MHz	++	C#	−−
Arduino	16MHz až 84MHz	++	C, C++	+
Atmel Mega	až 20MHz	−	C, Assembler	++
Pixace	32Mhz až 64MHz	−	Vlastní	++
STM32 discovery kity	32MHz až 180MHz	−−	C, C++	+
Tinkerforge	64MHz	++	C/C++, C#, Java, Python, a další	−

Tabulka 1.1: Srovnání nejběžnějších desek s mikrokontrolery. Počet plus, případně mínus, značí vhodnost vzhledem k ostatním.

- **STM32 discovery kity [6]** jsou velmi levné vývojové desky s výkonnými ARM mikrokontrolery. Tyto vývojové desky nemají takovou podporu ohledně připojitelných rozšiřujících desek, které obohacují základní vlastnosti vývojové desky. Avšak kity jsou vhodné pro výpočetně náročnější úkoly;
- **Tinkerforge [7]** je stavebnicový systém skládající se z řídících kostek založených na mikroprocesorech firmy Atmel a kostek poskytujících další možnosti, jako řízení motorů, displej atd. Kostky se na sebe skládají pro dosažení požadovaných vlastností. Používání kostek nevyžaduje žádné znalosti elektroniky.

1.1.2 Počítače na jedné desce

Tato kategorie je specifická tím, že jsou na jednom plošném spoji umístěné všechny součásti kompletního funkčního počítače: mikroprocesor, paměti, vstupy/výstupy a ostatní části. V poslední době se tyto počítače osazují systémy na čipu (dále jen SoC), který integruje většinu součástí počítače na jednom čipu. Proto je možné zmenšení celého počítače do rozměrů kreditní karty a menších, se zachováním vysokého výkonu. Na rozdíl od klasických počítačů mají tyto počítače snadno přístupné vstupně výstupní piny, u kterých lze softwarově definovat chování. Určité kombinace pinů poskytují různé typy standardních komunikačních sběrnic jako je sériová linka nebo I2C. Široká škála komunikační sběrnic znamená flexibilitu a možnost komunikace s nejrůznějšími druhy periférií či modulů.

Nejrozšířenější architekturou procesoru v této kategorii je ARM pro svojí nízkou cenu a spotřebu. S architekturou ARM je úzce spjatý operačního systému Linux. Pro některé počítače na jedné desce existují i oficiální porty Androidu. Linux s sebou přináší všechny aplikace známé z běžných počítačů a pro programátory a uživatele známé prostředí. Tyto výhody pomohly k výraznému rozšíření

počítačů na jedné desce mezi programátory, kteří nemají velké zkušenosti s vestavěnými zařízeními nebo uživatele, používající skriptovací jazyky jako Python.

V robotice malé počítače na jedné desce nahrazují větší počítačové desky nebo notebooky, neboť mají dostatečný výkon pro řešení složitějších úloh. Nezabírají tolik místa a dovolují snadnější montáž přímo do robota. V robotovi představují často hlavní výpočetní jednotku, která přes komunikační sběrnice ovládá hardwarové moduly s určitou mírou inteligence. Počítače na jedné desce jsou vhodným stavebním kamenem robotů pro svůj dobrý poměr velikosti, výkonu, ceny a nízké spotřeby.

Nejběžnější zástupci této kategorie a jejich srovnání v tabulce 1.2:

- **BeagleBone [8]** je otevřený hardware, který poskytuje velké množství vstupně výstupních pinů a komunikačních sběrnic za nízkou cenu. Není tolik známý jako Raspberry PI, ale je postaven na výkonnějším procesoru, s dvěma programovatelnými jednotkami;

- **Gumstix [9] DuoVero a Overo** jsou průmyslové minipočítače s nejrůznějšími rozšiřujícími moduly a ARM procesory. Nevýhodou je vysoká cena za toto řešení;

- **Intel Galileo [10]** je kompatibilní s platformou Arduino a jejím vývojovým prostředí. Výkonou jednotkou je 32bit procesor od společnosti Intel. Programy se vytváří a nahrávají stejně jako pro oficiální Arduino desky s mikrokontrolery Atmel. Kompatibilita s Arduino zajišťuje snadný přechod od méně výkonných Arduino desek k Intel Galileo a možnost připojení rozšiřujících modulů kompatibilních s Arduino Uno R3;

- **Raspberry PI [11]** je první levný a mediálně známý počítač na jedné desce. Okolo něj se vytvořila silná komunita, zajištující snadné použití pro začátečníky a poskytující rychlá řešení při vzniku problémů. Pro svou nízkou cenu se velmi rozšířil v nejrůznějších robotických projektech.

Název	Čip	Pamět RAM	GPIO	Sběrnice	Ethernet	Cena
BeagleBone	ARM Cortex-A8 720MHz	256MB	66	UART, I2C, SPI, CAN, USB	Ano	−+
BeagleBone Black	ARM Cortex-A8 1GHz	512MB	66	UART, I2C, SPI, CAN, USB	Ano	++
Raspberry PI model B	ARM11 700MHz	512MB	8	UART, I2C, SPI, USB	Ano	++
Gumstix DuoVero	Dual-core TI 1GHz	1GB	Modul	UART, I2C, SPI	Modul	−−
Gumstix Overo	ARM Cortex-A8 až 1GHz	256MB nebo 521MB	Modul	UART, I2C, SPI	Modul	−−
Intel Galileo	Intel Quark X1000 400MHz	256MB	14	UART, I2C, SPI, USB	Ano	Není uvedena *

*V současné době se dodává především pro výukové projekty.

Tabulka 1.2: Srovnání nejběžnějších minipočítačů. Poznámka modul znamená, že danou funkcionalitu u minipočítače lze zajistit pomocí přípojných modulů.

1.1.3 Desky se SoC a FPGA

Řešení se skládá ze SoC podobně jako u počítačů na jedné desce 1.1.2, ale deska je navíc osazena ještě čipem, obsahujícím programovatelné hradlové pole (dále jen FPGA čip). FPGA čip zajišťuje ovládání hardwaru a předzpracování signálů z čidel robota. Výrobci těchto desek většinou nabízejí k hardwaru vizuální programovací prostředí pro jednoduší a rychlejší práci, jako například LabVIEW [12] od National Instruments (dále jen NI) nebo MATLAB [13], s rozšiřujícím modulem Simulink [14] od MathWorks. Pro plnou kontrolu nad chováním je možné vytvořit ovládací software přímo v jazycích pro FPGA a SoC odděleně.

Využití takovéto desky na robotovi odstraňuje nutnost vytvářet samostatné inteligentní hardwarové moduly, ovládající hardware, jako jsou motory kol, serva či senzory. Vše je připojeno přímo k této desce a veškerá ovládací logika se implementuje na FPGA čipu a mikroprocesoru. Sjednocený přístup s možností vizuálního programování usnadňuje udržitelnost programů a rychlejší učení pro začátečníky.

Správně naprogramovaná jednotka FPGA zpracovává signál ze senzorů v kratším čase než univerzální mikroprocesor a s pevně danou dobou zpracování. Mikroprocesor má proto více času na řešení jiných problému a s hardwarem robota komunikuje na vyšší úrovni přes FPGA čip. Nevýhodou těchto desek je vysoká cena za hardware i programovací software. Velikost desky je často srovnatelná s počítačovými deskami rozměrů mini-atx nebo atx. Mají i nezanedbatelný odběr z baterií robota.

Zajímavý zástupci z této kategorie jsou:

- **myRIO od NI [15]** je univerzální vestavěné zařízení vytvořené speciálně pro studenty, aby mohli navrhovat a testovat reálné komplexní systémy snadno a rychle. Ovládací programy se navrhují ve vizuálním prostředí LabVIEW. NI myRio může fungovat jako různé typy zařízení, například osciloskop nebo řídící jednotka robota. Výhodou pro robotiku jsou malé rozměry a velká vybavenost pro komunikaci s okolím;

- **Terasic SocKit[16]** je robustní hardwarová platforma s velkou návrhovou flexibilitou, založená na ARM Cortex-A9 a FPGA čipu, který jsou propojený na jedné desce pomocí vysokorychlostní sběrnice. Deska obsahuje hardware pro síťovou komunikaci, audio, video a další. Nevýhodou jsou větší rozměry desky a cena.

1.1.4 Stavebnice

Stavebnice obsahují všechno potřebné, řídící jednotku, senzory, aktuátory a spojovací prvky pro stavbu menších robotů, kteří zvládají sami řešit jednodušší robotické problémy. Jsou ideální pro základní seznámení s robotikou nebo ověřování hypotéz, neboť odstiňují od složité stavby a nízkoúrovňového ovládání hardwaru. Stavebnice umožňují postupný inkrementální vývoj, od jednoduššího robota ke složitějšímu, přidáváním senzorů a snadným dostavováním. Pokud se dané řešení neosvědčí, stavebnice poskytuje možnost rychle přepracovat celou koncepci robota v krátkém časovém úseku.

Se stavebnicemi se dodávají různá vývojová prostředí pro jednoduší ovládání a rychlejší začátky. U rozšířenějších stavebnic, např. Lego Mindstorms [18], může uživatel ovládací software tvořit jak ve vizuálním programovacím prostředí, tak i upravených a rozšířených programovacích jazycích jako je například RobotC.

V případě, že má řídící jednotka nedostatečný výkon pro řešení zadané úlohy, některé poskytují možnost bezdrátové komunikace s počítačem nebo jinou řídící jednotkou. Bezdrátová komunikace se využívá i pro přenos a následné zobrazení stavových informací z robota. Slouží také pro sledování výpočtu a odhalování chyb v algoritmech.

Známí zástupci této kategorie jsou:

- **Fischertechnic Robotics [17]** nejsou pouze stavebnice pojízdných robotů, ale i automatizačních linek. Hlavní řídící jednotku stavebnice lze snadno použít samostatně a připojit k ní vlastní senzory nebo motory;

- **Lego Mindstorms [18]** obsahuje hlavní řídící kostku, ke které se pomocí kostek připojují senzory a motory. Stavba robota probíhá velmi rychle, ale stavebnice je nevhodná pro větší roboty. Kostky lega postrádají dostatečnou pevnost spojů. Lego Mindstorms pro komunikaci se senzory definuje vlastní komunikační protokol. Nelze proto stavebnici rozšiřovat snadno o vlastní typy senzorů;

- **Vex Robotics [19]** je stavebnice podobná v ČR známe stavebnici Merkur. Kovová konstrukce umožňuje stavbu větších a robustnějších robotů. Řídící software se píše v upraveném jazyce RobotC, jenž vychází z jazyku C.

- **Mechatronic Education**[1] nabízí robotické stavebnice, zaměřující se na podporu výuky mechatroniky a elektroniky ve školách. Se stavebnicemi se dodává vývojové prostředí jBlocks, umožňující sestavování řídících programů ve vizuálním prostředí, s následnou kompilací do spustitelného programu.

1.1.5 Notebooky a standardní počítače

Ve větších robotech jsou běžné počítače zabudované, protože roboti mají dostatečný prostor pro upevnění a sílu, aby dodatečnou váhu počítače uvezli. U menších robotů jsou mimo robota a komunikují s ním bezdrátově nebo po kabelu. Obvykle slouží pouze pro zobrazování dat a úpravu softwaru robota při testování. Robot proto obsahuje jinou, menší řídící elektroniku, se kterou je schopný plně fungovat.

Největší výhodou použití počítače nebo notebooku, je vysoký výpočetní výkon, jak procesoru, tak případně grafické karty, ve spojení s velkou operační pamětí a úložným prostorem. Pro programátora poskytují známé prostředí s podporou velké řady specializovaných nástrojů pro vývoj. Problémy a neočekávané chování může vytvářet operační systém, který často není real-time. Při vysoké zátěži můžou méně kritické úlohy přerušit běh kritických a způsobit tím ztrátu důležitých informací.

Notebooky a počítače vždy komunikují s určitou mezivrstvou, která nízkoúrovňově ovládá hardware robota. Nejsou uzpůsobeny pro přímé ovládání holého hardwaru, například motorů, jak po stránce softwaru, tak i hardwaru a nemají vyvedené adekvátní komunikační sběrnice, například sériovou linku nebo CAN. Absence komunikačních sběrnic se řeší USB převodníky pro daný typ sběrnice, například USB na sériovou linku. Tyto převodníky pak v systému vytvoří virtuální komunikační sběrnici daného typu. Použití USB převodníků může znamenat neočekávané a špatně odhalitelné problémy, plynoucí z definice komunikace po USB sběrnici, například zpozdění přenosu dat. Komunikace v případě komplikovanějších robotů může být řešena propojením jednotlivých modulů ethernetem.

Velikost robota, náročnost a typ řešeného úkolu určuje jaký notebook nebo počítač lze použít. Obecně se dá říct, že menší notebooky, případně počítače s vyšší odolností a menší spotřebou budou lepší volbou, protože nezvětšují tolik váhu robota a zároveň vydrží delší dobu pracovat, než úplně vyčerpají svojí baterii nebo robotovu. Pro mobilní roboty se hodí vybírat počítače pouze s SSD

[1] Česká společnost, která v době vzniku této práce vytvářela nový informační portál. Starý portál byl nahrazen komunikací prostřednictvím skupiny mechatronic.education na sociální síti facebook.com

disky, jelikož jsou méně náchylné na mechanické otřesy a vibrace. Mezi takovéto notebooky patří:

- **Thinkpad X240 [30]** s vysokou odolností konstrukce, malými rozměry, velkou výdrží na baterie a možností připojení k internetu, prostřednictvím mobilní sítě, je ideální pro mobilní roboty. Thinkpad X240 disponuje dostatečným výkonem na řešení velké škály problémů. Jeho nevýhodou je poměrně vysoká cena;
- **Dell Latitude 3330 [31]** má odolné provedení, lehkou konstrukci a je vybaven procesory střední třídy. Může umožnit pohodlné testování chování robota a dostatečný výkon za rozumnou cenu.

1.1.6 Tablety a mobilní telefony

V robotice se jedná o málo rozšířenou kategorii, používající se spíše pro zobrazování dat nebo přímé ovládání robota, bez vyšší míry samostatného chování. Ačkoliv nejmodernější mobilní telefony disponují velkým výpočetním výkonem a různými senzory, které lze přímo a snadno využít pro robotické účely, jako je video kamera, akcelerometr, gyroskop, případně gps modul, kvalita informací z těchto senzorů často není dostatečná, aby se daly použít. Zatím se víc setkáme s použitím mobilů a tabletů jako dálkových ovladačů pro robotické hračky.

Omezením pro ně je absence komunikačních sběrnic, případně USB převodníků. Robot musí být vybaven bluetooth modulem nebo wifi spojením, aby bylo možné s robotem komunikovat. Tyto typy bezdrátových spojeních jsou však velmi náchylné na rušení a nezřídka dochází k výpadkům spojení. Ani krátké komunikační vzdálenosti, pokud vše umístíme blízko sebe v robotovi, nejsou zárukou kvalitního spojení.

U této kategorie je také nutné, jako v případě použití notebooků a počítačů 1.1.5, aby nízkoúrovňové ovládání hardwaru zajišťovaly samostatné moduly, komunikující následně s telefonem či tabletem, neboť operační systém těchto zařízení není přizpůsoben pro real-time ovládání.

Využití těchto zařízení pro interakci mezi člověkem a robotem ještě není příliš prozkoumanou oblastí. Mobilní telefony nebo tablety mohou poskytnout jednotný způsob ovládání, protože je vlastní velké množství lidí. Pro jejich menší rozměry, nižší hmotnost a nezávislost na dalším zdroji energie se dají použít na menších robotech. V případě poruchy nebo potřeby je můžeme snadno vyměnit za jiný kus, protože nejsou nijak složitě propojené s robotem.

Zajímavé mobilní telefony nebo tablety jsou:

- **Google Nexus 5 [32]** jedná se o referenční telefon od firmy Google, ukazující možnosti současných technologií s kvalitní výbavou. Na rozdíl od jiných telefonů má jako operační systém čistý Android;
- **Evga Tegra NOTE 7 [33]** je 7palcový tablet. Tablet je postavený na procesoru NVIDIA Tegra 4. Větší rozměry displeje tabletu poskytují velkou plochu pro zobrazení informací o stavu robota;
- **NVIDIA Shield Tablet [34]** využívá technologie Tegra K1 s 192 grafickými jádry architektury Kepler. Tegra K1 je jediný mobilní GPU, podporující technologii NVIDIA CUDA, pro obecné výpočty na grafickém čipu.

Kategorie	Výkon	Paměť	Rozšiřitelnost	Velikost	Ovládání hardwaru	Cena
Desky založené na mikrokontrolerech	−−	−−	++	++	++	++
Jednodeskové počítače, SoC	+	+	+	++	+−	+
Desky se SoC a FPGA	+	+	++	−	++	−−
Stavebnice	−+	−	−	+	−−	+−
Tablety a mobilní telefony	+	+	−	+−	−−	+−
Notebooky a počítače	++	++	+−	−−	−	−

Tabulka 1.3: Porovnání kategorií se zaměřením na robotické využití.

1.1.7 Srovnání kategorií hw platforem

Každá, z výše uvedených hw platforem, má své využití v robotických projektech. U menších robotů, jako jsou sledovači čáry, má velikost a váha větší důležitost, než vysoký výpočetní výkon. Takoví roboti mají jako řídící jednotku desky s mikrokontrolery bez operačního systému, s programem psaným přímo pro určitý mikrokontroler. Úplným opakem jsou objevující se, plně automatické automobily, které mají dostatečný úložný prostor se zdrojem energie. Kvalitní zdroj energie a dostatek místa dovolil, že se zde nasazují standardní počítačové desky pro svůj vysoký výpočetní výkon, který je potřebný pro bezproblémový chod.

Pro každou z dříve uvedených hw platforem, existuje v robotice problém, pro jehož řešení je vhodná. Více typů hw platforem může být adekvátních pro řešení stejného problému a je na řešiteli jakou zvolí. Tabulka 1.3 poskytuje základní srovnání hw platforem mezi sebou, se zaměřením na robotické použití. Žádná z uvedených hw platforem není výhodná pro všechny robotické projekty. Často nejlepší řešení poskytuje kombinace více hw platforem pro dosažení požadovaného výsledku. Srovnávací tabulka se hodí pro použití při rozhodování, jaké hw platformy pro svůj robotický projekt vybrat.

1.2 Operační systémy

Zvolený operační systém zásadně ovlivňuje výsledný výkon robota. Při špatné volbě nebo nastavení, může způsobit nefunkčnost nebo neočekávané chování robota. Z pohledu robotiky dělíme operační systémy podle jejich schopností zajistit včasnost výsledků jednotlivých procesů. To je schopnost zajištění splnění mezních časů zpracování úkolů neboli dodržení deadline. Běžně používané operační systémy, jako je Windows či Linux, byly navrženy jako víceuživatelské systémy pro obecné použití. Cíle takovýchto systémů jsou v přímém rozporu s cíly realtime operačních systémů. Systémy obecného použití jsou vyvíjené pro co nejvyšší propustnost za cenu latence, naopak realtime systémy, které jsou velice

specializované, jsou vyvíjeny s důrazem na splnění časových požadavků za cenu nižší propustnosti.

U slabších mikroprocesorů a většiny mikrokontrolerů, použitých jako řídící jednotky nebo ve specifických případech, se operační systém nepoužije. Operační systém se na řadě mikrokontrolerů ani nevejde do paměti nebo pro řešení daného úkolu neposkytuje žádné výhody, jen spotřebovává systémové prostředky (jako výpočetní výkon a paměť) pro vlastní chod.

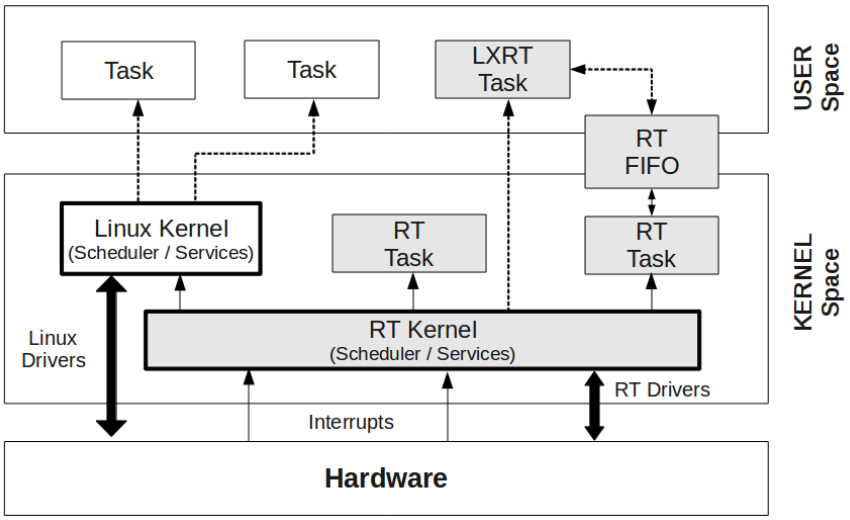
1.2.1 Linux

Použití Linuxu se v robotice rozšířilo pro jeho konfigurovatelnost, otevřenost a možnost změn. V základním nastavení Linux není realtime z důvodů zmíněných ve srovnání OS 1.2.5, a proto je vhodný pro zpracování časově nekritických úloh, jako je zobrazení informací o stavu robota, interakce s uživatelem a podobné úlohy. Zároveň ale byly vyvinuty různé postupy, jak určitou míru realtime chování zajistit. Nejčastěji se lze setkat se třemi odlišnými přístupy. První přístup je upravení samotného jádra Linuxu, aby se chovalo více deterministicky, zrychlily se odezvy jádra a aby procesy jádra, které nejsou v kritické sekci kódu, byly přerušitelné. Druhý přístup využívá RTOS, který plně ovládá hardware a Linux je spuštěn, jako realtime proces s nejnižší prioritou, případně jako nečinný proces, jenž je naplánován pouze tehdy, když žádný jiný realtime proces není připraven. Poslední variantou je využití nanojádra, jako je technologie ADEOS [21] (adaptivní doménové prostředí pro operační systémy), která poskytuje flexibilní prostředí pro sdílení hardwaru mezi více operačními systémy. Tyto technologie jsou podrobněji popsány v následujících podkapitolách.

Úprava jádra

Nejjednodušší způsob, jak dosáhnout částečného realtime chování, je pomocí aplikace takzvaného preemption patch na standardní Linuxové jádro. Od Linuxového jádra verze 2.6 je tento patch zařazen do hlavní vývojové větve, jako volba přerušitelné jádro při konfiguraci. Při povolení této volby se změní chování jádra následovně:

- Procesy v kritických sekcích jádra, chráněných pomocí spinlocků, jsou přerušitelné, pokud není explicitně řečeno, že se nemají přerušovat. Naopak procesy v kritických sekcích běžného Linuxového jádra nelze přerušit;
- Přeměna obsluhy přerušení na běžné procesy jádra, kterým lze měnit prioritu, případně je přeplánovat, když je připraven proces s vyšší prioritou. Obsluha přerušení u standardního Linuxové jádra má nejvyšší prioritu zpracování a nelze tuto prioritu měnit, ani přeplánovat, kdy se obsluha přerušení bude vykonávat;
- Jiný plánovač procesů, který má časovou složitost $O(1)$ místo běžného Linuxového plánovače s časovou složitostí $O(n)$, který pro nalezení procesu s nejvyšší prioritou musí procházet celé pole čekajících procesů;
- Změna API starého Linuxového časovače do samostatných struktur pro časovač jádra s vysokým rozlišením, který lze použít i v uživatelském prostředí a časovač pro timeout.



Obrázek 1.1: Znázornění architektury abstrakce přerušení.

Použití přerušitelného jádra poskytuje výhody, jako snížení maximálního zpoždění (latency) procesů z řádu desítek milisekund pro standardní Linuxové jádro, na 1 až 2 milisekundy, při použití přerušitelného jádra. Zároveň lze všechny procesy s realtime požadavky spouštět v uživatelském prostředí stejně jako běžné aplikace a využívat známé Linux/Posix API. Spuštění realtime procesů v uživatelském prostředí také zajišťuje ochranu paměti a při chybě v realtime procesu to neovlivní funkčnost jádra systému.

Úpravy snižují zpoždění vykonávání procesů, ale stále je zde dost nedeterminismu, takže nelze z Linuxového jádra udělat plně realtime jádro. Jsou úlohy pro něž je takto upravený Linux nedostatečný a musí se použít jiného řešení, které zajišťuje lepší realtime chování.

Linux jako proces RTOS s nejnižší prioritou

Jiný přístup, jak zajistit realtime časování s Linuxem, je rozlišit, co vyžaduje realtime chování a co nikoliv. Použije se malé realtime jádro, ve kterém se jako proces s nejnižší prioritou spustí Linux. Zpracování úloh s realtime požadavky probíhá jako samostatné procesy s vyšší prioritou na realtime jádru. Úlohy, jež nevyžadují realtime chování, jako grafický výstup či obsluha sítě, jsou zpracovávány běžným Linuxovým jádrem.

Tento přístup je nazýván 'abstrakce přerušení', protože realtime jádro přebírá obsluhu přerušení od Linuxového jádra. Základní chování je znázorněno na obrázku 1.1. Realtime jádro zachytává všechna přerušení, než se dostanou do Linuxu. Linux již nemá přímou kontrolu nad povolováním a zakazováním přerušení. Veškerou obsluhu přerušení zajišťuje realtime jádro. Při příchodu přerušení se rozhodne, zda přerušení má obsloužit realtime proces. Případně, pokud Linux má tento druh přerušení povolený, zavolá adekvátní obsluhu přerušení v Linuxu.

Realtime jádro poskytuje prostředky, jako FIFO nebo sdílenou paměť, umožňující komunikaci mezi realtime procesy a procesy spuštěnými v uživatelském prostředí.

Malé realtime jádro dovoluje zpracovat analýzu vykonávaných procesů a určit tak horní hranici zpoždění zpracování jednotlivých realtime procesů. Tento přístup také potřebuje úpravu Linuxového jádra, ale v menším rozsahu než předešlý způsob. Nejrozšířenější implementace abstrakce přerušení poskytují obalení pro nativní API POSIX vláken. Je možné nejdříve realtime proces testovat v uživatelském prostředí, s použitím standardní POSIX knihovny pro vlákna, podporou testování pomocí běžných nástrojů, jako gdb a následně otestovaný proces přenést do realtime prostředí.

Realtime procesy jsou spuštěné v jádře, což znamená, že časy odezvy budou velmi nízké (typicky pod 10 mikrosekund), ale pokud proces selže, často to naruší správnou funkčnost celého jádra. Podpora různých hardwarových platforem je nižší než u Linuxu a vyžaduje složitější instalaci na cílový hardware.

Nanojádro

Přímo na hardwaru je spuštěno nanojádro ADEOS [21] (adaptivní doménové prostředí pro operační systémy), které poskytuje sdílení hardwarových prostředků pro více operačních systémů. Každý operační systém je spuštěn ve vlastní doméně nad kterou má plnou kontrolu a operační systém ani nemusí vedět o ADEOSu.

Základní architektura je znázorněna na obrázku 1.2 a ukazuje čtyři druhy komunikace relevantních pro ADEOS. Komunikace A reprezentuje přístup do normální paměti a vstupně výstupní operace prováděné operačním systémem nezávisle na ADEOS. Komunikace B znázorňuje příjem řídících signálů od hardwaru jako výsledek hardwarových či softwarových přerušení. Také i posílání příkazů od ADEOS pro ovládání hardwaru. ADEOS používá komunikaci C pro předání přerušení operačnímu systému. Poslední komunikace D je obousměrná mezi doménou operačního systému a ADEOS, jenž lze použít pro sdílení prostředků.

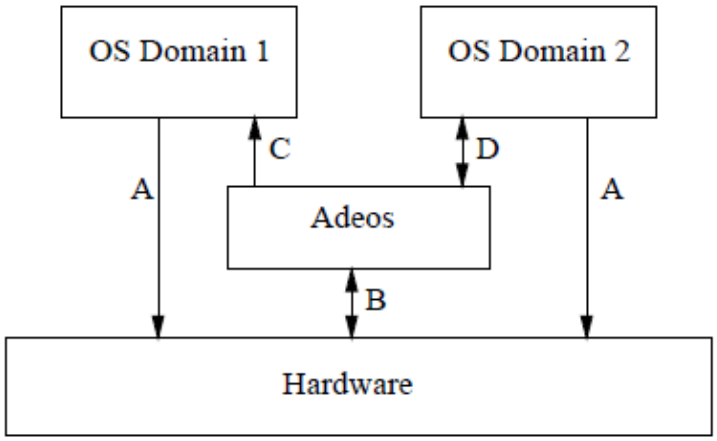
Pro předávání přerušení ADEOS využívá frontu domén operačních systémů, kde lze domény uspořádat podle nutnosti získávat hardwarové přerušení jako první. Každá doména může přijímat, ignorovat, zahazovat nebo případně ukončit přerušení.

Nanojádro ADEOS je spuštěno jako jádrový modul Linuxu a upraví tabulku globálních deskriptorů, kde sníží Linux na PL1 a samo dál běží na úrovni PL0. Mezi nejznámější implementace, používající tento princip, patří RTAI [22] nebo Xenomai [23], jenž poskytují plný realtime chování pod Linuxem. Nevýhodou je nutnost upravit jádro Linuxu a jiné API při psaní aplikací.

1.2.2 Windows

Stejně jako Linux, nemá Windows pro osobní počítače, podporu realtime chování. Platí zde stejná omezení ze srovnání OS 1.2.5, týkající se stránkování, přerovnávání vstupně výstupních operací atd. I přes tato omezení se na robotech operační systém Windows používá v kombinaci s osobními počítači, pro zobrazení informací, komunikaci s uživatelem nebo zpracování méně kritických úloh, jako rozpoznávání obrazu z kamery.

Pokud je vyžadováno realtime chování, existuje řešení, nazývané realtime rozšíření (dále jen RTX), používané v komerční sféře, kombinující Windows s realtime subsystémem nebo Microsoft poskytuje speciální OS Windows Embedded



Obrázek 1.2: Znázornění architektury ADEOS.

Compact 2013 [35] či Windows Embedded Automotive 7 [36] pro vestavěná zařízení.

Windows Embedded Compact 2013

Windows Embedded Compact 2013 je speciální operační systém od Microsoftu, optimalizovaný pro zařízení s malou pamětí. Poskytuje realtime podporu pro x86 i ARM architekturu. Zároveň integruje známé nástroje pro vývoj aplikací, jako Visual Studio 2012 a 2013. Disponuje flexibilní architekturou pro podporu velké škály hw řešení. Je zaměřen na vysokou míru bezpečnosti a možnosti bezdrátového propojení.

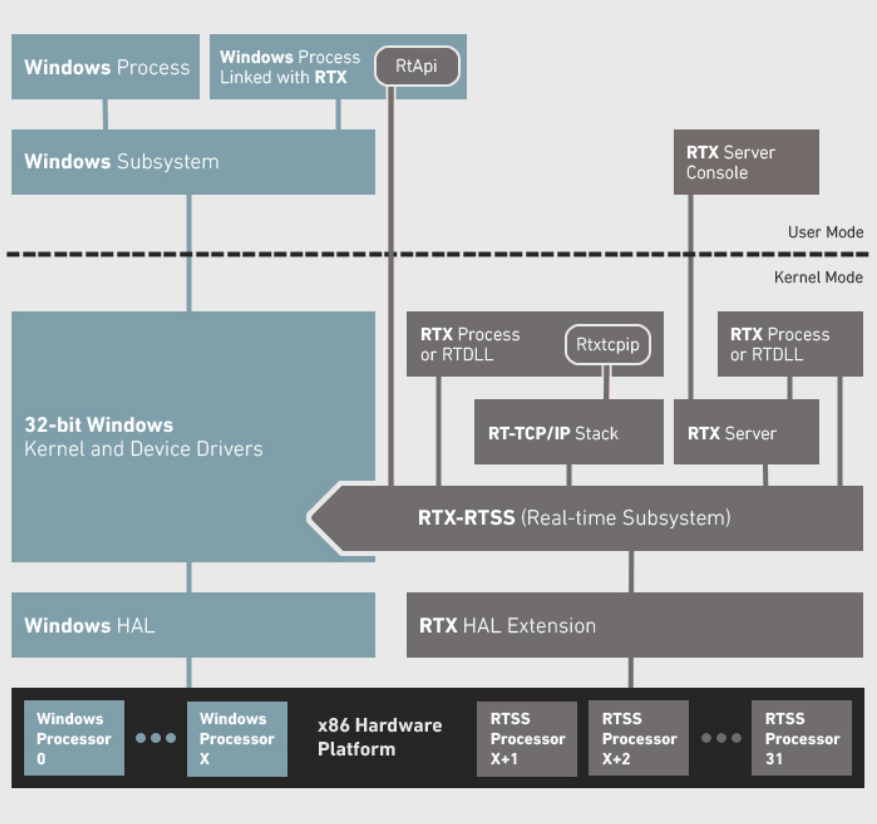
S OS Windows pro stolní počítače nebo notebooky má společnou podmnožinu nejčastěji používaných Win32 API funkcí. Sdílení API funkcí zaručuje možnost použít programy a programovací prostředky třetích stran. Významné je i snížení času vývoje aplikací a času nutného na naučení. Jako běžné Windows přímo podporuje různé druhy hw zařízení a množinu dostupných sw modulů.

Nevýhodou je uzavřenost systému, bez možností velkých změn, v kódu systému a tím přizpůsobení na specifické požadavky. Zároveň není poskytován zdarma, takže pro použití si uživatel musí koupit licenci.

RTX

RTX řešení rozšiřuje hardwarovou abstrakční vrstvu Windows a přidává realtime subsystém (dále jen RTSS), který plánuje a řídí všechny RTSS procesy nezávisle na Windows. V případě nasazení na jednoprocesorový systém sdílí RTX a Windows tento procesor. Realtime subsystém dává RTSS procesům vyšší prioritu před procesy či funkcemi operačního systému Windows. RTX poskytuje pro komunikaci s ovládanými prvky deterministické protokoly, jako EtherCat [24].

Při nasazení na systém se symetrickým multiprocesingem se rozdělí jádra procesorů mezi RTSS a Windows. RTSS plánuje úlohy na přidělená procesorová jádra, kde jsou vykonávány bez zásahů ze strany operačního systému nebo procesů Windows. Na obrázku 1.3 je znázorněna architektura RTX při použití



Obrázek 1.3: Znázornění architektury RTX.

symetrického multiprocesingu. Obrázek byl převzat ze stránek společnosti IntervalZero [26].

Vývoj aplikací pro RTX lze vykonávat ve vývojových prostředí jako Visual Studio, za použití běžných programovacích jazyků. Při nedostatku výkonu RTSS se dá přiřadit RTSS více procesorových jader nebo jednoduše celý systém přesunout na výkonnější stroj bez změny v kódu. Nasazení RTX umožňuje využití všech výhod poskytovaných platformou x86 nebo snadnou přenositelnost na jiné podporované platformy, například ARM. Nevýhodou je neexistence opensource implementace. Zakoupit lze komerční placené produkty, jako například Intime od tenAsys [25] nebo RTX od IntervalZero [26].

1.2.3 Realtime operační systémy

Realtime systém je systém, který reaguje na vnější události, vykonává funkce založené na nich a poskytuje reakce v rámci určitého času. Správnost funkce nezáleží pouze na správnosti výsledku, ale i na jeho včasnosti. Tím se liší od běžných operačních systémů, jako je Windows či Linux, kde nezáleží tolik na času kdy proces skončil, ale pouze na správnosti výstupu. Realtime operační systémy (dále jen RTOS) nemusí být výkonné, aby zajistily požadované vlastnosti. Naopak část výkonu je spotřebovávána na zajištění realtime vlastností.

Realtime operační systémy můžeme rozdělit na dvě skupiny podle požadavku na zajištění dokončení procesu v určitý čas (deadline). První skupina je částečně realtime (soft realtime) a systém se snaží, aby proces většinou skončil do určitého času (záruky jsou přibližné). Druhá skupina je plně realtime (hard realtime)

a všechny záruky jsou deterministické. Pro plně RTOS, pevně danou sadu procesů a zvolený typ plánovače, je možné provést analýzu, zda všechny procesy dokončí své zpracování do určeného času.

RTOS je často pouze sada nástrojů, jako například plánovač, struktury pro zamykání a funkce pro komunikaci mezi vlákny, ke kterým se musí připojit vlastní procesy řešící danou úlohu. Vzniklý celek se následně kompiluje do výsledného operačního systému s požadovanou funkcionalitou pro daný hardware a při každé změně ve vlastních procesech se musí celý RTOS překompilovat. Zkompilované systémy jsou hardwarově specifické a pro nasazení na jiném hardwaru se musí minimálně překompilovat. Proto se nejčastěji setkáváme s malými RTOS systémy. U složitějších systémů se využívá kombinace RTOS s Windows či Linux. Zástupci této kategorie jsou FreeRTOS, RTEMS nebo eCos.

1.2.4 Bez operačního systému

Hardware, jako je deska s mikrokontrolerem Atmega s malou pamětí pro program nebo řešení velmi specifických či jednoduchých úloh, vyžaduje psát řídící program přímo, bez podpory operačního systémů. Nad vzniklým řídícím programem máme plnou kontrolu, víme kdy je která část kódu zpracovávána a kolik výpočetního výkonu spotřebují jednotlivé části. Přímé psaní kódu pro specifický hardware umožňuje maximální využití výkonu dané platformy a dovoluje psát programy pro zpracovávání úloh s vysokými nároky na časování, například počítání otáček kol z enkodérů. Operační systém by v těchto případech přinášel zbytečnou složitost, spotřebovával výpočetní výkon a operační paměť, i když by neposkytoval výhody navíc.

Nevýhodou je kód specifický pro určitý typ hardwaru s nízkou přenositelností na jiný druh hardwaru. Vyplývá z toho nutnost při přenosu na jiný druh hardwaru část kódu upravit či přepsat a v nejhorším případě, při špatně napsaném kódu celý řídící kód přepsat. Nemluvě o podmínkách pro testování řídícího kódu za běhu, kde jsou velmi omezené možnosti ladění. U mikrokontrolerů se můžeme setkat s využití led diod pro indikaci stavu řídícího kódu nebo posílání stavových kódu po komunikační sběrnici, jako je sériová linka do připojeného počítače. Takovéto způsoby ladění však ovlivňují samotné vykonávání řídícího programu, změnou času vykonávání jednotlivých částí kódu, který se při odebrání ladících technik sníží nebo i funkčnost programu. Přidání ladících technik do programu může zapříčinit jiné chování než bude mít výsledný řídící program bez nich.

1.2.5 Srovnání operačních systémů

Robotika řeší velmi široké spektrum problémů a úloh, od sledování čáry robotem, přes průmyslové, až po roboty, kteří mají za úkol asistovat lidem v běžném životě. Jednodušší úlohy se dají snadno řešit pár řádkami kódu, propojujících vhodné knihovny bez nutnosti podpory funkcí operačního systému. Jiné úlohy jsou už komplexnější, s nutností přesné komunikace mezi různými moduly, pevně danými omezeními na časování a vyžaduje se provést kontrolu požadavků u vzniklého systému. Pro všechny typy takovýchto úloh je vhodné použít RTOS. Příkladem úlohy vyžadující přesné časování a komunikaci je řídící systém auta, letadla nebo nemocničních přístrojů. Kde zpoždění reakce na událost nebo špatné

načasování, může vést k vážným úrazům nebo smrti člověka. Naopak u úloh zahrnujících interakci s člověkem je vyžadován nějaký způsob komunikace, ať zvukové nebo vizuální, kde nehraje čas tolik důležitou roli, protože vnímání člověka není tak rychlé a dovoluje určité prodlení mezi požadavkem a odpovědí. K čemuž se dobře hodí současné operační systémy s možností snadné tvorby uživatelských rozhraních.

Řešení robotických úloh v sobě zahrnuje často více podúloh a každá z nich může mít jiné požadavky na včasnost výsledků. Výsledek zpracování obrazu z kamery může mnohokrát chvíli počkat, na rozdíl od řízení plnění motorů pro dosažení správné rychlosti. Proto nás velmi zajímá při tvorbě řídících systémů, jak se jednotlivé typy operačních systémů chovají vzhledem k včasnosti výsledků jednotlivých procesů řídícího systému běžících v rámci daného typu OS. Nezřídka se můžeme proto setkat s kombinací více typů operačních systému v rámci jednoho robota, neboť různé typy operačních systémů mají různé vlastnosti ohledně plnění časových požadavků. Požadavek na včasnost procesů v rámci daného typu OS znázorňuje obrázek 1.4 jako funkci ceny závislé na času vydání výsledku procesem. Obrázek byl převzat z předmětu Embedded and Real-Time Systems vyučovaného na Univerzitě Karlově.

Systémy jako jsou Windows nebo Linux nelze z následujících důvodů považovat za realtime (tyto důvody byly převzaty z [20]):

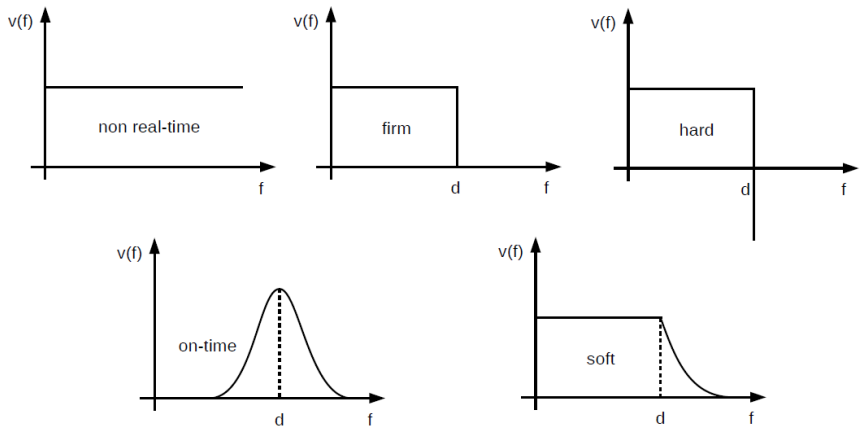
- **Hrubozrnná synchronizace** - pokud proces vstoupí do jádra, nelze ho přerušit do té doby, než je připravený jádro opustit. V případě, že nastane událost, tak proces pro její zpracování nelze naplánovat, dokud aktuálně vykonávající proces neopustí jádro. U některých systémových volání to můžou být desítky milisekund;

- **Stránkování** - proces přemisťování stránek z a do virtuální paměti není časově omezen. Není způsob, jak zjistit, kolik času zabere načtení stránky z disku. Nelze tedy určit horní časovou hranici, po kterou bude proces pozastaven při výpadku stránky;

- **Rovnost plánovače** - pozůstatek z dob více uživatelského časového sdílení. Plánovač se snaží všechny procesy plánovat férově. Proto naplánuje na procesor proces, který má nízkou prioritu, ale čekal dlouho, místo procesu připraveného pro vykonávání s vysokou prioritou;

- **Přerovnávání požadavků** - systém přerovnává vstupně výstupní požadavky, od více procesů, aby lépe využil hardware. Například blokové čtení z pevného disku procesu s nízkou prioritou bude zpracováno dříve, než požadavek na čtení od procesu s vyšší prioritou, aby se minimalizoval posun čtecí hlavy;

- **Dávkové zpracování** - systém se snaží provádět dávkové zpracování operací, aby využíval co nejlépe zdroje. Místo uvolnění jedné stránky systém počká, až jich bude víc a pak uvolní co nejvíce stránek. Což pozdrží vykonávání všech ostatních procesů.



Obrázek 1.4: Kritičnost časování u různých typů OS, znázorněná funkcí ceny, závislé na času vydání výsledků. OS bez časových požadavků jsou non realtime. Na opačné straně jsou naopak realtime systémy, jenž vyžadují stoprocentní splnění časových požadavků svých procesů.

1.3 Robotické frameworky

Robotikou se zabývá velké množství lidí a do dnešních dnů vzniklo nepřeberné množství podpůrných projektů (frameworků), pro usnadnění řešení základních problémů. Některé se částečně překrývají, případně jsou cíleny na různé druhy problému nebo vytvářejí sjednocující prostředí pro znovuvyužití a sdílení kódu mezi více lidmi. Společné pro všechny robotické frameworky je ale snaha urychlit vývoj a testování robotů.

Tyto projekty se dají rozdělit do dvou skupin podle způsobu programování, a to na vizuální a textové. Vizuální je založeno na grafickém prostředí, kde se propojují prvky s přesně definovanými rozhraními a následně po namodelování celého řešení se grafická podoba automaticky převede na spustitelný kód. Textové je naopak celé založené na textovém programování v nějakém běžném vyšším jazyce například C/C++, Python, Java, Lisp atd., kde frameworky poskytují knihovny pro základní ovládání a řešení nejčastějších problémů. Některé poskytují i vlastní vývojová prostředí.

1.3.1 Vizuální

Hodně vizuálních prostředí je zaměřeno na osoby bez znalostí programování, spíše pro vědecké pracovníky nebo děti podle typu a schopností grafického rozhraní. Existují ale i vysoce specializované, určené pro průmyslové použití, kde se schopnost programovat předpokládá. Vizuální prostředí usnadňuje a urychluje učení základů, zároveň jsou některé produkty dostatečně propracované i pro tvorbu velmi komplexních systémů. Uživatel ve vizuálním prostředí používá vyšší úroveň abstrakce.

Společnosti nebo komunity, vytvářející vizuální prostředí, poskytují roboty, kteří jsou připraveni pro použití s daným vizuálním prostředím. Uživatel je odstíněn od nízkoúrovňového ovládání robota nebo technických detailů, jakým způsobem se nahrává kód do robota. Prostředí obsahuje knihovny pro nízkoúrovňové ovládání robota a zavádí určitou míru abstrakce. Proto lze stejný kód použít na

více typech robotů nebo místo reálného robota použít simulátor.

Vizuální forma programu není pro všechny typy úloh vhodná, vyžaduje velký monitor pro přehledné zobrazení a sledování propojovacích čar mezi prvky nemusí být přehledné. Zároveň, pokud uživatel přejde k produktu jiné společnosti, se musí učit jiný způsob práce a nemůže použít své hotové projekty z jiného produktu.

Následuje pár ukázkových vizuálních prostředí vhodných pro tvorbu robotických projektů:

- **MATLAB [13] a Simulink [14]** je vizuální programovací prostředí s vlastním vizuálním jazykem, umožňující tvorbu a simulaci modelu s analýzou výsledků. Modely lze vytvářet jak lineární tak i nelineární případně hybridní. Prostředí je zaměřeno hlavně na vědecké účely. V základu podporuje širokou škálu cílových hardwarů jako Arduino, Lego NXT nebo Raspberry PI;

- **NI LabVIEW [12]** má vlastní vizuální jazyk G pro tvorbu modelů. Disponuje velkým množství doplňujících modulů, jako například modul pro zpracování signálu. NI umožňuje vytvářet vizuální nadstavby nad vývojovým prostředím nebo vizuálním jazyce. Výsledný vizuální model je převeden do jazyka C/C++ nebo HDL dle cílového hardwaru. S LabVIEW je možné zakoupit kompatibilní robotický hardware připravený k okamžitému použití;

- **LEGO NXT-G** je výsledkem spolupráce LEGO a NI a je základním programovacím nástrojem pro LEGO Mindstorms NXT postaveným nad LabVIEW od NI. Důraz je kladen na intuitivnost a jednoduchost celého vývojového prostředí, včetně procesu programování. S LEGO NXT-G můžou pracovat i žáci základních škol bez zkušeností s programováním.

1.3.2 Textové

Textová prostředí očekávají určitou míru programátorských zkušeností, případně jsou poskytovány jednoduché příklady, na kterých se lze velmi rychle naučit základům. Některé projekty mají vlastní vývojová prostředí, jako například Arduino nebo se jedná o sady knihoven, zavádějící vyšší míru abstrakce a sjednocující API pro různé nízkoúrovňové hardware. Větší z projektů, jako například ROS, poskytují i celý ekosystém na znovupoužití částí kódu.

Pro uživatele s programátorskými zkušenostmi znamená použití těchto frameworků jen přidání knihoven do projektu a psaní kódu ve svém oblíbeném jazyce jako je C/C++. Projekty poskytují většinou programátorské rozhraní pro nejznámější vyšší programovací jazyky. Případně existuje více podobných projektů pro různé programovací jazyky, mající stejné vlastnosti.

Následuje pár ukázkových textových frameworků, vhodných pro tvorbu robotických projektů:

- **Arduino [3]** je vývojové prostředí založené na projektu Processing s vlastními hardwarovými moduly. Řídící kód se píše přímo pro hw modul v jazyce C/C++, kde běží bez operačního systému. Na internetu a v rámci projektu Arduino existuje nepřeberné množství ukázkových programů a knihoven pro snadné a rychlé použití běžné elektroniky, jako lcd displeje, teplotní čidla

nebo serva. Nevýhodou vývojového prostředí je, že neposkytuje funkcionalitu pro ladění chyb napsaného kódu;

- **Microsoft Robotics Developer Studio** je prostředí pro Windows pro řízení a simulaci robotů. Využívá .NET knihovnu pro asynchronní paralelní úkoly a primárním programovacím jazykem je C#. Studio umožňuje psát programy i ve vizuálním prostředí s následným propojením se simulátorem robotů nebo reálnými roboty;

- **ROBOTC [27]** je robotický programovací jazyk, založený na jazyku C s rozšířením pro snadné robotické použití. Obsahuje knihovny pro ovládání motorů, serv atd. S jazykem je dodáváno i vývojové prostředí, které má pokročilejší funkce, jako dokončování názvů, ladění programů nebo simulátor robotů ve virtuálním prostředím. ROBOTC se používá pro psaní kódů pro reálné roboty, například LEGO Mindstorms NXT, VEX Cortex a jiné;

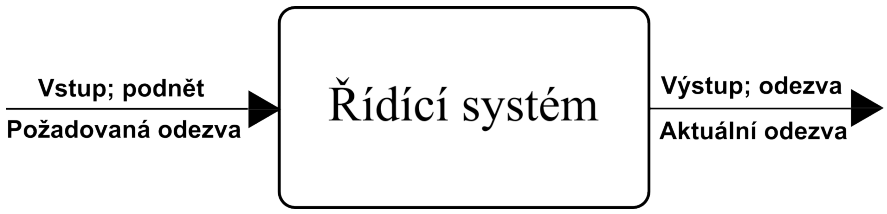
- **ROS [28]** je open source množina softwarových knihoven a nástrojů zjednodušující vývoj robotických aplikací. Výsledné programy jsou založeny na grafové struktuře propojující výpočetní uzly, kde uzly zpracovávají vlastní informace a přijímají nebo odesílají informace (zprávy) do ostatních uzlů. ROS zavádí vlastní systém instalace balíčků, čímž usnadňuje znovupoužití těchto balíčků a udržuje si databázi o již vytvořených balíčcích. V rámci balíkovacího systému je řešena integrace se simulátorem robotů Gazebo. Hlavním programovacím jazykem je C++ a Python;

- **URBI [29]** je open source platforma pro ovládání robotů a komplexních systémů. Celá platforma je založena na C++ komponentové knihovně UObject, od které se odvozují jednotlivé výpočetní jednotky robota. Dále je jde vlastní skriptovací jazyk urbiscript, kterým se může nastavovat propojení mezi UObjecty a psát programy řízené na základě asynchronních událostí a plně paralelně. Pro známější roboty (Aibo, Pioneer) jsou vytvořeny komunikační UObjecty přes, které se dají tito roboti ovládat.

1.4 Řídící systém

Řídící systém se skládá z podsystémů a procesů sestavených za účelem ovládání výstupů z procesů [37]. Například rychlost otáčení kola robota jako výsledek ovládání velikosti napětí na motoru. V tomto procesu podsystémy, zvané výkonová část a ovladače výkonové části jsou použity pro regulaci rychlosti otáčení kola kontrolováním výstupního signálu z enkodéru, umístěného na kole. Řídící systém ve své nejjednodušší formě poskytuje výstup nebo odezvu pro daný vstup nebo podnět, jak je znázorněno na obrázku 1.5.

Řídící systémy nám dovolují pohybovat s manipulátory a jinými zařízeními s vysokou přesností, které bychom jinak nemohli dosáhnout. Roboti, navrhovaní podle principů řídících systému, mohou kompenzovat lidskou nedokonalost. Dále řídící systémy jsou užitečné v nebezpečných a vzdálených místech. Například vzdáleně řízená ruka robota může manipulovat věcmi v radioaktivitou zamořeném prostředí. Jsou vhodné pro změnu typu vstupu a dokáží kompenzovat vnější



Obrázek 1.5: Zjednodušený popis řídícího systému.

rušení. Typicky v robotice kontrolujeme proměnné, jako rychlost a pozici. Systém musí udržovat správnou hodnotu i v případě vnějších rušení. Předpokládejme systém, který udržuje rychlost otáčení kol robota, nastavováním výkonu motorů. Pokud robot pojede do kopce a následně po rovině nebo po různě přilnavém povrchu. Systém musí být schopný detekovat vnější změny a upravit výkon motorů, aby dosáhl požadované rychlosti. Očividně se nezmění vstup systému, aby se provedla správná změna. Současně musí systém sám měřit vnější rušení a změny, aby mohl správně nastavit výkon motorů pro dosažení požadované rychlosti dané vstupem.

Jak bylo zmíněno dříve, řídící systém poskytuje výstup nebo také odezvu na daný vstup či podmět. Vstup reprezentuje požadovanou odezvu. Zatímco výstup je aktuální odezva. Dva faktory tvoří výstup rozdílný od vstupu. Za prvé, změna vstupu je okamžitá, oproti změně výstupu, která je u fyzických entit postupná, neboť nedokáží měnit svůj stav okamžitě. Přesnost dosažení požadované odezvy je druhý faktor, způsobující rozdílnost výstupu od vstupu.

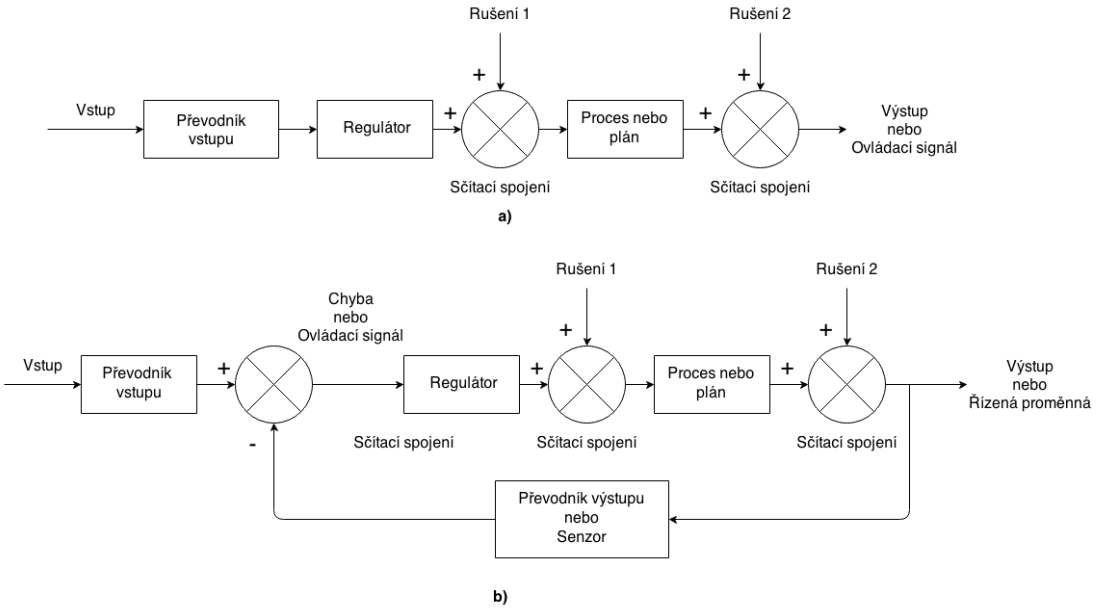
Existují dvě hlavní konfigurace řídících systémů: otevřená smyčka a uzavřená smyčka.

1.4.1 Systém s otevřenou smyčkou (direktivní)

Obecný systém s otevřenou smyčkou je ukázán na obrázku 1.6(a). Otevřená smyčka začíná s podsystémem, zvaným převodník vstupu, který převádí vstup na jiný typ zpracovávaný regulátorem. Regulátor provádí proces nebo plán. Vstup je občas nazýván referencí, zatím co výstup může být nazýván kontrolovanou proměnnou. Celý systém je dále zatížen rušením na vstupu do regulátoru i na výstupu z něj. Toto rušení negativně ovlivňuje výstup. Hlavní odlišností systému s otevřenou smyčkou od zpětnovazebního (uvedeného níže) je nemožnost provádět kompenzace rušení přidaného do řídícího systému.

1.4.2 Systém s uzavřenou smyčkou (se zpětnou vazbou)

Nevýhody systémů s otevřenou smyčkou, jmenovitě citlivost na rušení a neschopnost oprav těchto rušení, mohou být překonány systémy s uzavřenou smyčkou. Obecný systém s uzavřenou smyčkou je ukázán na obrázku 1.6(b). Výstupní převodník nebo senzor měří výstupní odezvu a převádí jí do formy používané regulátorem. Změřený výstup je přes zpětnou vazbu přiveden zpět na vstup, kde je odečten od vstupního signálu a vznikne ovládací signál. Nicméně u systémů, kde vstupní a výstupní převodník mají jednotkový zisk (převodník svůj vstup násobí 1), je ovládací signál roven rozdílu mezi vstupem a výstupem. V takovém případě je ovládací signál nazýván chybou.



Obrázek 1.6: Blokový diagram řídícího systému.
a. systém s otevřenou smyčkou
b. systém s uzavřenou smyčkou

Systém s uzavřenou smyčkou kompenzuje rušení měřením výstupní odezvy, vrací zpět toto měření pomocí zpětné vazby a porovnává tuto odezvu se vstupem. Pokud je zde nějaký rozdíl, systém provádí plán přes ovládací signál pro provedení korekce. Jestliže není žádný rozdíl, systém neprovádí plán, poněvadž odezva již má požadovanou hodnotu.

Systémy s uzavřenou smyčkou mají očividně výhodu vyšší přesnosti, než systémy s otevřenou smyčkou. Nejsou tolik citlivé na šum, rušení a změny v prostředí. Na druhou stranu jsou složitější a dražší než systémy s otevřenou smyčkou.

1.5 Vztah mezi OS, frameworky a řídícími systémy

Propojení a důležitost OS, frameworku a řídícího systému se projeví funkčností výsledného řešení problému. Operační systém představuje důležitou, avšak často opomíjenou část, jež při důkladném výběru, zvážení všech požadavků a omezení umožňuje subsystémům řídícího systému získat maximální užitek ze zvoleného hw řešení při dodržení všech požadavků a omezeních.

Propojovacím mechanizmem a základními stavebními kameny subsystémů jsou často frameworky jako ROS [28], URBI [29] atd. Frameworky poskytují základní komunikační mechanizmy, hotová řešení nejběžnějších problémů v podobě modulů, případně možnost rozdělit subsystémy na různé počítače pro rovnoměrné využití výpočetního výkonu. Moduly frameworků obvykle stačí jen správně uspořádat a propojit nebo mírně upravit, aby odpovídaly našim požadavkům. Znovu použití modulů šetří velké množství času, který je možné věnovat na nové a neprozkoumané oblasti nebo vypořádání se s obtížnějšími problémy.

Nakonec všechny subsystémy jsou poskládány do jednoho řídícího systému, který dohlíží na správnou funkčnost jednotlivých částí. Složení do jednoho systému poskytuje snazší jednotné ovládání a větší rozmanitost možností, než jednotlivé subsystémy sami o sobě. Výsledný řídící systém odstiňuje od práce s jednotlivými subsystémy a můžeme jej začlenit do složitějších systémů znova jako jeden podsystém.

2. Návrh řídícího systému

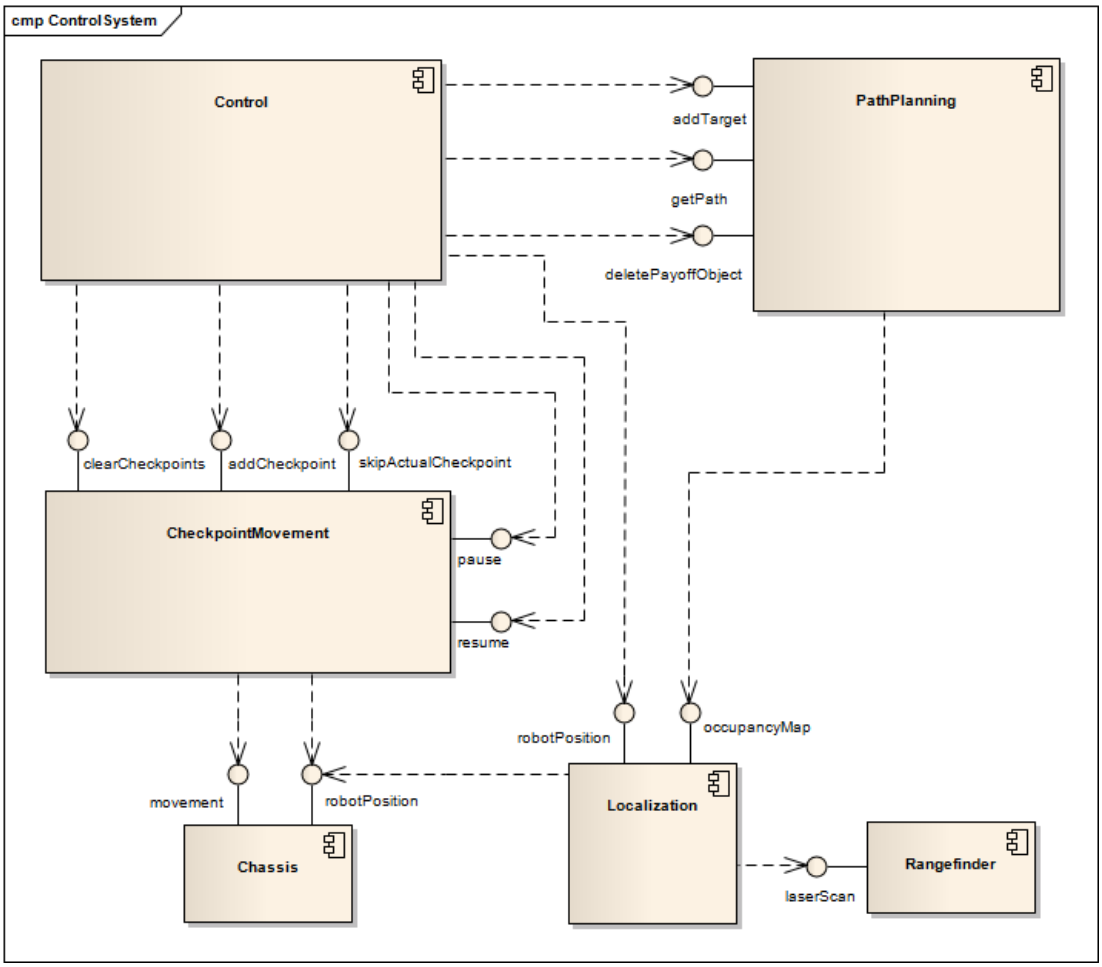
Obecný návrh řídícího systému a jeho jednotlivých podsystémů zavádí abstrakci vyšší úrovně. Abstrakci zprošťující uvažování od technických detailů a algoritmů. Můžeme proto celý řídící systém lépe rozdělit dle řešených dílčích problémů do odpovídajících podsystémů. Každý podsystém řeší potom oddělitelný problém a je ho možné umístit do samostatného znova použitelného modulu. V této kapitole popíšeme obecnou strukturu celého řídícího systému a jeho podsystémů. Použité algoritmy popisuje následující kapitola.

Řídící systém je koncipován s uzavřenou smyčkou pro dosažení co nejvyšší přesnosti pohybu robota. Hlavní schopností systému je samostatná jízda robota po zadaných kontrolních bodech, v neznámém prostředí. Proto si jeden z podsystému vytváří a udržuje mapu okolí robota, s neustálou aktualizací, umožňující dosahovat nízkého rozptylu chyby pozice robota. Systém současně zajišťuje vyhýbání se kolizím robota s překážkami. Jednotlivé podsystémy jsou tvořeny samostatnými funkčními moduly, které se dají znova použít i v jiných projektech.

2.1 Architektura řídícího systému

Struktura řídícího systému je popsána šesti podsystémy, jmenovitě řízení (Control), jízda po průjezdních bodech (CheckpointMovement), plánování cesty (PathPlanning), lokalizace (Localization), kontrola podvozku (Chassis) a laserový dálkoměr (Rangefinder). Architektura řídícího systému a propojení jednotlivých subsystémů je znázorněna na obrázku 2.1.

- **Control** zajišťuje nejvyšší logiku a ovládání jednotlivých subsystémů, spravuje cílové body, převádí souřadnice ze světových do relativních vůči Chassis, detekuje překážky, zabraňuje kolizím, přidává body do PathPlanningu skrze volání služby addTarget a provádí dotazy na PathPlanning getPath ohledně k nejbližších průjezdních bodů pro robota;
- **PathPlanning** vytváří plán pohybu robota na základě vložených cílových bodů a mapy prostředí. Zároveň tento subsystém odpovídá na dotazy ohledně vytvořeného plánu pohybu;
- **CheckpointMovement** se stará o dosažení vložených průjezdních bodů ve stejném pořadí jako byly vloženy. Průjezdní body jsou vkládány subsystémem Control voláním addCheckpoint;
- **Chassis** poskytuje relativní pozici podvozku vzhledem ke startovní pozici a zajišťuje pohyb po kružnici (definované poměrem dopředné a rotační rychlosti);
- **Localization** vytváří průběžně mapu prostředí robota, ve které udržuje aktuální pozici robota a dále ji poskytuje pro výpočty PathPlanningu. Mapa je tvořena na základě dat ze subsystému Rangefinder a Chassis;
- **RangeFinder** poskytuje data z laserového dálkoměru.



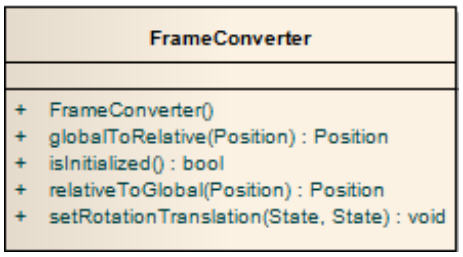
Obrázek 2.1: UML diagram znázorňující architekturu řídícího systému.

Pro snížení složitosti podsystémů jsou zavedeny dva souřadnicové systémy. Všechny subsystémy, kromě Control, používají výhradně jenom jeden. Neznalost druhého souřadného systému eliminuje nutnost převodu pozic mezi nimi a tak i složitost celého podsystému. První souřadnicový systém je definován počáteční pozicí robota a je používán subsystémy Control, Chassis a CheckpointMovement. Druhý souřadnicový systém definuje mapa vytvářená podsystémem Localization. Tento souřadnicový systém je tedy využit podsystémy Control, Localization a PathPlanning. Použití jednoho nebo druhého typu rozděluje architekturu na dvě vrstvy, se spojovacím prvkem v podobě Control.

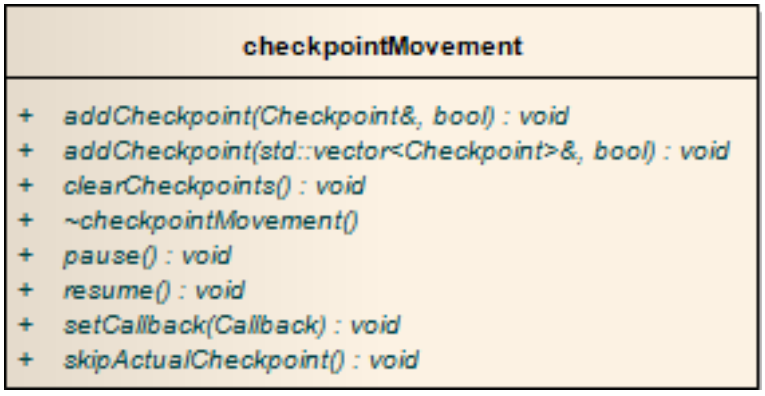
2.1.1 Control

Přijímá od uživatele množinu cílových bodů $T = \{t|t \in \mathcal{R}^2\}$, kde jednotlivé body $t = [x, y]$ jsou cílovými body robota v rovině. Control nezajišťuje průjezd cílovými body ve stejném pořadí jako byly zadány uživatelem, ale v nejvhodnějším vzhledem k PathPlanning. Cílové body jsou předány do PathPlanning voláním služby addTarget. Při dosažení cílového bodu robotem s určitou přesností ϵ je odstraněn z množiny T a zavolána služba deletePayoffObject na PathPlanning.

Control si průběžným dotazováním na PathPlanning službu getPath udržuje k nejbližších průjezdních bodů, směrem k nejvhodnějšímu cílovému bodu. Průjezdní



Obrázek 2.2: Třída FrameConvert.



Obrázek 2.3: Interface CheckpointMovement a struktura pro uchovávání průjezdních bodů (Checkpointů).

body v CheckpointMovement jsou v souřadnicovém systému vzhledem ke startu robota, na rozdíl od přijatých průjezdních bodů z getPath a cílových bodů, které jsou v souřadnicovém systému mapy. Převod mezi těmito souřadnicovými systémy provádí modul FrameConvert znázorněný na obrázku 2.2. Před předáním voláním služby addCheckpoint na CheckpointMovement jsou průjezdní body převedeny do relativního souřadnicového systému robota.

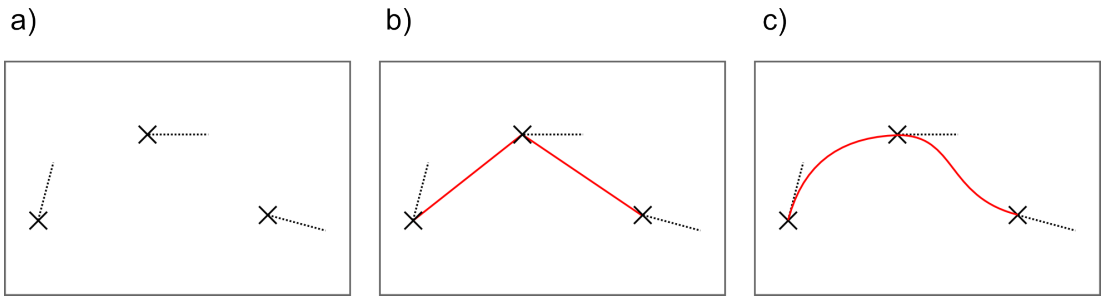
V případě nenadálých změn prostředí nebo špatně naplánované cestě z nedostatečné mapy, by mohlo dojít ke kolizi robota s překážkou. Tomu je zabráněno detekcí kolizí z laserového dálkoměru s následným přeplánováním trasy.

2.1.2 CheckpointMovement

Subsystém udržuje frontu vložených průjezdních bodů (Checkpoints) a umožňuje nad ní provádět základní operace, jako vložení bodu na začátek nebo konec, vymazání aktuálního průjezdního bodu nebo celé fronty. Ovládání subsystému dovoluje pozastavit a znova obnovit jízdu robota. Pokud je definována callback funkce je tato funkce volána se stavem fronty průjezdních bodů, vždy když robot dorazí na aktuální průjezdní bod. Celý interface CheckpointMovement je znázorněn na obrázku 2.3.

Na základě aktuálního průjezdního bodu a pozice robota z Chassis generuje CheckpointMovement ovládací příkazy, volané na interface Chassis tak, aby robot projel daným bodem s co nejvyšší přesností a správným směrem ho opustil. Možné trajektorie robota pro množinu průjezdních bodů jsou znázorněny na obrázku 2.4.

Průjezdní body $c = (p, \overrightarrow{o})$ jsou definovány jako uspořádaná dvojice pozice a výstupního vektoru, kde pozice $p = [x, y]; x, y \in \mathcal{R}$ určuje pozici v rovině, kterou



Obrázek 2.4: Trajektorie robota pro pevně danou množinu průjezdních bodů. Část a) obsahuje jednotlivé průjezdní body s výstupními vektory. Části b) a c) ukazují možné trajektorie robota těmito body.

robot má projet vzhledem k startu robota a výstupní vector $\overrightarrow{o} = [x, y]; x, y \in \mathcal{R}$ určuje směr, kterým bude robot průjezdní bod opouštět.

2.1.3 PathPlannig

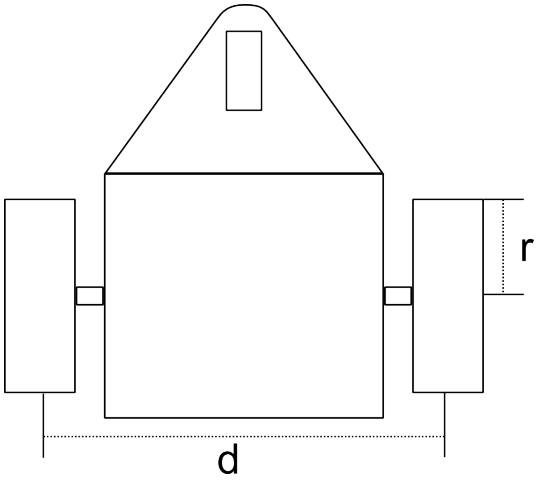
Na základě uživatelem definovaných cílových bodů a mapy podsystém PathPlanning vytváří cestu robota tak, aby se vyhýbal statickým překážkám a projel všechny zadané cílové body. Volání služby getPath s pozicí v rámci mapy vrací pozici, na kterou by robot měl jet z této pozice. Pokud je požadováno n navazujících pozic $(c_1, \ldots, c_n)$ z pozice s, musí se n-krát volat getPath postupně, s pozicí s, c_1 až c_{n-1}.

2.1.4 Localization

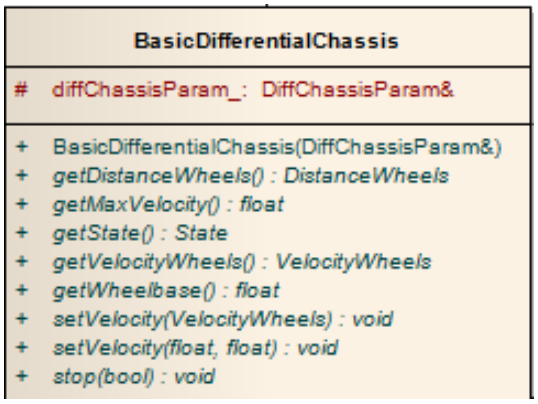
Subsystém Localization vytváří mapu prostředí robota, ve které současně provádí lokalizaci robota. Localization přijímá data z modulu Rangefinder 2.1.6, poskytujícího informace o vzdálenostech k překážkám a relativní pozici robota z modulu Chassis 2.1.5. Tyto údaje jsou použity pro stavbu mapy prostředí i samotnou lokalizaci robota v ní. Vypočítaná pozice robota z lokalizace se používá pro průběžný výpočet převodních matic mezi souřadnicovými systémy robota a mapy v subsystému Control.

2.1.5 Chassis

Chassis představuje vyšší vrstvu nad hw robota, jako je řízení výkonu kol a práce s kvadraturními enkodéry. Umožňuje ovládat podvozek robota pomocí nastavování rychlosti jednotlivých kol nebo poměrem mezi dopřednou a rotační rychlostí robota přes funkce setVelocity. Modul je vhodný pro všechny roboty s diferenciálním podvozkem, protože modulu se při inicializaci předávají základní parametry podvozku a nastavení. Logika ovládání je pro všechny diferenciální roboty stejná, mění se pouze parametry podvozku. Parametry podvozku, předávané jako struktura DiffChassisParam, jsou rozchod kol d, poloměr kola r, počet kroků enkodérů na jednu otáčku kola, maximální rychlost podvozku, parametry PID regulátoru pro pravé a levé kolo a třídy pro komunikaci s enkodéry a výkonovou částí motorů. Obecný diferenciální podvozek je znázorněn na obrázku 2.5.



Obrázek 2.5: Nákres obecného diferenciálního podvozku se zvýrazněným rozchodem kol d a poloměrem kola r.



Obrázek 2.6: Interface Chassis pro základní ovládání podvozku robota.

Chassis z ujeté vzdálenosti na jednotlivých kolech počítá relativní pozici podvozku v rovině vzhledem ke startu robota, kterou vrací volání funkce getState jako uspořádanou trojici (x, y, θ) kde $x, y \in \mathcal{R}$ je pozice a θ natočení robota. Další moduly mohou získat základní nastavení podvozku pomocí funkcí getMaxVelocity a getWheelBase vracející maximální rychlost a vzdálenost mezi středy kol. Interface Chassis je zobrazen na obrázku 2.6.

2.1.6 Rangefinder

Rangefinder se stará o komunikaci a získávání dat z laserových dálkoměrů, které poskytují vzdálenostní informace v určitém rozsahu, například 270 stupňů s centimetrovou přesností a několikametrovým dosahem dle typu. Informace o vzdálenostech k překážkám jsou obohaceny o časovou značku a základní informace o typu senzoru, jako je maximální dosah a počet vzorků v jednom měření. Tato měření jsou používána modulem Localization pro tvorbu mapy a modulem Control pro zabránění kolizí robota s překážkami.

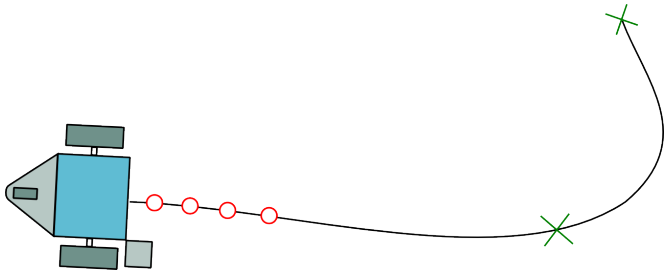
3. Použité algoritmy

3.1 Control

Control propojuje všechny moduly do jednoho funkčního řídícího systému, který řídí robota tak, aby postupně navštívil všechny dostupné uživatelem definované cílové body. Dosažení jednotlivých cílových bodů probíhá přes množinu k navazujících bodů $\{b_1, b_2, \ldots, b_k\}$, získaných z modulu PathPlanning. První bod v množině je vygenerován na základě aktuální pozice, druhý na základě prvního atd. Body množiny představují část cesty mezi aktuální pozicí a cílovými body, jak je znázorněno na obrázku 3.1. Když robot projede prvním bodem z množiny, je tento bod z množiny odstraněn a Control si vyžádá navazující bod na poslední bod z množiny. Získaný bod z PathPlanning je převeden do souřadnicového systému podvozku, zařazen na konec množiny a přidán CheckpointMovement na konec fronty.

Udržováním nejbližších k bodů cesty je umožněna průběžná úprava cesty při změně nebo stavbě mapy prostředí. Cesta z bodu b_k se předá CheckpointMovement, až robot projede bodem b_1, takže cesta od bodu b_k se může dynamicky měnit v závislosti na změnách prostředí, do doby, než robot projede bodem b_1. Jestli by došlo k situaci, že nějaký bod cesty byl už naplánován a předán do CheckpointMovement a tento bod leží v překážce nebo kolizní vzdálenosti od překážky. Tak se daná skutečnost projeví pomocí reaktivní detekce překážek. V takovém případě budou všechny body ve frontě CheckpointMovement vymazány a znova vygenerovány podle změněné mapy.

Velikost parametru k určuje délku zafixované části cesty, předané do CheckpointMovement. Předanou část cesty dále neupravujeme kromě případu, kdy by robot měl narazit do překážky. Naším cílem je minimalizovat délku předané části cesty a tedy i minimalizovat velikost parametru k. Čím bude předaná část cesty kratší, tím líp bude celý systém reagovat na změny prostředí. Pokud zvolíme parametr k moc malý, například jedna, projeví se to zastavováním robota, vždy když dorazí na poslední předaný bod. Zastavování robota způsobuje časová prodleva během komunikace mezi Control a PathPlanning. Abychom dosáhli plynulé jízdy bez zastavování, musíme hodnotu parametru k nastavit dostatečně velkou. Proto se snažíme minimalizovat hodnotu parametru k, dokud se robot nezačne zastavovat v průběhu jízdy na cílové body.



Obrázek 3.1: Černá křivka přestavuje naplánovanou cestu modulem PathPlanning, procházející cílovými body (zelené křížky). Červená kolečka reprezentují jednotlivé body z k-prvkové množiny. V tomto případě je k rovno čtyři.

```
1  Algoritmus Control:
2
3  while true do
4    if prekazka then
5      smaz vse z CheckpointMovement
6      popojed dozadu
7    endif
8
9    for i zbyvajicich bodu to k do
10     bod_abs := PathPlanning ziskej bod z bod_{i-1}
11     bod_rel := preved_souradnice(bod_abs)
12     CheckpointMovement pridej checkpoint bod_rel
13   endfor
14 endwhile
```

3.1.1 Reaktivní detekce překážek

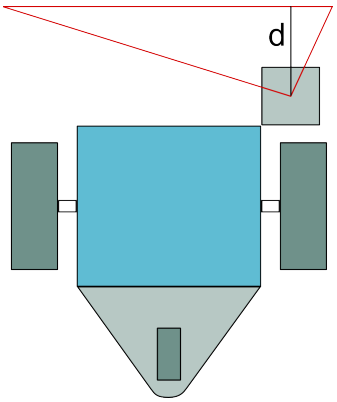
Detekce překážek využívá informací z laserového dálkoměru, umístěného na robotovi. Detekce vytváří zónu virtuálního nárazníku ve vzdálenosti d od senzoru znázorněnou na obrázku 3.2. Na rozdíl od přímočarého řešení, pomocí kruhové výseče, je tento způsob detekce vhodnější pro zvolený typ robota. Kruhová výseč se středem, odpovídajícímu středu senzoru, definuje nevhodnou detekční zónu, kdy překážky přímo před senzorem jsou brány jako kolizní předčasně, pokud zvolíme poloměr výseče tak, aby překážky na druhé straně robota byly registrovány jako kolizní ve správné vzdálenosti.

Virtuální zóna je definována úhly min a max, určující maximální a minimální úhel použitých měření vzhledem k senzoru. Libovolná překážka, která se dostane do této zóny, je brána jako kolizní. Při zjištění možnosti kolize se robot okamžitě zastaví a přeplánuje svojí dosavadní cestu.

```
1  Algoritmus detekce_prekazek(z_t ,d):
2
3  for k:= 1 to K do
4    if z_t^[k] < d / cos(uhel z_t^[k]) a uhel z_t^[k] ∈ [min, max] then
5        return 1
6    endif
7  endfor
8
9  return 0
```

3.1.2 Převod souřadnic

Řídící systém obsahuje dva souřadnicové systémy, jeden je definovaný mapou prostředí E_m a druhý relativně ke startu robota E_r. Abychom byli schopní vygenerované body z PathPlanningu, definované v E_m, použít jako průjezdní body pro podvozek v modulu CheckpointMovement, který operuje v E_r musíme mezi



Obrázek 3.2: Červený trojúhelník před robotem ohraničuje zónu virtuálního nárazníku.

těmito systémy být schopní body převádět [1]. Možná vzájemná poloha souřadnicových systémů je znázorněná na obrázku 3.3.

Převod mezi těmito souřadnicovými systémy můžeme popsat v homogenních souřadnicích translačními a rotačními maticemi [40]. Pro převod bodu z E_m do E_r (budeme značit T_{rm}) platí, že musíme prvně provést posun bodu a poté rotaci.

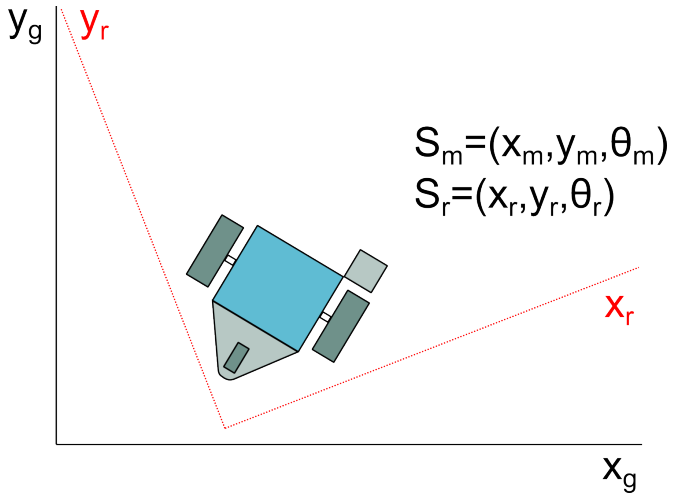
Rotační matice a matice posunu v homogenních souřadnicích jsou definovány následně pro úhel otočení ψ a posun (p_x, p_y):

$$T_{rot} = \begin{pmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{pmatrix} T_p = \begin{pmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.1}$$

Pokud známe stav robota v čase t pro oba systémy $S_{m,t}$ a $S_{r,t}$, vypočítáme matici T_{rm} jako násobek rotační matice rozdílu úhlů $S_{r,t}$ a $S_{m,t}$ a matice posunu vzniklé rozdílem převedené relativní pozice a pozice robota v mapových souřadnicích:

$$\begin{aligned} T_{rot} &= \begin{pmatrix} cos(\theta_{r,t} - \theta_{m,t}) & -sin(\theta_{r,t} - \theta_{m,t}) & 0 \\ sin(\theta_{r,t} - \theta_{m,t}) & cos(\theta_{r,t} - \theta_{m,t}) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ T_{pos} &= T_{rot}^{-1} \begin{pmatrix} x_{r,t} \\ y_{r,t} \\ 1 \end{pmatrix} - \begin{pmatrix} x_{m,t} \\ y_{m,t} \\ 1 \end{pmatrix} \\ T_{rm} &= T_{rot} \begin{pmatrix} 1 & 0 & \\ 0 & 1 & T_{pos} \\ 0 & 0 & \end{pmatrix} \end{aligned} \tag{3.2}$$

[1] Důvod je vysvětlený v kapitole 2.1.



Obrázek 3.3: Dva souřadnicové systémy v rámci řídícího systému. Černý je definovaný mapou prostředí a červený pozicí startu robota. S_m a S_r je aktuální stav robota vzhledem ke středu otáčení v jednotlivých souřadných systémech.

Pro matici převodu relativních souřadnic do mapových T_{mr} prohodíme relativní stav za stav robota v mapě a obráceně:

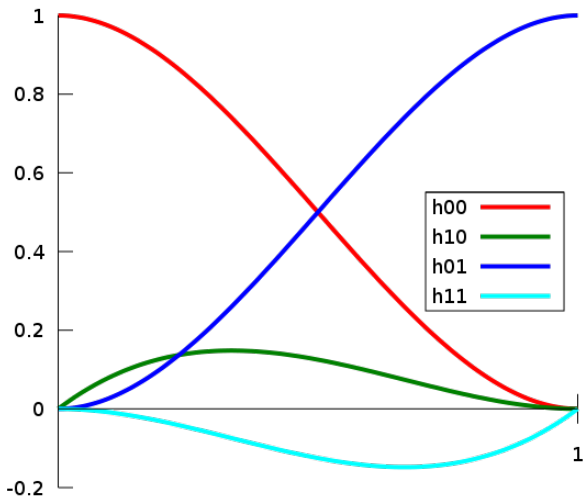
$$T_{rot} = \begin{pmatrix} cos(\theta_{m,t} - \theta_{r,t}) & -sin(\theta_{m,t} - \theta_{r,t}) & 0 \\ sin(\theta_{m,t} - \theta_{r,t}) & cos(\theta_{m,t} - \theta_{r,t}) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$T_p = T_{rot}^{-1} \begin{pmatrix} x_{m,t} \\ y_{m,t} \\ 1 \end{pmatrix} - \begin{pmatrix} x_{r,t} \\ y_{r,t} \\ 1 \end{pmatrix}$$
$$T_{mr} = T_{rot} \begin{pmatrix} 1 & 0 \\ 0 & 1 & T_{pos} \\ 0 & 0 \end{pmatrix} \tag{3.3}$$

Převod mezi pozicemi stavů S_m a S_r je následující:

$$S_r^T = T_{rm} S_m^T \quad tj. \quad \begin{pmatrix} x_r \\ y_r \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\psi) & -sin(\psi) & p_x \\ sin(\psi) & cos(\psi) & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_m \\ y_m \\ 1 \end{pmatrix} \tag{3.4}$$

3.2 CheckpointMovement

Pro dosažení plynulého průjezdu podvozku robota definovanými průjezdními body se generují hermitovské křivky [40] mezi dvojicemi po sobě jdoucími body, jenž jsou kopírovány pohyby podvozku robota. Hermitovské křivky jsou definovány jednotlivými průjezdními body tak, že pozice průjezdního bodu p je bodem hermitovské křivky a výstupní vektor $\overrightarrow{o}$ průjezdního bodu je i zároveň výstupním vektorem bodu na hermitovské křivce. Hermitovské křivky byli zvoleny pro výpočetně nenáročnou implementaci s přímočarým převodem mezi průjezdními body a body hermitovské křivky a svojí vlastností hladce interpolovat mezi jednotlivými body, pokud konečný bod minulé křivky je zároveň začátečním bodem



Obrázek 3.4: Čtyři základní funkce ze kterých se skládá hermitovská křivka.

následující křivky [2]. Hermitovské křivky při správném pořadí průjezdních bodů a nastavení výstupních vektorů snižují nutnost zastavovat robota. Robot není nucen vykonávat prudké změny směru při sledování výsledné křivky a může proto definovanými průjezdními body projet za kratší čas vyšší rychlostí, než při přístupu, kdy by se vždy natočil na místě směrem k dalšímu průjezdními bodu a následně na něj jel přímo.

3.2.1 Hermitovská křivka

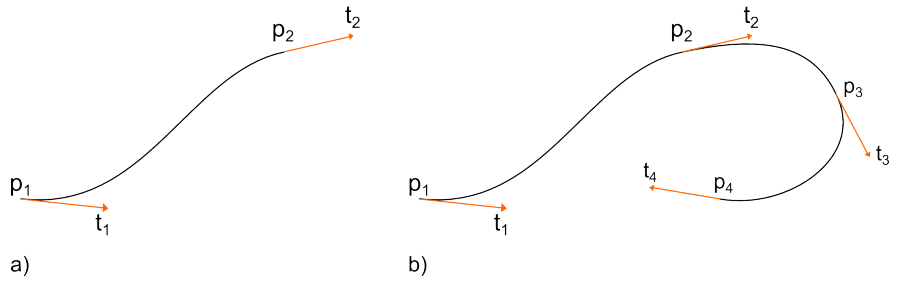
Hermitovská křivka je kubická polynomiální funkce definována dvojicí bodů p_1 a $p_2 \in \mathcal{R}^2$ v rovině a jejich výstupními vektory $\overrightarrow{t_1}$ a $\overrightarrow{t_2} \in \mathcal{R}^2$. Pro množinu bodů $(p_1, p_2, \ldots, p_n)$ a jejich výstupních vektorů $(\overrightarrow{t_1}, \overrightarrow{t_2}, \ldots, \overrightarrow{t_n})$ se hermitovská formule 3.5 aplikuje postupně na jednotlivé intervaly $((p_1, \overrightarrow{t_1}, p_2, \overrightarrow{t_2}), (p_2, \overrightarrow{t_2}, p_3, \overrightarrow{t_3}), \ldots, (p_{n-1}, \overrightarrow{t_{n-1}}, p_n, \overrightarrow{t_n}))$. Výsledná hermitovská křivka je spojitá a má i spojitou první derivaci.

Hermitovská formule je složená ze čtyř základních kubických funkcí zobrazených na obrázku 3.4. pro jednotlivé vstupní parametry a je definována na jednotkovém intervalu (0,1). Výpočet bodu hermitovské křivky definované body p_1 a p_2, jejich výstupními vektory $\overrightarrow{t_1}$ a $\overrightarrow{t_2}$ a hodnotě $s \in (0, 1)$ v maticovém zápisu má následující formu:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} s^3 & s^2 & s^1 & 1 \end{pmatrix} \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ \overrightarrow{t_1} \\ \overrightarrow{t_2} \end{pmatrix} \tag{3.5}$$

Na obrázku 3.5 jsou vykresleny dvě hermitovské křivky. Křivka a) je definovaná body p_1 a p_2 a jejich výstupními vektory $\overrightarrow{t_1}$ a $\overrightarrow{t_2}$. Křivka b) začíná stejnými body jako první, které doplňují body p_3 a p_4 a výstupní vektory $\overrightarrow{t_3}$ a $\overrightarrow{t_4}$.

[2] V takovém případě je výsledná cesta spojitá přes všechny průjezdní body a robot jí zvládne projet.



Obrázek 3.5: Vykreslené dvě hermitovské křivky. Křivka a) je definovaná body p_1 a p_2 a jejich výstupními vektory $\overrightarrow{t_1}$ a $\overrightarrow{t_2}$. Křivka b) začíná stejnými body jako předešlá, které doplňují body p_3 a p_4 a výstupní vektory $\overrightarrow{t_3}$ a $\overrightarrow{t_4}$.

3.2.2 Algoritmus sledování hermitovské křivky

V předchozí podkapitole bylo shrnuto, jak vygenerovat jednotlivé body hermitovské křivky, ale robot musí být schopný tyto body projet plynule s co nejvyšší přesností a pokud možno rychle. Proto byl v rámci této práce vyvinut následující algoritmus pro sledování hermitovské křivky:

```
1   Algoritmus pro sledovani
2   hermitovske krivky(p1,p2,t1,t2):
3
4   pr := aktualni pozice robota
5   s := 0
6   pn := bod_hermit(p1,p2,t1,t2,s)
7   smer := rovne
8   while vzdalenost p2 a pr > epsilon do
9     while vzdalenost pn a pr < predikcni_vzdalenost
10      a vzdalenost pn a p2 > predikcni_vzdalenost do
11      pn := bod_hermit(p1,p2,t1,t2,s)
12      s += krok
13    endwhile
14    if vzdalenost pn a p2 < predikcni_vzdalenost
15      pn := p2
16    uhel := uhel mezi pn a pr
17    vzdalenost := vzdalenost pn a pr
18    smer := zjisti_smer(uhel)
19    switch(smer)
20      rovne:
21        setVelocity(vzdalenost,uhel)
22      rotace:
23        setVelocity(0,uhel)
24    pr := aktualni pozice robota
25  endwhile
```

Algoritmus přijímá jako své parametry body p_1 a p_2 a jejich výstupní vektory $\overrightarrow{t_1}$ a $\overrightarrow{t_2}$. Na začátku cyklu testuje, zda-li už není aktuální pozice robota v epsilon okolí cílového bodu p_2, pokud ano, tak jsem dosáhli koncového bodu a stejný algoritmus použijeme pro další dvojici průjezdních bodů. V případě,

že vzdálenost je větší, tak si postupně vygenerujeme bod na hermitovské křivce (řádek 9-13), který musí být ve větší vzdálenosti než predikční od aktuální pozice robota a zároveň koncový bod musí být ve vzdálenosti větší než predikční. Následně, pokud je koncový bod blíž než predikční vzdálenost, tak nový bod, na který robot směřuje je koncový. Na řádcích 16-18 se zjišťuje vzájemná poloha, pomocí zjisti_smer, nového interpolovaného bodu a aktuální pozice robota na základě, které se robot bude otáčet na místě, pokud úhel mezi pozicemi je větší než 90 stupňů nebo v opačném případě pojede po kružnici, definované poměrem vzdálenosti a úhlu. Otáčení na místě je zavedeno vzhledem k technickým parametrům použité robotické platformy (popsané v kapitole 5), které se mění střed otáčení, pokud má jet po kružnici menší než polovina rozchodu kol. Změna středu otáčení má za následek zvětšení chyby výpočtu relativní pozice robota.

3.3 PathPlannig

Výběr algoritmu pro plánování cesty robota je závislý na typu cílových konfigurací, robota a mapy. Pro testování a implementaci jsme vybrali prostředí, které je menší řádově jednotky až desítky metrů a má dostatečný volný prostor pro pohyb robota, jako například běžný pokoj nebo čtvercové hřiště 2x2 metry. Zároveň algoritmus FastSlam 3.4.1 s mapou obsazenosti prostoru použitý v lokalizačním modulu nám určuje vstupní typ mapy prostředí.

Dále si zafixujeme typ podvozku robota diferenciální, případně všesměrový. U diferenciálního a všesměrového podvozku nejsme nijak omezeni předchozím směrem pohybu. Pouze u diferenciálního v případě, že následující směr je kolmý na současný, se musí podvozek nejdříve otočit na místě vhodným směrem a následně jet. Výhodou diferenciálního, oproti ackermanovu je, že nepotřebují k otočení na místě víc místa než vzdálenost mezi středem otáčení a nejvzdálenějším bodem robota. Proto uvažujeme pouze diferenciální a všesměrový podvozek.

Nakonec pro zjednodušení budeme používat pouze dosažení určitého bodu v prostředí, určeného uživatelem s libovolnou orientací, jako cílovou konfiguraci robota.

Předešlá rozhodnutí a z nich plynoucí omezení nám zužují skupinu algoritmů, použitelných pro vytvoření plánu cesty. Budeme uvažovat následující algoritmy jako je A*, iterace hodnot, pravděpodobnostní silniční mapy (dále jen PRM) a částečně pozorovatelné markovské rozhodovací procesy (dále jen POMDP) [41].

Budeme používat reálný hw v reálném prostředí, kde není zajištěno stoprocentní vykonávání akcí. Například, místo přesné jízdy rovně, mohou v průběhu pohybu proklouznout jednotlivá kola robota a výsledná trajektorie se změní. Potřebujeme proto, aby plánovací algoritmus byl pravděpodobnostní a zahrnoval v průběhu vytváření plánu možnost nepřesného vykonávání pohybů, což A* nesplňuje. Dále výpočetní výkon řídící jednotky robota není veliký, požadujeme online vytváření plánů s průběžným přepočítáváním podle změn v prostředí a použitý algoritmus by měl být i ilustrativní a snadno pochopitelný pro výukové účely.

PRM algoritmy mají dvě fáze, kde v první se generuje hustý graf použitý v druhé fázi pro hledání nejkratší cesty Dijkstrovým algoritmem. Jelikož robot staví mapu průběžně, dochází ke změnám často a vyhledávací graf by se musel často dostavovat nebo stavět od začátku, podle typu změn v mapě. Časté změny grafu a následné přepočítávání cesty je náročné na výpočetní výkon a tato

technika hledání cesty se používá častěji s pevnou mapou.

Zobecněním iterace hodnoty o nejistotu v měření, dostaneme POMDP. POMDP počítá s nejistotou v měření i pohybu robota. Očekává, že stav robota v prostředí není plně pozorovatelný. My, ale předpokládáme, že stav robota je v každé chvíli plně pozorovatelný.

Proto jako plánovací algoritmus byla zvolena iterace hodnot, která je snadno pochopitelná a tudíž ilustrativní pro výukové účely, při plánování pohybu se počítá s nepřesností vykonávání, známe v každé chvíli stav robota a pohybujeme se v malém prostředí, s omezenou množinou typů pohybů.

3.3.1 Iterace hodnot

Iterace hodnot rekurzivně vypočítává prospěch z jednotlivých akcí relativně k užitkové funkci, předpokládá, že stav prostředí je plně pozorovatelný v každé chvíli. Akce robota (pohyby) jsou určovány požadovanými cíli, v našem případě dosažením určené pozice v prostředí a současně optimalizováním ostatních proměnných, jako je ujetá vzdálenost pomocí ceny. Pro sjednocení kladných a záporných cen se používá užitková funkce. Užitková funkce, značená r, je funkce stavu s a akce u robota. Jednoduchá užitková funkce může vypadat následovně:

$$r(s,u) = \begin{cases} +50 & : \textit{jestliže } u \textit{ vede do cílové pozice} \\ -1 & : \textit{jinak} \end{cases}$$

Tato užitková funkce odmění robota $+50$, jestliže je cílová pozice dosažena, zatím co penalizuje robota -1 za každý krok, kdy není. Takováto užitková funkce bude mít maximální souhrnný výnos, když robot dorazí na cílovou pozici v nejmenším možném čase.

Zajímá nás, jak generovat akce tak, aby optimalizovaly očekávané užitky. Takové generování se nazývá řídící politika a je značené v případě, že máme plně pozorovatelný stav:

$$\pi : x_t \rightarrow u_t \tag{3.6}$$

V kontextu tvorby řídící politiky je důležitý plánovací horizont, aneb jak nás do budoucna ovlivňují současné akce. Pokud se robot pohybuje na cílovou pozici, získá finální užitkovou cenu až při dosažení cíle poslední akcí, a proto může být užitek pozdržen. Vhodná řídící politika by proto měla vybírat akce tak, aby suma budoucích užitků byla maximální. Tato suma se nazývá souhrnný užitek a protože je svět nedeterministický, můžeme pouze optimalizovat očekávaný souhrnný užitek. Kdyby byl svět deterministický, tak lze očekávaný souhrnný užitek vypočítat přesně. Očekávaný souhrnný užitek je běžně značený jako

$$R_T = E\left[\sum_{\tau=1}^{T} \gamma^{\tau} r_{t+\tau}\right] \tag{3.7}$$

Zde je očekávání $E[]$ bráno přes budoucí okamžité užitky $r_{t+\tau}$, které může robot dosáhnout mezi časy t a $t+T$. Jednotlivé užitky $r_{t+\tau}$ jsou násobeny faktorem γ^{τ}, nazývaný slevový faktor. Hodnota γ je specifická pro různé problémy a leží v intervalu $(0;1]$. Menší hodnoty γ než jedna exponenciálně slevují budoucí užitky. Dřívější užitky tvoří exponenciálně důležitější než pozdější.

R_T je suma přes T časových kroků. T se nazývá plánovací horizont. U plánovacího horizontu rozpoznáváme tři důležité případy:

1. $T = 1$. Nazývaný nenasytný případ ("greedy"). Robot se snaží minimalizovat užitek jen z následující akce. Akce dále, než následující časový krok, nehrají žádnou roli, ale i tak má praktické využití. Existují robotické problémy, kde nenasytný případ je v současnosti nejlepší známé řešení, spočítatelné v polynomiálním čase (například problémy převoditelné na problém obchodního cestujícího);

2. T větší než 1, ale konečné. Běžně nazývaný jako případ s konečným horizontem. Typický užitek není slevován v čase, $\gamma = 1$. Praktické úlohy mají většinou konečný časový horizont, ale optimality v konečném horizontu je často těžší dosáhnout, než ve slevovaném nekonečném horizontu (viz dále). Blízko konce časového horizontu se může optimální politika podstatně lišit od optimální akce, vybrané dříve, dokonce při stejných podmínkách. Výsledkem je, že plánovací algoritmus s konečným horizontem musí udržovat různé plány pro různé horizonty. Více plánu přidává nežádanou složitost;

3. T nekonečné. Běžně nazývaný jako případ s nekonečným horizontem. Tento případ netrpí stejným problémem jako případ s konečným horizontem, jelikož počet zbývajících časových kroků je pro všechny body v čase stejný (nekonečný). Naopak zde je důležitý slevový faktor γ, který rozlišuje, mezi případy, kdy v každém kroku je užitek kladný, ale rozdílné velikosti. Představme si dva řídící programy, jeden v každém kroku získá užitek $+1$ a druhý $+5$ a $\gamma = 1$. Při libovolném konečném horizontu je druhý program jasně lepší, ale pokud máme nekonečný horizont, oba nám vydělají nekonečné množství peněz. Očekávaný souhrnný užitek R_T je v obou případech ∞ a tedy nedostatečný pro vybrání lepšího programu.

   Za předpokladu, že každý individuální užitek r je shora ohraničený nějakým číslem r_{max}, zajišťuje slevování, že R_∞ je konečné navzdory tomu, že suma má nekonečně členů. Dokonce dostaneme

$$R_\infty \leq r_{max} + \gamma r_{max} + \gamma^2 r_{max} + \ldots = \frac{r_{max}}{1-\gamma} \tag{3.8}$$

   Předešlá rovnice dokazuje, že R_∞ je konečné právě když $\gamma < 1$.

Nalezení optimální řídící politiky Začneme s definováním optimální politiky pro plánovací horizont $T = 1$. Zajímá nás politika, která maximalizuje příští okamžitý užitek. Takovou politiku si označíme $\pi_1(x)$ a získáme jí maximalizací očekávaného jednokrokového užitku, přes všechny akce:

$$\pi_1(x) = \operatorname*{argmax}_u r(x, u) \tag{3.9}$$

Odsud, optimální akce je taková, jenž maximalizuje příští očekávaný okamžitý užitek. Každá politika má přidruženou funkci hodnoty, která měří očekávanou hodnotu (souhrnný zlevněný budoucí užitek) této specifické politiky. Pro π_1 je funkce hodnoty jen očekávaný okamžitý užitek zlevněný slevovým faktorem γ:

$$V_1(x) = \gamma \max_u r(x, u) \tag{3.10}$$

Tato hodnota je pro delší plánovací horizonty T definována rekurzivně, stejně tak i optimální politika. Z čehož plyne následující:

$$\pi_T(x) = \underset{u}{\operatorname{argmax}} \left[r(x,u) + \int V_{T-1}(x') p(x'|u,x) dx' \right] \tag{3.11}$$

$$V_T(x) = \gamma \max_u \left[r(x,u) + \int V_{T-1}(x') p(x'|u,x) dx' \right] \tag{3.12}$$

V případě nekonečného plánovacího horizontu má optimální funkce hodnoty sklony dosáhnout rovnovážného stavu (existují deterministické systémy, kde tomu tak není).

$$V_\infty(x) = \gamma \max_u \left[r(x,u) + \int V_\infty(x') p(x'|u,x) dx' \right] \tag{3.13}$$

Výpočet funkce hodnoty Označíme si aproximaci funkce hodnoty jako $\hat{V}$. Na začátku je nastavena na r_{min}, minimální možný okamžitý užitek.

$$\hat{V} \leftarrow r_{min} \tag{3.14}$$

Iterace hodnoty potom postupně upraví aproximaci pomocí následujícího rekurzivního pravidla, který počítá funkci hodnoty pro zvyšující se horizonty:

$$\hat{V}(x) = \gamma \max_u \left[r(x,u) + \int \hat{V}(x') p(x'|u,x) dx' \right] \tag{3.15}$$

Pravidlo iterace hodnoty je blízce podobné výpočtu optimální politiky pro horizont T výše. Iterace hodnoty konverguje pokud $\gamma < 1$ a ve speciálních případech i pro $\gamma = 1$. Pořadí, v jakém jsou stavy upravovány, nehraje roli, dokud platí, že každý stav je upravován nekonečně krát často. V každém bodě času funkce hodnoty $\hat{V}(x)$ definuje politiku:

$$\pi(x) = \underset{u}{\operatorname{argmax}} \left[r(x,u) + \int \hat{V}(x') p(x'|x,u) \right] \tag{3.16}$$

Pro konečný stavový prostor se integrál, ve všech předešlých rovnicích, může implementovat jako konečná suma přes všechny stavy. Často je tato suma výpočetně efektivní, protože $p(x'|x,u)$ je nenulová jen pro malý počet stavů x a x'.

Z předešlých rovnic vychází následující dva algoritmy:

```
1  Algoritmus pro iteraci_hodnoty():
2  for i := 1 to N do
3     V̂(x_i) := r_min
4  endfor
5
6  while nekonverguje
7     for i := 1 to N do
8        V̂(x_i) := γ max_u [ r(x_i,u) + Σ_{j:=1}^{N} V̂(x_j) p(x_j|x_i,u) ]
9     endfor
10 endwhile
11 return V̂
```

```
1 Algoritmus optimalni_politiky(x, V̂):
2 return argmax_u [r(x,u) + Σ_{j=1}^{N} V̂(x_j) p(x_j|x,u)]
```

První algoritmus pro výpočet iterace hodnoty pro diskrétní konečný stavový prostor, inicializuje funkci hodnoty na řádku 3. Na řádcích 6 až 10 je implementován rekurzivní výpočet funkce hodnoty. Až iterace hodnoty dokonverguje, tak funkce hodnoty $\hat{V}$ vytváří optimální politiku.

Druhý algoritmus optimální politiky pro daný stav x a optimální funkci hodnoty $\hat{V}$ vrací akci u, která maximalizuje očekávanou hodnotu.

3.4 Localization

Modul vytváří mapu prostředí, ve kterém se robot pohybuje a zároveň provádí lokalizaci, dle specifikace modulu 2.1.4. V robotice je tento problém známý jako simultánní lokalizace a mapování (dále jen SLAM). To znamená, že se současně vytváří mapa prostředí a provádí se lokalizace robota vzhledem k této mapě. Tento problém je velmi těžký, více než lokalizace robota v pevně dané mapě s neznámou počáteční pozicí, kdy se robot snaží svojí pozici vzhledem k mapě nalézt během cesty.

Řešení SLAM problému lze rozdělit do dvou skupin. Obě dvě jsou stejně důležité. Jedna skupina je známá jako online SLAM, protože odhaduje stav proměnných aktuálních v čase t. Většina algoritmů pro online SLAM je inkrementálních: zahazují minulá měření a informace o řízení potom, co jsou zpracovány. Druhá skupina je plný SLAM. Kde se snažíme spočítat odhady stavů přes celou cestu společně s mapou.

Algoritmů vhodných pro řešení SLAM problému je velké množství. Nejznámější jsou:

1. **EKF SLAM** [41] je nejstarší technika založená na rozšířeném kalmanově filtru (dále jen EKF). Používá EKF pro řešení online SLAM problému pomocí maximální pravděpodobnosti asociace dat. EKF SLAM je založen na velkém množství aproximací a omezujících předpokladů:

   Mapa prostředí, která shromažďuje informace jen o objektech s význačnými vlastnostmi, je takzvaně *feature-based* mapa. Z výpočetních důvodů musí obsahovat pouze informace o malém počtu objektů (řádově stovky).

   Jako všechny ostatní algoritmy, založené na EKF, předpokládá normální rozdělení šumu pro pohyb a vnímání robota.

   Nedokáže využít negativní informace, jako že objekt chybí v daném měření, ač podle mapy by tam měl být;

2. **GraphSLAM** [41] počítá řešení pro offline případ, definovaný přes všechny pozice a měření. Pozadí problému úplného SLAM přirozeně tvoří řídký graf. Tento graf vede k sumě nelineárních kvadratických omezení. Optimalizace omezeních vydá nejpravděpodobnější mapu a odpovídající množinu pozic robota. Hrany grafu odpovídají jednotlivým nelineárním omezením a vrcholy grafu jsou pozice robota a měření. Suma všech omezení ústí do nelineárního problému nejmenších čtverců;

Stavba grafu je následována samostatnou výpočetní fází, ve které se tato informace přemění na odhad stavu. GraphSLAM může získat o mnoho větší mapu, než zvládne EKF SLAM, ale výpočet je pozdržen. Do grafu získává informace bez jejich řešení.

Nejlépe se hodí pro řešení problémů, kdy máme pevně danou datovou množinu a chceme získat mapu. Výsledné mapy jsou přesnější než mapy, které generuje EKF SLAM.

3. **SEIF SLAM** (řídký rozšířený informační filtr) [41], stejně jako EKF SLAM, integruje poslední pozice robota a udržuje si pouze pravděpodobnost přes současnou pozici robota a mapu. Zároveň, stejně jako GraphSLAM, si SEIF udržuje informační reprezentaci všech znalostí. Na základě toho, se aktualizace SEIF stane pomalou operací informačního posunu, která je kvalitnější, vzhledem k pro aktivní aktualizaci EKF SLAM. Odsud se na SEIF můžeme podívat jako na kombinaci nejlepších vlastností z obou předchozích případů. SEIF se počítá online a je výpočetně efektivní;

4. **FastSLAM** [41] je částicový přístup k SLAM problému. Využívá Rao-Blackwellized filtr, jenž popisuje pravděpodobnost nějaké proměnné pomocí částic, společně s normálním rozdělením nebo jinými pravděpodobnostními technikami pro reprezentaci ostatních proměnných.

   FastSLAM používá částice pro odhad cesty robota. Částice reprezentuje stav robota v prostředí a zároveň si udržuje vlastní mapu ve zvolené reprezentaci. Mapa může být stejně jako u EKF, SEIF nebo GraphSlamu *feature-based* nebo v poslední době častěji využívaná mapa obsazenosti prostoru (occupancy map), reprezentována jako více rozměrné pole informací o obsazenosti prostoru.

   Výpočetní náročnost FastSLAM algoritmu, oproti algoritmům založených nad základním EKF SLAM, je nižší. Hlavní výhodou FastSLAM algoritmu je schopnost interpretovat data zvlášť pro jednotlivé částice. Proto si FastSLAM udržuje více různě pravděpodobných odhadů stavu, nikoliv jen nejpravděpodobnější odhad, na rozdíl od EKF SLAM.

   Schopnost sledovat více možných asociací dat současně, utváří FastSLAM robustnější, vzhledem k problémům asociací dat, než algoritmy založené na inkrementální nejvíce pravděpodobný asociaci dat.

   Další výhoda nad ostatními SLAM přístupy, pramení ze schopnosti vypořádat se s nelineárním modelem pohybu robota. Kde předešlé techniky pomocí aproximačních metod linearizují model.

   Použití částicového filtru vytváří neobvyklou situaci, že FastSLAM řeší oba dva typy SLAM problému, jak plný, tak online. FastSLAM je formulován tak, že počítá pravděpodobnost celé cesty, řeší tedy plný SLAM. Nicméně výpočet částicových filtrů probíhá přes jednotlivé stavy v čase a FastSLAM je tedy i online algoritmus.

Pro výslednou implementaci byl zvolen FastSLAM algoritmus pro svojí robustnost, vzhledem k ostatním metodám. Možnost řešit, jak plný, tak online SLAM kde nás bude zajímat řešení online SLAM problému a schopnost udržovat více odhadů stavu robota než jen nejpravděpodobnější. Nejpravděpodobnější

stav robota z množiny odhadů se dá snadno získat nalezením částice s nejvyšším ohodnocením vzhledem k aktuálním datům.

Testovací robotická platforma poskytuje měření z laserového dálkoměru, které se nejvíce hodí na stavbu mapy obsazenosti. Na rozdíl od *feature-based* nebo vektorové mapy, pro které bychom museli z měření extrahovat zajímavé objekty, jako jsou rohy, případně přímky. Vektorové mapy a mapy obsazenosti nám poskytují navíc informaci o rozmístění překážek, což je důležité pro plánovací algoritmy. Nevýhodou mapy obsazenosti jsou větší nároky na paměť, závislé na velikosti mapy. Nakonec byla vybrána mapa obsazenosti pro FastSLAM algoritmus, protože prostředí, ve kterém se bude robot pohybovat, je menší a mapu je možné přímo použít pro plánovací algoritmy.

3.4.1 FastSLAM

Algoritmus si udržuje množinu částic χ, popisující aktuální stav robota v čase t značený $x_t^{[k]}$. Každá částice má v čase t vlastní mapu obsazenosti $m_t^{[k]}$ prostředí. Kde k je index částice. Celkový počet částic je označen M.

Algoritmus v první části, hlavní smyčce, provádí aktualizaci všech částic. Prvním krokem je získání částice, reprezentující pravděpodobnost v čase $t-1$ a upravení stavu robota pro čas t, použitím pravděpodobnostního modelu pohybu popsaného dále. Následující krok vypočítává váhu $w_t^{[k]}$ částice na základě nového stavu robota $x_t^{[k]}$, měření ze senzorů z_t a mapy $m_{t-1}^{[k]}$ částice. Nakonec se upraví mapa každé částice tak, aby odpovídala měření a novému stavu robota. Úprava mapy probíhá v metodě uprava_mapy popsané dále.

Když máme částice a jejich mapy aktualizované, tak se na základě spočítané váhy částic vybírá nová množina částic, která bude použita pro další časový krok jako vstup. Částice jsou do nové množiny vybrány s pravděpodobností $\alpha w_t^{[i]}$, kde α je normalizační faktor:

$$\alpha = \frac{1}{\sum_{k=1}^{M} w_t^{[k]}} \tag{3.17}$$

Výběr částic zajišťuje, že algoritmus si neustále udržuje množinu velmi pravděpodobných stavů robota, která ale může obsahovat i méně pravděpodobné stavy, které budou v budoucnosti lepší.

Celý FastSLAM algoritmus je popsán v tabulce 3.1.

Mapa

Základní myšlenkou map obsazenosti [41] je reprezentovat mapu jako pole náhodných proměnných, uspořádaných do rovnoměrné mřížky (gridu). Každá náhodná proměnná je binární a odpovídá obsazenosti prostoru, který reprezentuje. Algoritmus FastSLAM implementuje aproximaci odhadu pravděpodobnosti těchto náhodných proměnných.

Výhodou použití gridové mapy obsazenosti je, že algoritmus nepotřebuje žádné předdefinované orientační body a bez dalších úprav ji lze předat jako vstup algoritmů pro plánování cesty.

Nechť m_i značí buňku mapy, potom mapa obsazenosti rozdělí prostor na spoustu buněk mřížky:

$$m = \{m_i\} \tag{3.18}$$

```
1  Algoritmus FastSLAM_grid_map(χ_{t-1}, u_t, z_t):
2  χ̄_t := χ_t := ∅
3  for k := 1 to M do
4     x_t^[k] := model_pohybu(u_t, x_{t-1}^[k])
5     w_t^[k] := mereni_z_mapy(z_t, x_t^[k], m_{t-1}^[k])
6     m_t^[k] := uprava_mapy(z_t, x_t^[k], m_{t-1}^[k])
7     χ̄_t := χ̄_t + (x_t^[k], m_t^[k], w_t^[k])
8  endfor
9
10 for k:= 1 to M do
11    vyber i s pravdepodobnosti αw_t^[i]
12    χ_t := χ_t + (x_t^[i], m_t^[i])
13 endfor
14
15 return χ_t
```

Tabulka 3.1: FastSLAM algoritmus (převzato z [41]).

Každá m_i má přiřazenou binární proměnou, popisující obsazenost dané buňky. Pro obsazenou buňku budeme předpokládat hodnotu 1 a pro neobsazenou 0. Potom $p(m_i = 1)$ nebo $p(m_i)$ popisuje pravděpodobnost, zda je buňka obsazená. Abychom snížili výpočetní náročnost, tak se problém nalezení pravděpodobnosti mapy rozdělí do spousty oddělených podproblémů a to nalezení pravděpodobnosti pro jednotlivé buňky.

Úprava mapy Rozdělením problému jsme získali podproblémy, které jsou binárními odhady se statickým stavem. Na řešení tedy můžeme použít binární Bayesův filtr a místo násobení pravděpodobností použijeme logaritmickou reprezentaci pro snížení výpočetní náročnosti:

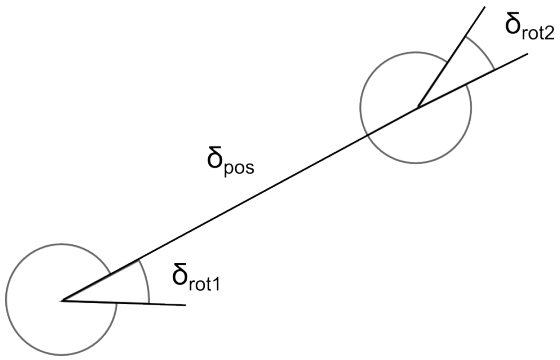
$$l_{t,i} = \log \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})} \tag{3.19}$$

Zpětně můžeme získat pravděpodobnost snadno pomocí vzorce:

$$p(m_i|z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 - e^{l_{t,i}}} \tag{3.20}$$

Následný algoritmus upravuje pomocí Bayesova filtru pravděpodobnosti obsazenosti jednotlivých buněk (převzato z [41]):

```
1  Algoritmus uprava_mapy(z_t, x_t^[k], m_{t-1}^[k]):
2
3  for k:=1 to K
4     pro vsechna m_i od pozice senzoru ve smeru mereni
5     a do vzdalenosti z_t
6        l_{t,i} := l_{t,i} + l_vol - l_0
7  endfor
8  pro m_i od pozice senzoru ve smeru mereni a vzdalenosti z_t
9  l_{t,i} := l_{t,i} + l_obs - l_0
```



Obrázek 3.6: Model pohybu robota v čase $[t-1, t]$ na základě informace z odometrie. Aproximací otočením o úhel δ_{rot1}, následovaný posunem δ_{pos} a druhým otočením o úhel δ_{rot2}.

Model pohybu

Modul Chassis poskytuje informaci o odometrii robota v pravidelných intervalech popisující relativní pozici robota vzhledem ke startu. Tuto informaci budeme brát jako vstup modelu pohybu, proto budeme tento model nazývat odometrický. Odometrický model pohybu počítá s měřeními z odometrie, namísto informací o řízení.

Praktické experimenty potvrzují, že odometrie, stále chybová, je častokrát přesnější než výpočty založené na řídících informacích. Obojí trpí na prokluzy kol a posun, ale řídící informace jsou zatíženy ještě nepřesností mezi aktuálním pohybem a jeho hrubým matematickým modelem. Ačkoliv je odometrie dostupná pouze zpětně, po tom, co se robot pohnul, FastSLAM algoritmu to nečiní problém.

V časovém intervalu $[t-1, t]$, se robot přesune z pozice $\overline{x}_{t-1} = (\overline{x}, \overline{y}, \overline{\theta})$ do pozice $\overline{x}_t = (\overline{x}', \overline{y}', \overline{\theta}')$. Známe tedy posun v relativních souřadnicích robota, definující pohybovou informaci jako:

$$u_t = \begin{pmatrix} \overline{x}_{t-1} \\ \overline{x}_t \end{pmatrix} \tag{3.21}$$

Pro získání relativní odometrie je u_t transformováno do série tří kroků, které definují libovolnou změnu pozice v rovině: rotace na místě startu, posun po přímce a jiné rotace na místě cíle. Obrázek 3.6 znázorňuje tuto dekompozici.

Algoritmus modelu pohybu, popsaný v tabulce 3.2, popisuje odometrický model pohybu, který na začátku počítá jednotlivé kroky předešlé dekompozice. Následně na základě jejích výsledků posune vstupní pozici x_{t-1} stejným pohybem, jen zatíženým chybou s normálním rozdělením v každé části (rotace, posun, rotace).

Model měření z mapy

Lokalizace v naší implementaci předpokládá použití laserového vzdálenostního senzoru. Jeho výhodou je vysoká přesnost měření s malým rozptylem a úzkým kuželem paprsku. Z důvodů použití mapy obsazenosti a úzkého kužele měření, budeme zacházet s jednotlivými měřeními, jako kdyby to byli přímé paprsky. Dále předpokládáme, že chyba měření je popsána normálním rozdělením se středem z_t a směrodatnou odchylkou σ_{mer}, danou typem použitého laserového dálkoměru.

```
1  Algoritmus model_pohybu(u_t, x_{t-1}):
2
3  δ_rot1  := atan2(ȳ' − ȳ, x̄' − x̄) − θ̄
4  δ_pos   := hypot(ȳ' − ȳ, x̄' − x̄)
5  δ_rot2  := θ̄' − θ̄ − δ_rot1
6
7  δ̂_rot1  := δ_rot1 + chyba(0, σ²_rot1)
8  δ̂_pos   := δ_pos + chyba(0, σ²_pos)
9  δ̂_rot2  := δ_rot2 + chyba(0, σ²_rot1)
10
11 x'  := x + δ̂_pos cos(θ + δ̂_rot1)
12 y'  := y + δ̂_pos sin(θ + δ̂_rot1)
13 θ'  := θ + δ̂_rot1 + δ̂_rot2
14
15 return x_t = (x', y', θ')
```

Tabulka 3.2: Algoritmus modelu pohybu (převzat z [41]).

```
Algoritmus mereni_z_mapy(z_t, x_t^[k], m_{t-1}^[k]):
w := 1
for k := 1 to K do
  vypocet z_map vrhanim paprsku v mape
  p := f(z_t^[k] − z_map)
  w := w * p
endfor
return w
```

Tabulka 3.3: Algoritmus mereni z mapy.

Pro ohodnocení částice daným měřením z_t a stavem robota x_t, musíme být schopní získat vzdálenost překážky od robota v určitém směru z obsazenostní mapy. Za předešlých předpokladů můžeme vypočítat vzdálenost překážky v mapě od robota, technikou vrhání paprsku [40], známou z počítačové grafiky. Která prochází jednotlivé buňky mapy ve směru paprsku, dokud nedosáhne vzdálenosti buňky, od pozice senzoru rovnou maximálnímu dosahu senzoru nebo nenarazí na buňku, která má hodnotu náhodné proměnné rovnou obsazené buňce. Následně vrátí vzdálenost nalezené buňky od pozice senzoru nebo maximální dosah senzoru.

Váhu částice spočítáme jako součin hodnot funkce normálního rozdělení 3.22 se středem odpovídajícím měřením $z_t^{[i]}$ a rozptylem σ^2_{mer}, pro odhadnuté vzdálenosti $z^{[i]}_{map}$ z mapy pro aktuální stav robota. Výsledný algoritmus měření z mapy je popsán v tabulce 3.3.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.22}$$

3.5 Chassis

Vzhledem k vzniklé robotické platformě (popsané v kapitole 5) v rámci této diplomové práce, budeme uvažovat typ řízení podvozku diferenciální (tank). Diferenciální podvozek jsme zvolili pro jeho lepší manévrovací schopnosti oproti ackermanovu a možnosti přestavět školní robotickou platformu MOB-2. Pro tento typ podvozku bude modul Chassis zajišťovat udržování zvolených rychlostí na jednotlivých kolech, pomocí PID regulační smyčky.

PID regulační smyčka je velmi rozšířený regulační mechanizmus pro ovládání procesů, na základě měření ze senzorů a řízením manipulační proměnné. Tento způsob regulace je velmi stabilní, při správném nastavení parametrů nedochází k oscilacím a je výpočetně nenáročný.

Na základě informací z enkodérů (umístěných na hnaných kolech) modul dále vypočítává svojí relativní pozici vzhledem ke startu. Výpočet relativní pozice je založen na ujeté vzdálenosti a změně úhlu robota, vypočítaných na základě informací o změně otočení kol a typu podvozku. Pohyb robota je aproximován modelem, kdy se robot prvně otočí na místě o změnu úhlu a následně jede rovně ujetou vzdálenost. Tento model budeme nazývat natočení-jízda. Dalším modelem pohybu by mohla být jízda po kružnici, definovanou změnou úhlu a ujetou vzdáleností, případně jiné složitější matematické modely. Složitější modely vyžadují více výpočetního výkonu a je otázkou, zda poskytují vyšší míru přesnosti vzhledem k vynaloženému úsilí.

Základní model, natočení-jízda, s vysokou frekvencí aktualizací a malými změnami pozice dobře aproximuje jak jízdu po kružnici, tak i po jiných křivkách. O udržování informace o přesné pozici se stará modul Localization 2.1.4, proto můžeme v tomto modulu použít model pohybu jednodušší, s vyšší nepřesností. Důležitější je výpočetní nenáročnost zvoleného modelu.

3.5.1 PID regulátor

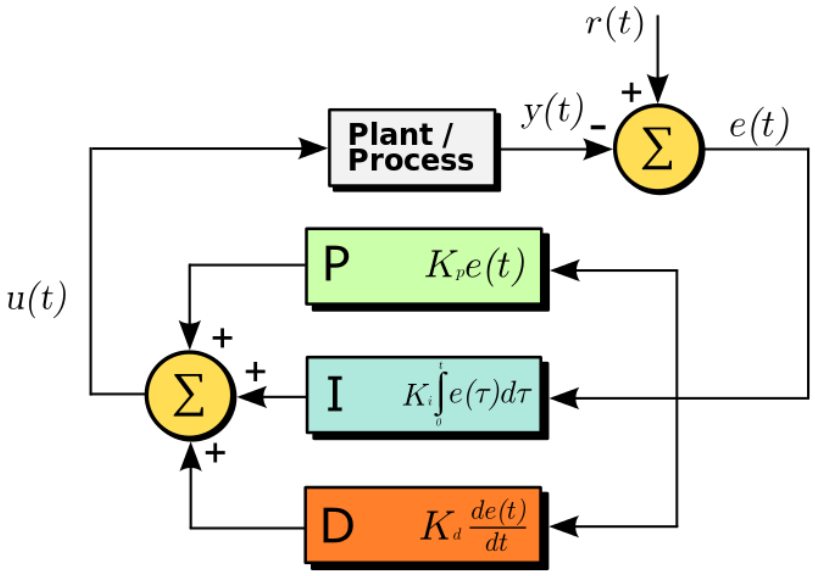
PID regulátor je řídící systém se zpětnou vazbou. Tudíž se regulátor snaží minimalizovat regulační odchylku $e(t)$, upravováním řízeného procesu pomocí manipulační proměnný $u(t)$.

Algoritmus PID regulátoru zahrnuje tři samostatné části: proporční, integrační a derivační značený P,I a D. Vstupem regulátoru je regulační odchylka $e(t)$ definovaná jako rozdíl mezi požadovanou hodnotou procesu $r(t)$ a měřenou hodnotou $y(t)$ pomocí senzorů. Části regulátoru lze interpretovat vzhledem k času následovně: P záleží na aktuální regulační odchylce a pevné konstantě K_p 3.23, I jsou nasčítány minulé regulační odchylky, vynásobené K_i 3.24 a D je předpověď budoucí regulační odchylky, jako směrnice změny aktuální regulační odchylky a poslední vynásobené K_d 3.25.

$$P = K_p e(t) \tag{3.23}$$

$$I = K_i \int_0^t e(t) \tag{3.24}$$

$$D = K_d \frac{de(t)}{dt} \tag{3.25}$$



Obrázek 3.7: Diagram popisující PID řídící smyčku.

Suma všech částí dohromady vytváří akční veličinu, která je výstupem PID regulátoru a je použita k úpravě řízeného procesu pomocí manipulační proměnné. Obecný PID regulátor je znázorněn na obrázku 3.7.

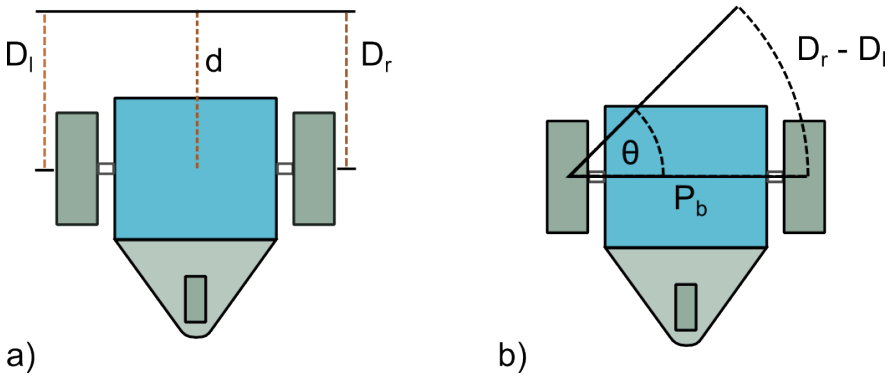
3.5.2 Výpočet relativní polohy

Pro diferenciální podvozek platí, že na základě změny pozice kol D_l a D_r se posun středu podvozku d rovná polovině součtu ujetých vzdáleností, tato skutečnost je ukázána na obrázku 3.8 část a) a popsána rovnicí 3.26. Dále si zvolíme střed levého kola jako referenční bod, který se nehýbe a vzhledem k tomuto bodu si odvodíme změnu natočení podvozku o úhel $\overline{\theta}$. Protože je pravé kolo kolmo k ose opisuje jeho pohyb, vzhledem k referenčnímu bodu, kružnici s poloměrem definovaným vzdáleností mezi středy kol P_b znázorněný na obrázku 3.8 část b). Střed robota se může sám pohybovat, ale diferenciální podvozek uvažujeme jako pevný, takže změna rotace se týká všech bodů podvozku stejně vzhledem k referenčnímu bodu. Na základě tohoto pozorování změna natočení středu podvozku odpovídá úhlu otočení $\overline{\theta}$ pravého kola. Úhel otočení pravého kola je v radiánech roven poměru rozdílu ujeté vzdálenosti mezi pravým D_r a levým D_l kolem podvozku a vzdálenosti středů kol P_b. Tato závislost je popsána v rovnici 3.27.

$$d = (D_l + D_r)/2 \tag{3.26}$$

$$\overline{\theta} = (D_r - D_l)/P_b \tag{3.27}$$

Relativní pozice podvozku $S = (x, y, \theta)^T$, kde x a y jsou souřadnice podvozku ve 2D vzhledem ke startu v metrech a úhel θ v radiánech. Iniciální pozice podvozku je $S = (0, 0, 0)^T$. Výpočet změny relativní pozice podvozku popisuje aproximační model, natočení-jízda, kdy se robot nejprve natočí o změnu úhlu $\overline{\theta}$ a následně rovně ujede vzdálenost d. Model, natočení-jízda, je popsán rovnicí 3.28. Aproximace výpočtu relativní pozice pomocí natočení a následné jízdy rovně, dává dobré výsledky, pokud se počítá často a na malých úsecích.



Obrázek 3.8: Obrázek popisuje ujetou vzdálenost d diferenciálním podvozkem a otočení diferenciálního podvozku o úhel θ, v závislosti na ujeté vzdálenosti D_l, D_r kol podvozku. Část a) zobrazuje ujetou vzdálenost d a část b) popisuje změnu natočení podvozku o úhel θ.

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} d\cos(\theta_{t-1} + \overline{\theta}) \\ d\sin(\theta_{t-1} + \overline{\theta}) \\ \overline{\theta} \end{pmatrix} \tag{3.28}$$

3.6 RangeFinder

Modul komunikuje pouze s laserovými dálkoměry firmy Sick, který po správné inicializaci vrací v pravidelných intervalech datové pakety, obsahující informace o vzdálenostech k nejbližším překážkám. Implementace komunikace byla převzata z databáze ROS uzlů.

4. Implementace řídícího systému

Řídící systém byl nasazen na robotech s operačním systémem Linux, s aplikovanou preempt úpravou jádra pro dosažení většího determinismu chování jádra a zajištění lepších odezev jednotlivých procesů změnou plánovače. Linux s preempt úpravou jádra byl zvolen, protože poskytuje známé vývojové a běhové prostředí pro programátora. Úlohy kritické na správné časování jsou řešeny v samostatných hw modulech, viz. kapitola 5, se kterými se komunikuje po I2C nebo sériové lince. Některé procesy spuštěné v Linuxu mají vyšší nároky na časování, ale občasné mírné zpoždění nenarušuje jejich funkčnost, preempt jádro toto poskytuje. Celý řídící systém a i vzniklá robotická platforma jsou zamýšleny jako výuková školní platforma, proto musí být co nejvíce uživatelsky přívětivá i za cenu mírné ztráty výkonu.

Komunikaci mezi moduly řídícího systému zajišťuje ROS [28] a jeho systém zveřejňování zpráv na návěstí (topics) a volání služeb zaregistrovaných do ROS. ROS dovoluje snadné zaznamenávání jednotlivých zpráv do souboru a následné přehrávání zaznamenaných zpráv zpět do řídícího systému. Záznam zpráv usnadňuje testování systému bez robota a vizualizaci, co se v řídícím systému dělo. Při testování se hodí možnost spouštět jednotlivé ROS uzly (moduly) na různých počítačích, s komunikací uzlů prostřednictvím zpráv a služeb po síti. Komunikaci po síti zajišťuje sám ROS.

ROS obsahuje různé sady podpůrných nástrojů, například rviz pro vizualizaci dat a databázi již vytvořených uzlů. Pokud je celý systém tvořen pouze ROS uzly, dá se seskupit do takzvané package, kterou si další uživatel může snadno nainstalovat. ROS uzly řídícího systému jsou pouze obalením samostatných tříd až na Control, který se dají samostatně použít bez frameworku ROS.

Jazyky pro tvorbu řídícího systému byli zvažovány tyto C++, C# a Java, protože to jsou všechno moderní objektové jazyky, vhodné pro tvorbu složitějších systémů, založených na objektovém návrhu. C# byl vyřazen jelikož jako operační systém robota byl vybrán Linux a v něm není úplná implementace. Dále u robota byl předpokládán nižší výpočetní výkon a omezující velikost paměti RAM, která bude intenzivně využívána řídícím systémem. Na základě tohoto kritéria byl zvolen jazyk C++ pro tvorbu řídícího systému. Překladač jazyka C++ zdrojový kód překládá přímo do strojového kódu a nepotřebuje virtuální stroj, který interpretuje mezikód, jako Java, čímž spotřebovává další výkon a operační paměť. Výhodou jazyka Java by byla vysoká přenositelnost mezi různými druhy hw, ale u robota se počítá se stejnou hw architekturou a nevadí kompilace. Jazyk C++ je velmi rozšířený při tvorbě výkonově náročných aplikací, mezi které bezpochyby robotika patří. Obzvláště při použití výkonově a paměťově méně vybavených hw řešení. V neposlední řadě, vybraný robotický framework ROS má plnou podporu pro jazyk C++ na rozdíl od Javy, kde je podpora stále experimentální.

4.1 Sdílené struktury a funkce

Při návrhu modulů systému se ukázalo, že více modulů bude používat stejné datové struktury a funkce. Společné části byli odděleny do samostatných souborů, kompilovaných do dvou dynamických knihoven tsqueue a basic. Výhodou

dynamických knihoven je neredundance informací, dynamické linkování za běhu a při chybě v knihovně mimo hlavičkový soubor odpadá nutnost překompilovávat celý modul. Dynamickou knihovnu stačí zkompilovat samostatně a zaměnit za starou.

4.1.1 tsqueue

Šablonová třída tsqueue implementuje oboustrannou frontu pro libovolný datový typ nebo třídu, která je bezpečná pro použití s vlákny. Třída tsqueue je postavena na základní třídě std::deque ze standardní šablonové knihovny (STL).

Šablonová třída std::deque je oboustranná fronta bez zamykacích mechanizmů a nemůžeme od ní očekávat žádné pevně dané chování při čtení a zápisu z více vláken zároveň.

Následuje definice třídy tsqueue a jejích privátních členů bez veřejných funkcí:

```
1 template<class T>
2 class tsqueue {
3   std::deque<T> dataDeque;
4
5   std::mutex lck;
6   std::condition_variable emptyCondition;
7 public:
```

Tsqueue za použití třídy std::mutex a std::lock_guard uzamyká celou třídu při přístupu k jejím prvkům. Čímž je zajištěna konzistence dat při současných požadavcích z více vláken. Získaní prvku z fronty může být blokující, pokud je fronta prázdná a volající použije funkci front() a back() nebo neblokující při použití funkce try_front(T &element) a try_back(T &element). Blokující přístup uspí volající vlákno přes std::condition_variable a probudí ho až bude dostupný požadovaný prvek ve frontě.

Následuje ukázka kódu zamknutí třídy, při pokusu o získání prvního prvku neblokujícím způsobem z fronty a dotazu na prázdnost fronty:

```
1   bool try_front(T &element) {
2     std::lock_guard < std::mutex > guard(lck);
3     if (dataDeque.empty())
4       return false;
5
6     element = dataDeque.front();
7     return true;
8   };
9
10  bool empty() {
11    std::lock_guard < std::mutex > guard(lck);
12    return dataDeque.empty();
13  }
14  ;
```

V obou případech se na začátku funkce čeká na získání zámku třídy. Při získání zámku je zajištěno, že nikdo jiný nebude přistupovat k prvkům ani funkcím

std::deque. Funkce try_front musí otestovat jestli není fronta prázdná, výsledek testu vrací jako návratovou hodnotu a pokud existuje první prvek, tak je přiřazen do vstupní proměnné. Funkce empty po získání zámku vrací, zda-li je fronta prázdná či nikoliv.

Ukázka blokujícího volání pro získání prvního prvku a vložení prvku na konec fronty:

```
1  T front() {
2    std::unique_lock < std::mutex > uq_lck(lck);
3    emptyCondition.wait(uq_lck,
4      [this] {return !dataDeque.empty();});
5    T copy = dataDeque.front();
6    dataDeque.pop_front();
7    return copy;
8  }
9  ;
10
11 void push_back(T element) {
12   std::lock_guard < std::mutex > guard(lck);
13   dataDeque.push_back(element);
14   emptyCondition.notify_one();
15 }
16 ;
```

V první funkci je použita std::condition_variable pro uspání vlákna, na základě lambda funkce testující prázdnost fronty, dokud fronta neobsahuje nějaký prvek. Druhá funkce při vložení prvku do fronty probudí jedno z čekajících vláken na prvek, pokud takové existuje, voláním funkce notify_one() na proměnné std::condition_variable.

4.1.2 basic

Knihovna basic sdružuje obecné datové struktury, jako pozici v rovině, reprezentovanou dvojicí doublů strukturou Position a stav robota State, poděděný od Position s přidaným doublem reprezentujícím natočení robota. Obsahuje i obecné funkce, používané v robotice, jako normalizaci úhlu do intervalu $[-\pi, \pi]$, ořezání hodnoty do určitého intervalu $[min, max]$ a výpočet kontrolního součtu ověřující integritu přenesených dat z dekodéru signálu z kvadraturních enkodérů (popsaného v kapitole 5.1.4).

Struktuře Position a State byli přetíženy základní operátory + a - tak, aby bylo možné struktury stejného typu sčítat a odčítat po složkách. Zároveň ještě byli přetíženy operátory * a / pro skalární násobení a dělení složek. Struktura Position má navíc implementovány funkce distance(const Position &position) a angle(const Position &position).

Funkce distance vrací vzdálenost mezi Position, použitou pro volání funkce a předanou pozicí, jako parametr funkce. Výpočet je realizován voláním matematické funkce std::hypot, vracející velikost přepony trojúhelníku s odvěsnami, definovanými vstupními parametry.

Funkce angle vrací úhel v pravotočivé soustavě, který svírá přímka s osou x,

definovaná pozicí třídy, použité pro volání a předanou pozicí. Výpočet je realizován voláním matematické funkce std::atan2, vracející úhel v plném rozsahu $[-\pi, \pi]$.

Kód funkce distance a angle:

```
1  double distance(const Position &position) const{
2    return std::hypot(x - position.x, y - position.y);
3  }
4
5  double angle(const Position &position) const{
6    return std::atan2(position.y - y,position.x - x);
7  }
```

Funkce valueInRange je šablonou funkcí pro ořezání hodnoty do určeného intervalu, vyžadující aby použitý typ měl definovanou operaci porovnání. Funkce je použita například pro ořezání výstupu z PID regulátoru, do intervalu vstupu plnění motorů. Pro porovnání, zda-li je hodnota v intervalu jsme použili ternární operátor. Uživatel může použít dvě přetížení této funkce. První ořezává hodnotu do symetrického intervalu, definovaného jednou vstupní proměnnou. Druhý ořezává hodnotu do asymetrického intervalu, definovaného minimální a maximální hodnotou intervalu.

Kód funkce valueInRange:

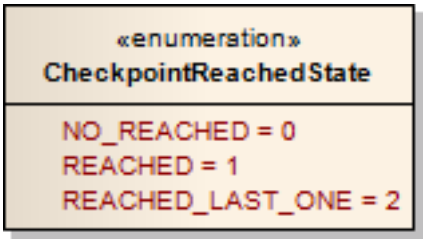
```
1  template <class V> inline V valueInRange(
2  V value, V lowerBoundary, V higherBoundary) {
3    return value < lowerBoundary ? lowerBoundary :
4         value > higherBoundary ? higherBoundary : value;
5  }
6
7  template <class V> inline V valueInRange(
8  V value, V range) {
9    return valueInRange(value,-range,range);
10 }
```

4.2 Control

Modul je rozdělen na hlavní řídící vlákno, které spravuje volání služeb, poskytovaných ostatními moduly dle stavu robota a příjem zpráv z ostatních modulů skrze zaregistrování na odpovídající ROS topic, voláním funkce subscribe() s názvem topic, velikostí fronty zpráv a ukazatelem na funkci, která bude daný typ zpráv zpracovávat. Příjem zpráv a volání odpovídajících funkcí zpracovávajících daný typ zprávy zajišťuje ROS ve funkci spin().

4.2.1 Příjem a zpracování zpráv

Control přijímá zprávy popisující relativní pozici podvozku z modulu Chassis, pozici robota v mapě z modulu Localization, cílový bod v mapě z vizualizačního nástroje rviz, informace o dosažení jednotlivých checkpointů a stavu fronty



Obrázek 4.1: UML třída možných hodnot zprávy, popisující stav fronty průjezdních bodů modulu CheckpointMovement.

z modulu CheckpointMovement a informace o vzdálenostech k nejbližší překážce z modulu RangeFinder.

Zprávy o relativní pozici a pozici robota v mapě obsahují časové značky, které jsou použité pro párování odpovídajících dvojic relativní pozice a pozice v mapě. V případě, že časové značky úplně neodpovídají, tak se vybere dvojice relativních pozic, mezi které časově zapadá pozice v mapě a lineární interpolací se ze dvou vybraných relativních pozic vytvoří nová relativní pozice se stejnou časovou značkou, jakou má pozice v mapě. Každá dvojice pozic relativní a mapový, se stejnou časovou značkou, se použije pro správné nastavení transformačních matice mezi těmito souřadnicovými systémy. Tyto transformační matice spravuje třída FrameConverter a vytváří se voláním funkce setRotationTranslation s parametry, odpovídající relativní a mapové pozici. Výpočet transformačních matic popisují rovnice 3.2 a 3.3.

Zpráva popisující cílový bod je použita pro volání služby add_payoff_ellipse na modulu PathPlannig pro vložení cílového bodu do užitkové funkce a zároveň je nový cílový bod přidán do množiny cílových bodů modulu Control.

Na základě informací o dosažení jednotlivých průjezdních bodů, vygenerovaných v řídící smyčce modulu Control a předaných do modulu CheckpointMovement a stavu fronty průjezních bodů, je upravována proměnná actualSteps_ udržující stav o počtu předaných průjezdních bodů. Při úpravě proměnné actualSteps_ mohou nastat dva případy: buď přijde zpráva jen o dosažení průjezdního bodu, pak je proměnná actualSteps_ dekrementována o jedničku nebo dorazí zpráva informující o dosažení posledního průjezdního bodu ve frontě, což vede k vynulování proměnné actualSteps_. Obrázek 4.1 popisuje všechny možné hodnoty zprávy nesoucí informaci o stavu fronty průjezdních bodů.

Vzdálenosti od překážek ze zpráv od modulu RangeFinder jsou procházeny podle algoritmu reaktivní detekce 3.1.1. Pokud virtuální nárazníková zóna obsahuje překážku, tak je robot okamžitě zastaven a je nastavena řídící proměnná wall_ na true, aby hlavní řídící smyčka mohla adekvátně reagovat. Když je virtuální nárazníková zóna prázdná, je wall_ nastaven na false.

4.2.2 Hlavní řídící vlákno

Algoritmus Control, popsaný v části 3.1 o použitých algoritmech, je implementován v hlavním řídícím vlákně s malou úpravou, že na začátku vlákno čeká, než je na inicializována třída FrameConverter. Dokud FrameConverter není na inicializován, tak nejsme schopní převádět body mezi souřadnicovými systémy, což je nutná funkčnost pro správné vykonávání řídící smyčky.

Řídící smyčka se vykonává neustále, dokud není ukončen modul Control v pravidelných intervalech 100Hz, zajištěných ROS třídou ros::Rate. Interval 100Hz byl zvolen na základě omezené maximální rychlosti robota na 0.3 ms^{-1} a výpočetního výkonu řídící jednotky. Zvýšení frekvence řídící smyčky by znamenalo častější reakci na aktualizovanou pozici robota, ale také zvýšení využitého výpočetního výkonu řídící jednotky hlavním řídícím vláknem. Při této frekvenci je maximální změna relativní pozice 3mm mezi dvěma cykly smyčky. Reakce na změnu pozice a stavu robota je dostatečně častá pro správné vykonávání řídící smyčky při zvolené frekvenci.

4.3 CheckpointMovement

Dříve popsané chování modulu implementuje třída checkpointMovementHermit, starající se o postupný průjezd vloženými průjezdními body, do oboustranné fronty tsqueue 4.1.1, podle algoritmu pro sledování hermitovské křivky 3.2.2. Třída checkpointMovementHermit implementaci algoritmu sledování hermitovské křivky obaluje nekonečnou smyčkou, která je ukončena až voláním destruktoru třídy při ukončení práce modulu.

Nekonečná smyčka, jejíž kód je popsán v tabulce 4.3, se na začátku pokusí vyjmout první průjezdní bod z fronty, pokud nebyl celý modul pozastaven (pozastavení modulu indikuje proměnná pause_), když je fronta prázdná (nepovedlo se získat průjezdní bod) nebo je modul pozastaven, smyčka se na chvíli uspí a po probuzení se pokusí znovu získat první průjezdní bod z fronty. Jestliže byl získán nový průjezdní bod je předán společně s předešlým průjezdním bodem nebo aktuální pozicí (pokud robot stál) funkci moveToCheckpoint implementující algoritmus pro sledování hermitovské křivky. Po návratu z funkce moveToCheckpoint je aktuální průjezdní bod změněn na předešlý, pokud jím podvozek robota projel.

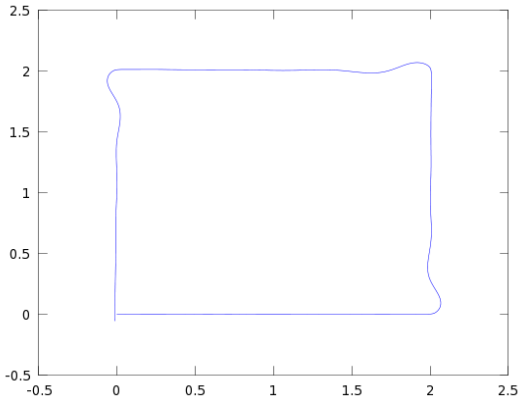
Funkce moveToCheckpoint je blokující, dokud robot nedorazí na aktuální průjezdní bod nebo není zvenčí volána funkce měnící začátek fronty, případně pozastavující celý modul. Pozastavení a obnovení jízdy na aktuální průjezdní bod zvenčí, probíhá přes nastavování hodnoty proměnné pause_. Když je průjezdní bod dosažen, je zavolána callback funkce, definována zvenčí s informací o aktuálním stavu fronty. Callback funkce je v současné implementaci použita pro generování ROS zpráv, informující ostatní moduly o stavu fronty.

Z venčí třídy je práce s průjezdními body omezena na přidávání nových bodů na začátek nebo konec fronty a smazání všech průjezdních bodů. Dosažení průjezdního bodu s určitou tolerancí vyvolá jeho smazání z fronty. Při přidání průjezdního bodu na začátek fronty je ihned ukončena jízda na aktuální průjezdní bod a je následně započata na přidaný průjezdní bod.

Algoritmus pro sledování hermitovské křivky má dva parametry predikční vzdálenost a velikost kroku generování bodů na hermitovské křivce. Při testovacích jízdách se ukázalo 5cm jako vhodná vzdálenost pro predikci bodů a nastavení velikosti kroku generování bodu rovnicí:

$$s = 1.0/(vzdalenost * bodu_{metr}) \tag{4.1}$$

Tato rovnice popisuje velikost kroku jako převrácenou hodnotu vzdálenosti mezi krajními body křivky a počtem kroků na metr vzdálenosti (v implementaci nastavený na tisíc). Pomocí těchto nastavení je robot schopný hermitovskou křivku



Obrázek 4.2: Zobrazení odometrických informací při sledování hermitovské křivky definované vrcholy čtverce o straně délky 2m s nulovými výstupními vektory. Vyboulení po průjezdu vrcholy čtverce je způsobeno jízdou po otáčení na místě, dokud není úhel mezí následujícím bodem hermitovské křivky a aktuálním natočením robota menší než 30 stupňů.

sledovat plynule. Testovací jízdy byli prováděny na rovné ploše o rozloze 3x4metry a byl použit robot o rozměrech 30x35cm, popsaný v kapitole 5. Trajektorii jedné z testovacích jízd podle informací z odometrie, ukazuje obrázek 4.2.

4.4 PathPlanning

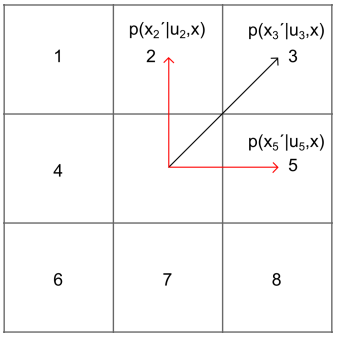
Modul PathPlanning pro generování cesty, na základě požadavku, používá implementaci iterace hodnoty. Popsané algoritmy iterace hodnoty a nalezení optimálni politiky 3.3.1 z předešlé kapitoly, jsou implementovány třídou CostMap. Třída CostMap vypočítává funkci hodnoty 3.15 do dvourozměrného pole odpovídajícího velikostí mapě prostředí. Toto dvourozměrné pole je reprezentováno třídou Grid, udržující si informace o velikosti pole, implementující přetížený operátor přiřazení pro správné kopírování obsahu dynamicky vytvořeného pole a obalující přístup k prvkům pole funkcí, která může testovat, zda se nesnažíme získat prvek mimo rozsah pole. Na základě vypočítané funkce hodnoty a možných pohybech robota je založená funkce getBestMove(), implementující algoritmus nalezení optimální politiky 3.3.1, vracející nejlepší typ pohybu pro libovolnou pozici v mapě předanou jako argument.

4.4.1 Pravděpodobnost pohybu robota

Výpočet funkce hodnoty provádíme nad dvourozměrným polem, kde předpokládáme, že se robot pohybuje pouze po mřížce a může vždy z aktuálního pole do všech sousedících. Neomezuje nás v budoucím pohybu aktuální natočení robota. Zároveň zavádíme určitou míru nepřesnosti, kdy robot může na místo cílového políčka skončit v sousedících polích s parametrem p_{chyba} určenou pravděpodobností. Pro pohyb u_3 a pozici x je pravděpodobnost, že skončíme v sousedním políčku 2 $p(x'_2|u_2, x) = p_{chyba}$, jak znázorňuje obrázek 4.3. Z předchozích předpokladů plyne, že pro každé políčko existuje právě devět pohybů (pohyb do všech sousedících polích a stání na místě).

```
while (!end_) {
  bool newCheckpoint = false;
  if (!pause_) {
    std::lock_guard < std::mutex > lock(frontMutex_);
    newCheckpoint = checkpointsQueue_.try_front(target);
    checkpointChanged_ = false;
  }

  if (!pause_ && newCheckpoint) {
    if (incorrectLast) {
      State state = chassis_.getState();
      VelocityWheels velocityWheels =
        chassis_.getVelocityWheels();
      double velocity = (velocityWheels.left +
        velocityWheels.right) / 2.0;

      last.position = state;
      last.outVector = Vector(
        cos(state.theta) * velocity,
        sin(state.theta) * velocity);
      incorrectLast = false;
    }

    moveToCheckpoint(last, target);
    stopRobot = true;

    if (checkpointChanged_) {
      incorrectLast = true;
    } else {
      last = target;
    }
  } else {
    if (stopRobot) {
      chassis_.stop(true);
      incorrectLast = true;
      stopRobot = false;
    }
    std::this_thread::sleep_for(
      std::chrono::milliseconds(20));
  }
}
```

Tabulka 4.1: Nekonečná smyčka modulu CheckpointMovement, vypočítávající výstupní vektory pro aktuální checkpoint. Aktuální checkpoint spolu s minulým je poté vložen do algoritmu pro sledování hermitovské křivky.



Obrázek 4.3: Ukázka pohybu robota po mřížce. Černá šipka znázorňuje chtěný pohyb a červený možnou chybu ve vykonávání.

Pohyb robota po mřížce je popsán strukturou Move, obsahující dvě proměnné x a y, reprezentující posun ze současného pole na sousedící, takže hodnoty proměnných jsou z množiny $\{-1, 0, 1\}$. Všech devět pohybů je zapsáno do tabulky 9x3, kde máme pro každý pohyb popsané i možné chybové pohyby.

Funkce počítající zisk z daného pohybu je popsána v tabulce 4.4.1. Pokud koncová pozice pohybu není v mapě, použije se jako zisk z této pozice nejmenší možná hodnota.

4.4.2 Užitková funkce

Pro výpočet funkce hodnoty je esenciální rychlá a snadno spočítatelná užitková funkce. Jelikož uživatel může definovat různý počet cílových bodů a vstupní mapa prostředí je popsána dvourozměrným polem obsazenosti prostoru. Nejpřímočařejší a výpočetně nenáročnou možností jak reprezentovat užitkovou funkci je dvourozměrné pole, reprezentované třídou Grid, . Jednotlivé buňky dvourozměrného pole užitkové funkce odpovídají jedna ku jedné buňkám mapy prostředí. Hodnota každé buňky přestavuje užitek, který robot získá, pokud by na odpovídající pozici dorazil. Všechny prvky pole jsou při každém přepočítání inicializováno na -1, odpovídající užitku robota za každý krok (výsledná funkce hodnoty upřednostňuje kratší cesty).

Pole užitkové funkce může být měněno payoff objekty, reprezentující mapu obsazenosti a elipsu použitou pro reprezentaci cílových bodů. Změna pole užitkové funkce většinou probíhá jako součet aktuální hodnoty prvku pole a hodnoty získané z payoff objektu. Obrázek 4.4 znázorňuje základní třídu PayoffObject a odvozené třídy PayoffOccupancyMap, reprezentující užitek na základě mapy obsazenosti a třídy PayoffElipse, používané jako cílový bod nebo cílová oblast. Každý payoff objekt musí mít implementovánu metodu updatePayoffTable, která upravuje pole reprezentující užitkovou funkci. Funkce getMinPayoff vrací nejmenší možný užitek, který daný payoff objekt vkládá do pole užitkové funkce.

Payoff objekty můžou být buď permanentní nebo dočasný. Permanentní jsou vkládány do speciální fronty pro znovu aplikování na pole užitkové funkce, při vynucené aktualizaci funkcí updatePayoffTable. Dočasné upraví pole pouze jednou a následně jsou ihned zapomenuty. Cílové body se reprezentují permanentními objekty a jsou mazány při průjezdu robota daným cílovým bodem. Mapa prostředí je popsána dočasným objektem, protože se průběžně mění a znova posílá do modulu PathPlanning z modulu Localization.

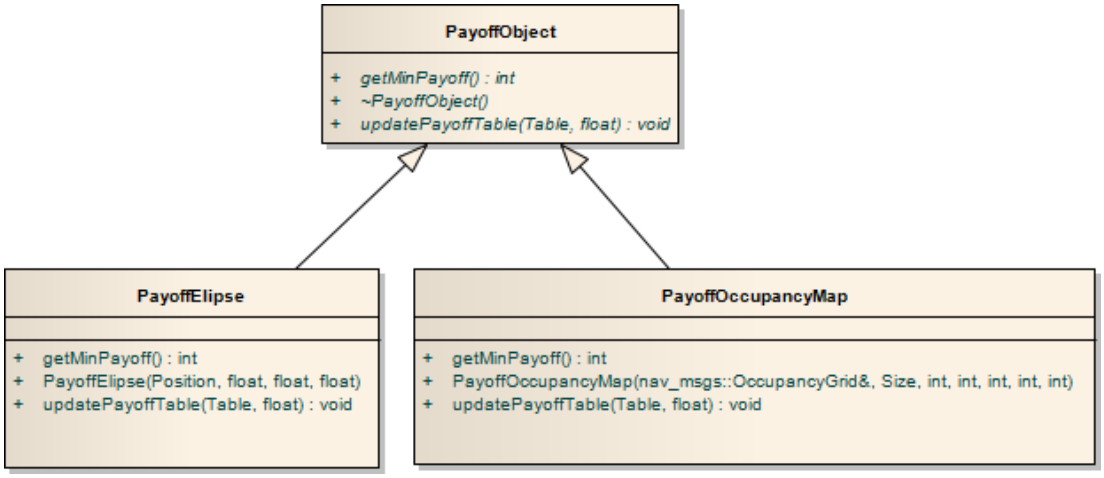
```
CostMap::Type CostMap::getPayoffFromMove(
  const unsigned int x, const unsigned int y,
  const unsigned int move, GridPtr costMap) {
  if (!payoffTable_.inGrid(x + moves_[move][1].x, y
    + moves_[move][1].y)) {
    return std::numeric_limits<Type>::min();
  }
  Type payoff = payoffTable_.value(x + moves_[move][1].x,
      y + moves_[move][1].y) - 1;
  payoff += 0.05
      * (costMap->inGrid(x + moves_[move][0].x, y
        + moves_[move][0].y) ?
          costMap->value(x + moves_[move][0].x,
          y + moves_[move][0].y) :
            std::numeric_limits<Type>::min());
  payoff += 0.9
      * costMap->value(x + moves_[move][1].x, y
      + moves_[move][1].y);
  payoff += 0.05
      * (costMap->inGrid(x + moves_[move][2].x, y
        + moves_[move][2].y) ?
        costMap->value(x + moves_[move][2].x,
        y + moves_[move][2].y) :
          std::numeric_limits<Type>::min());
  return payoff;
}
```

Tabulka 4.2: Funkce výpočtu zisku z daného pohybu pro určenou pozici v mapě. Pokud koncová pozice pohybu není v mapě použije se jako zisk z této pozice nejmenší možná hodnota.

PayoffOccupancyMap PayoffOccupancyMap objekt popisuje změnu užitkové funkce pro vloženou mapu prostředí. Každá buňka mapy je v jednom ze tří stavů prázdná, obsazená a nevíme vzhledem k pravděpodobnosti obsazenosti dané buňky. Podle stavu buňky je ke stejné buňce užitkové funkce přičten odpovídající užitek. V současné implementaci je to pro prázdnou buňku a buňku o které ten stav nevíme 0. Nepenalizujeme ani neodměňujeme robota při vstupu na pozice těchto buněk. Naopak v případě obsazené buňky, reprezentující překážku, penalizujeme robota extrémní hodnotou, aby se takovýmto buňkám vyhýbal.

4.4.3 Funkce hodnoty

Funkce hodnoty se vypočítává v samostatném vlákně, v případě požadavku z venčí, z důvodu delší časové náročnosti tak, aby třída mohla reagovat na ostatní podměty, jako je přidávání payoff objektů. Při probíhajícím výpočtu je současně uchovávána stará verze funkce hodnoty, na které se do dokončení výpočtu nové provádí hledání optimální politiky. Když je dokončena nová funkce hodnoty, tak je prohozena se starou pro ušetření alokace a dealokace paměti.



Obrázek 4.4: UML diagram payoff objektů.

4.5 Localization

Základní algoritmus MCL je vytvořen jako šablonová třída Mcl s dvěma parametrickými typy. První typ reprezentuje částice, které budou použité pro lokalizaci. Druhý definuje pro tyto částice obecný visitor podle návrhového vzoru visitor. Návrhový vzor visitor byl zvolen pro možnost rozšiřování práce s částicemi o nové senzory, výpočty rozptylu pozice částic a další. Takováto koncepce umožňuje použít šablonovou třídu Mcl jak pro řešení SLAM problému, tak obyčejné lokalizace se známou mapou. Obojí pro roboty, pohybující se v rovině, tak i ty pohybující se v prostoru (3D). Řešení specifického problému je určeno použitým typem částic a visitor tříd nad nimi.

Třída Mcl poskytuje funkci pro resamplování částic (nový výběr částic podle jejich váhy), vkládání částic do filtru, aplikaci visitor tříd na všechny částice a vrácení kopie nejlepší částice.

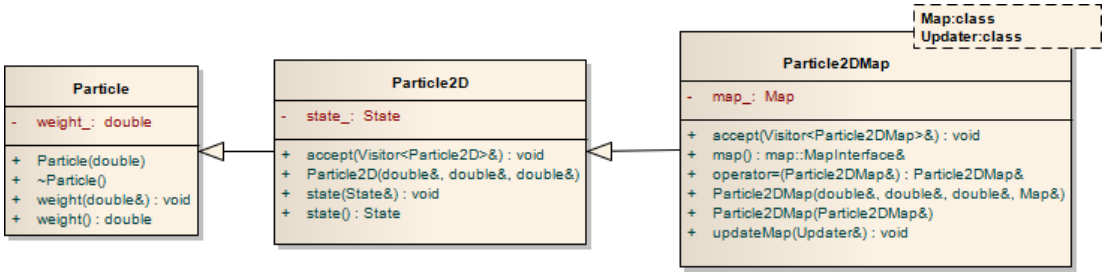
4.5.1 Částice

Nejmenší možná implementace částice, pro použití šablonovou třídou Mcl, je popsána třídou Particle. Třída Particle obsahuje jen proměnnou, reprezentující váhu částice, používanou pro výběr částic a odpovídající funkce sloužící pro čtení a nastavování této váhy. Na obrázku 4.5 je znázorněno, pomocí UML, dědění od této základní částice až k částici Particle2DMap, používané pro řešení SLAM problému.

Částice Particle2DMap obsahuje kromě své váhy také pozici v rovině vzhledem k mapě a mapu prostředí, reprezentovanou mapou obsazenosti. Funkcí updateMap se aktualizuje mapa částice na základě přijatých dat z laserového dálkoměru. Funkce accept slouží pro aplikaci visitor tříd na částici.

4.5.2 Visitory

Jak už bylo zmíněno výše, práce s částicemi probíhá na základě návrhového vzoru visitor. Všechny částice jsou umístěny ve spojovém seznamu, který je postupně procházen. Na každé částici je zavolána metoda accept s aktuálním visitorem.



Obrázek 4.5: UML diagram popisující základní třídu Particle a postupné dědění od ní.

```
1
2 template <class AdvancedParticle>
3 class Visitor{
4 public:
5     virtual ~Visitor(){};
6   virtual void visit(AdvancedParticle *particle) = 0;
7 };
```

Tabulka 4.3: Základní třída Visitor.

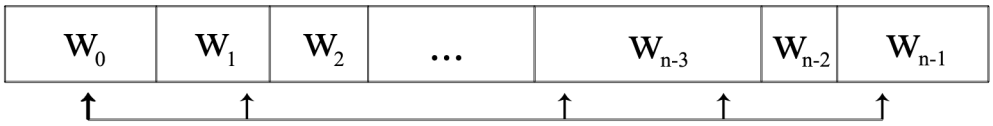
Visitor aplikuje požadovanou akci na každou částici nebo provede výpočet na základě informací částice. Základní třída Visitor je znázorněna v tabulce 4.5.2. Pro správnou funkčnost algoritmu FastSlam a výpis informací o částicích vznikli následující visitory:

- **Model pohybu** implementuje algoritmus modelu pohybu 3.2. Parametry pohybu se třídě předají při inicializaci (úhly rotace a vzdálenost posunu). Současně je při inicializaci třídy možné dále určit míru chyby jednotlivých částí pohybu. Bez určení míry chyby se předpokládá u každého pohybu chyba velikosti jednoho procenta z dané části pohybu. Chyba je modelována normálním rozdělením $\mathcal{N}(0, \sigma)$, kde σ odpovídá míře chyby.

  Aby generování velikosti chyb bylo náhodné, byl vybrán generátor náhodných čísel std::mt19937 ze standardní knihovny C++. Třída std::mt19937 je implementací Mersenne Twister pseudo náhodného generátoru, vracející náhodná čísla s rovnoměrným rozdělením pravděpodobnosti. Chybu modelu ale popisujeme normálním rozdělením, proto čísla z std::mt19937 jsou ještě modifikována třídou std::normal_distribution<double>. Tato třída generuje náhodná čísla tak, aby rozdělení pravděpodobnosti vygenerovaných čísel odpovídalo normálnímu;

- **Úprava váhy** vychází z principů popsaných v odstavci 3.4.1. Pozice středu měření, minimální a maximální úhel, rozptyl měření, velikost kroku, maximální dosah a posun jsou parametry konstruktoru visitoru. Posun definuje kolikáté měření z dálkoměru se použije pro výpočet váhy částice. Tato úprava slouží pro snížení výpočetní náročnosti a při vhodném nastavení nesnižuje přesnost celého FastSlamu.

  Visitor postupně prochází použitá měření a pro každé si z mapy a modelu senzoru vypočítá, jakou by mělo mít hodnotu. Rozdíl odhadnuté a namě-



Obrázek 4.6: Ukázka náhodného výběru pevného počtu částic. Pravděpodobnost, že částice na indexu i bude vybrána je určena její váhou w_i.

řené hodnoty je vstupem funkce $\mathcal{N}(0, \sigma_{senzor})$ 3.22. Součin všech výsledků z předešlé funkce pro danou částici, je uložen jako váha této částice;

- **Aktualizace mapy.** Na každou částici a měření z laserového dálkoměru aplikuje algoritmus úpravy mapy 3.4.1. Samotný algoritmus je implementován v samostatné třídě OccupancyUpdater, aktualizující předanou mapu. Před voláním aktualizace mapy částice, prostřednictvím OccupancyUpdateru, se musí spočítat pozice středu měření vzhledem k mapě. K tomu slouží rotační a translační matice, převádějící relativní pozici středu měření, vzhledem ke středu robota na absolutní pozici středu měření v mapě;

- **Průměr pozice částic** počítá aritmetický průměr pozice všech částic:

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{4.2}$$

- **Výpočet rozptylu pozic částic.** Při inicializaci očekává jako vstup aritmetický průměr pozic částic, aby mohl počítat rozptyl od tohoto průměru. Závisí tedy na výpočtu předchozí visitor třídy. Rozptyl pozice částic se vypočítá následujícím způsobem:

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (\overline{x} - x_i)^2 \tag{4.3}$$

4.5.3 Resamplování částic

Výběr nové množiny částic je realizován s časovou náročností $O(n)$, kde n je počet částic a pole $w = \{w_0, \ldots, w_{n-1}\}$ jsou váhy částic. K výběru nové množiny stačí dva průchody pole částic $c = \{c_0, \ldots, c_{n-1}\}$. Při prvním průchodu se spočítá součet vah částic $v = \sum_{i=0}^{n-1} w_i$. Před druhým průchodem se vypočítá posun $p = v/n$ a začátek $z = p * rand(0, 1)$. Během druhého průchodu se počítá průběžná suma $s = \sum_{i=0}^{k} w_i$ a částice c_i je vybírána dokud $s > ch * p + z$, kde ch je počet dosud vybraných částic. Ukázka výběru částic je na obrázku 4.6. Tento způsob výběru částic zachovává pravděpodobnost výběru částice a je rychlejší než n-krát generovat náhodné číslo, odpovídající vybrané částici.

```
int difference = newValue - oldValue;

if (abs(difference) < MAX_DIFFERENCE) {
  // overflow test
  return difference;
} else { // overflow
  if (difference < 0) {
    // top overflow
    return MAX_UINT16 + difference;
  } else {
    // bottom overflow
    return difference - MAX_UINT16;
  }
}
```

Tabulka 4.4: Test přetečení 16bitového čítače dekodéru.

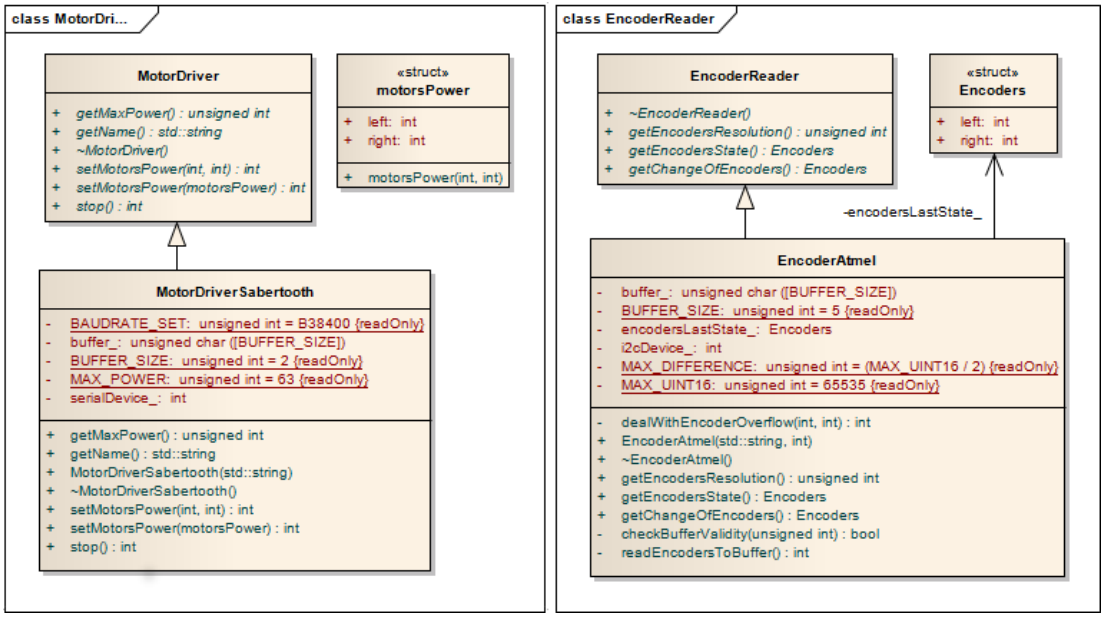
4.6 Chassis

Modul chassis je implementací interface BasicDifferentialChassis, popsaného v kapitole o Architektuře řídícího systému 2.1.5, pro testovací robotickou platformu třídou MobDifferentialChassis. Třída MobDifferentialChassis je závislá na implementacích rozhraní EncoderReader a MotorDriver znázorněných na obrázku 4.7.

EncoderReader je rozhraní pro získávání informací o otáčení kol robota, který třída EncoderAtmel implementuje. EncoderAtmel komunikuje s hw dekodérovou deskou 5.1.4, vyčítající signál z kvadraturních enkodérů, po sběrnici I2C. Informace o změně stavu kol z dekodérové desky je posílána jako dvojice 16bit unsigned int a 8bitový kontrolní součet těchto čísel. Kde 16bit unsigned int představuje stav čítačů dekodérové desky. Tato pětice je znova přepočítána stejným algoritmem pro výpočet kontrolního součtu jaký používá dekodérová deska, aby se ověřila integrita dat, jestli nedošlo k poškození dat během přenosu.

Informace o stavu kol se dá získat z třídy EndoderAtmel dvěma způsoby: dotazem na stav čítačů nebo na změnu od posledního dotazu. Pokud se ptáme na změnu, tak je vytvořen rozdíl mezi minulými a současnými přijatými daty a tento rozdíl je počítán jako typ int. Výsledný rozdíl je pak otestován, zda-li nedošlo k přetečení 16bitového čítače v dekodéru, pokud ano je rozdíl upraven přičtením nebo odečtením rozsahu uint16. Zdrojový kód testu je popsán v tabulce 4.6, kde MAX_DIFFERENCE je polovina velikosti rozsahu 16bit unsigned int.

MotorDriver je rozhraní pro komunikaci s výkonovou částí motorů, která nastavuje výkon a směr otáčení jednotlivých motorů. Poděděná třída MotorDriverSabertooth komunikuje s modulem Sabertooth 2x5 [38] od společnosti DimensionEngineering, který zajišťuje udržování výkonu jednotlivých kol dle přijatých hodnot. Výkon jednotlivých kol se nastavuje metodami setMotorsPower, které jsou přetížené a přijímají dvojici int samostatných nebo jako strukturu v rozmezí $[-127, 127]$ (-127 je plný výkon motoru vzad, 0 stop motoru a 127 plný výkon vpřed). Mezní hodnota 127 je vracena funkcí getMaxPower.



Obrázek 4.7: Znázornění třídy EncoderReader a od děděné EncoderAtmel a třídy MotorDriver a od děděné MotorDriverSabertooth pomocí UML.

PID regulátor O udržení správné rychlosti otáčení kol se stará dvojice PID regulátorů s frekvencí 200Hz. Každé kolo je řízeno samostatným PID regulátorem 3.5.1. Vstupem regulátoru $r(t)$ je požadovaná rychlost otáčení kol v ms^{-1} v čase t, od níž se odečte aktuální rychlost otáčení kola $y(t)$ v ms^{-1} v čase t, čímž vznikne regulační odchylka $e(t)$ v čase t. Parametry PID regulátoru jsou dané, dle typu motorů, strukturou DiffChassisParam 2.1.5. Výsledek regulační smyčky je oříznut do rozsahu výkonu motorů a předán funkci setMotorsPower třídy MotorDriver jako parametr.

Správná funkčnost PID regulátorů je závislá na časování a proto jsou regulační smyčky vykonávány v samostatném linuxovém vlákně, kterému je zvýšena priorita na 90 z rozsahu $[0, 99]$ a změněn typ plánovače na SCHED_FIFO, který zajišťuje, že vlákno, pokud je připraveno a nezpracovává se žádné jiné vlákno s vyšší prioritou, bude spuštěno. Manipulátor vlákna, priorita a typ plánovače je předán funkci pthread_setschedparam [39], která zajistí správné nastavení těchto hodnot pro vlákno, určené manipulátorem, v operačním systému. Nastavení priority a typu plánovače ukazuje následující část kódu:

Na konci PID smyčky se vždy vlákno uspí na zbývající dobu periody. Doba, na jakou se má proces uspat, je spočítána tak, že na začátku a konci PID regulátoru je sejmuta časová značka pomocí systémových hodin s mikrosekundovým rozlišením a jejich rozdíl je použit na uspání vlákna po správnou dobu pomocí funkce std::this_thread::sleep_for(std::chrono::microseconds(sleepMicro)).

Změna ujeté vzdálenosti kol a relativní pozice Vedlejším efektem PID regulátoru, na základě informací o změně natočení kol získaných funkcí getChangeOfEncoders() v podobě dvojice $C = (C_l, C_r); C_l, C_r \in \mathcal{Z}$, je výpočet relativní pozice, známé také jako Dead reckoning. Proměnné C_l a C_r obsahují počet kroků enkodéru na levém a pravém kole od posledního volání funkce.

$$D = ((C_l/wt) * r, (C_r/wt) * r) \tag{4.4}$$

```
loopPidThread_ = std::move(
  std::thread(
    &MobDifferentialChassis::updateEncoders,
    this,5)
  );
// set thread higher priority and FIFO order
struct sched_param param;
param.__sched_priority = 90;
if(pthread_setschedparam(
  loopPidThread_.native_handle(),
  SCHED_FIFO,&param)){
    fprintf(stderr,
    "Warning: Failed to set priority and scheduler.\n");
}

if(mlockall(MCL_CURRENT)){
  fprintf(stderr,"Warning: Failed to lock memory.\n");
}
```

Tabulka 4.5: Nastavení priority a plánovače Linux vláknu.

Pro převod kroků enkodérů na vzdálenost v metrech je použita rovnice 4.4, kde $D = (D_l, D_r); D_l, D_r \in \mathcal{R}$ představuje vzdálenost ujetou na obou kolech v metrech jako dvojici, wt je počet kroků enkodérů na jednu otáčku kola a r poloměr kola v metrech. Z vypočítané změny vzdálenosti jednotlivých kol je určena aktuální rychlost kol, která spolu s požadovanou rychlostí kol tvoří regulační odchylku $e(t)$ PID regulátoru a je upravena relativní pozice podvozku podle vzorců 3.26, 3.27 a 3.28 pro diferenciální podvozek.

4.7 RangeFinder

Pro vyčítání dat a komunikaci s laserovými dálkoměry firmy Sick je použit ROS uzel sick_tim3xx z databáze ROS uzlů. Uzel čte data z dálkoměru a poskytuje je jako zprávy, obsahující naměřené vzdálenosti v daném rozsahu, maximální a minimální dosah, počáteční a konečný úhel měření, úhlovou vzdálenost mezi jednotlivými měřeními a časovou značku, kdy bylo měření provedeno.

5. Testovací robotická platforma

V rámci diplomové práce vznikla nová robotická platforma přestavbou školní robotické platformy, postavené na robotickém podvozku MOB-2. Podvozek MOB-2 je diferenciálního typu, s opěrným všesměrovým kolem. Kola pohání 12V stejnosměrné motory s převodovkami a kvadraturními enkodéry. Mezi motory se vejdou dvě 12V olověné baterie o celkové kapacitě 4.2Ah.

Z původního robota se převzal základní podvozek MOB-2 s novými enkodéry a výkonová část motorů. Řídící jednotka byla nahrazena výkonnější. Výměna řídící jednotky znamenala vytvoření nového hw modulu pro komunikaci s enkodéry umístěnými na kolech robota. Současně bylo přepracováno napájení jednotlivých modulů. Na základě toho vznikla nová napájecí deska s odpojovačem, chránící baterie před přílišným vybitím.

Komunikace s řídící jednotkou je možná buď přes ethernetový kabel nebo při připojení wifi adaptéru přes bezdrátovou síť.

Nové hw moduly a rozmístění vyústilo ke změně úchytných desek. Desky byli vyřezány z překližky a plexiskla. Robot byl obohacen o laserový dálkoměr, jehož úchyt byl vytisknut na 3D tiskárně. Podkladové soubory k výrobě dekodérové desky, napájecí desky, úchytných desek a úchytu dálkoměru jsou k nalezení na přiloženém cd.

Vzniklá robotická platforma je hlavně myšlena pro další akademické použití, jako je výuka a testování vznikajících nových řešení robotických problémů. Proto byl kladen důraz na jednoduchost zvolených řešení a modulární přístup.

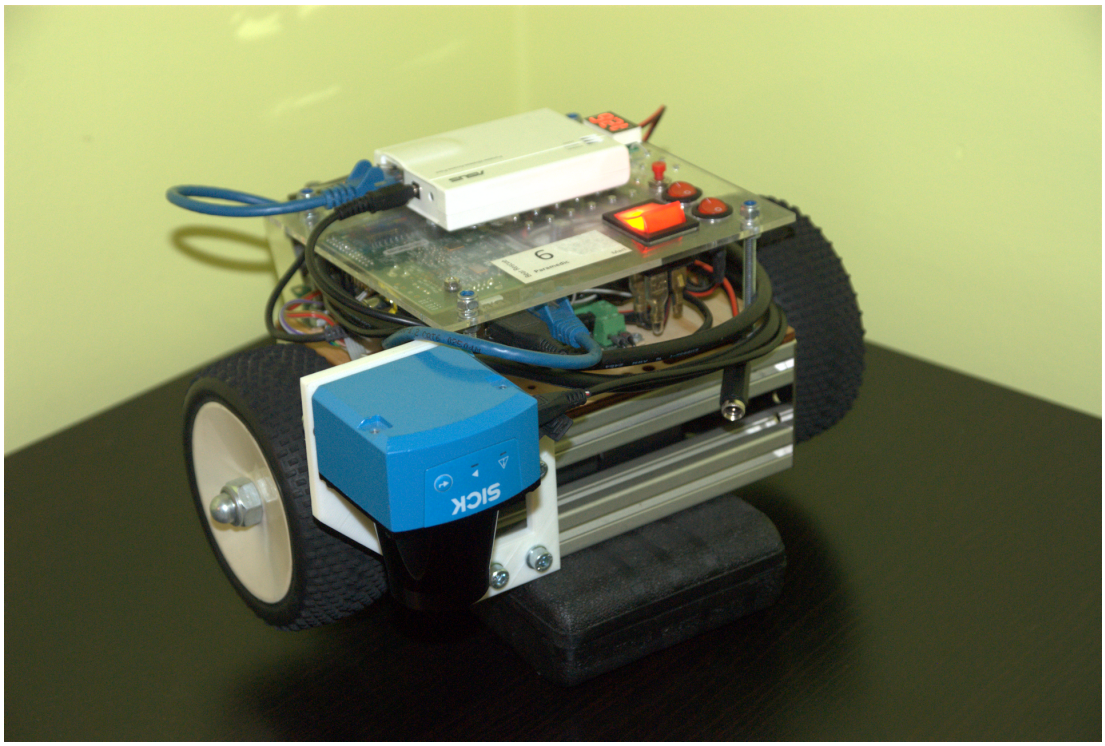
Výsledná robotická platforma je znázorněna na obrázku 5.1.

5.1 Architektura robotické platformy

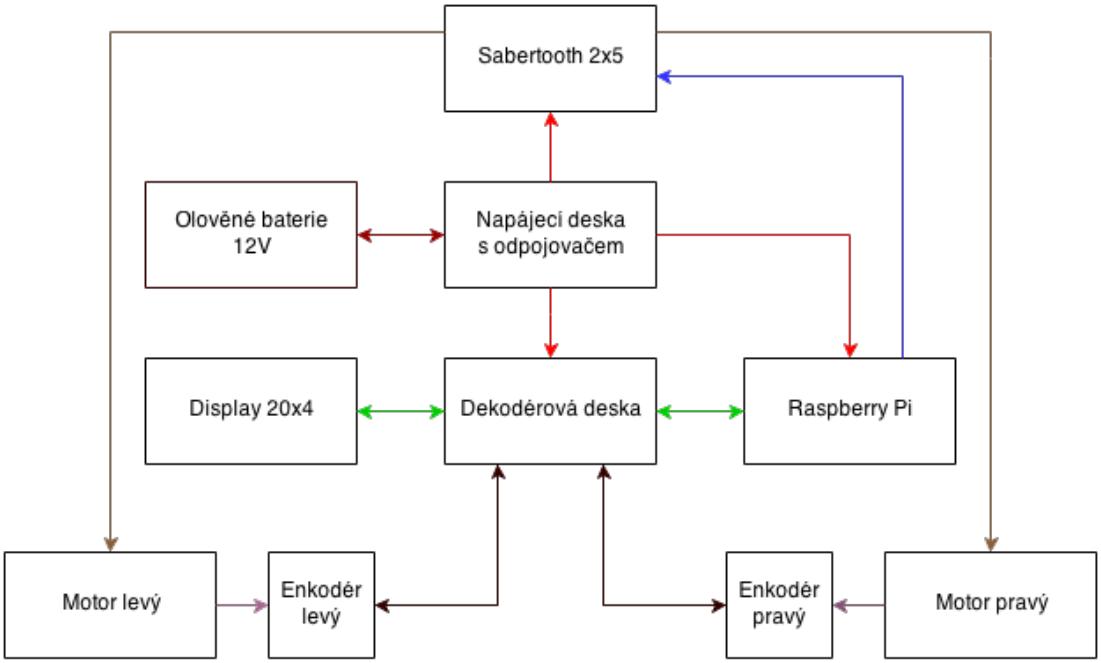
Základ tvoří řídící jednotka, mini počítač na jedné desce RaspberryPi [11]. Řídící jednotka je propojena sériovou linkou s výkonovým modulem Sabertooth 2x5 řídící plnění a směr otáčení motorů 5.1.2. Zpětná vazba o rychlosti otáčení kol, dekódováním informací z kvadraturních enkodérů 5.1.4, se získává z modulu dekodéru po sběrnici I2C, ovládané řídící jednotkou. Na sběrnici I2C je také zapojen alfanumerický displej 20x4 znaků.

Všechny moduly jsou napájené ze dvou 12V olověných baterií přes napájecí desku. Napájecí deska je chráněna proti zkratu odpojovací tavnou pojistkou. O ochranu baterií před podvybitím se stará odpojovací část, která při poklesu napětí na bateriích pod 10.6V odpojí baterie od zátěže. K napájecí desce je možné připojit moduly, vyžadující 12V nebo 5V napájení. Na 12V větev je zapojena výkonová část motorů. Všechny ostatní moduly jsou připojeny k 5V větvi, vytvořené spínaným zdrojem z 12V větve.

Celé propojení všech modulů je zakresleno na obrázku 5.2. Šipky znázorňují směr komunikace a napájení.



Obrázek 5.1: Fotografie testovací robotické platformy.



Obrázek 5.2: Propojení jednotlivých modulů vzniklé robotické platformy.

5.1.1 Řídící jednotka

Hlavní řídící a výpočetní jednotkou bylo zvoleno Raspberry Pi pro svojí malou velikost, poměr výpočetního výkonu a spotřeby, možnost komunikovat s dalšími moduly prostřednictvím I2C a sériové linky, nízkou pořizovací cenu a dobrou komunitní podporu.

Okolo Raspberry Pi je velká komunita uživatelů a vývojářů, což zajišťuje jistotu aktualizací optimalizovaných pro tento hw a vlastní předkompilovaný klon Linuxu nazvaný Raspbian. Distribuovaný Raspbian má linuxové jádro s aplikovanými PREEMPT úpravami. Tyto úpravy zajišťují rozumné chování celého linuxového jádra pro robotické účely.

Na internetu je k nalezení nepřeberné množství nejrůznější návodů na připojení různorodé elektroniky k Raspberry Pi. Takováto podpora je důležitá pro lidi bez elektronických znalostí, usnadňující vývoj.

5.1.2 Regulování výkonu motorů

O regulování výkonu motorů se stará zakoupený samostatný modul Sabertooth 2x5. Modul je přímo připojen k motorům kol a pulzně modulovaným signálem určuje plnění jednotlivých motorů. Při vysokých proudových špičkách se automaticky vypne a zabrání tím svému zničení. Dlouhodobě může poskytovat až 5A na motor. Na krátkou dobu dokáže vydržet proud 10A na motor.

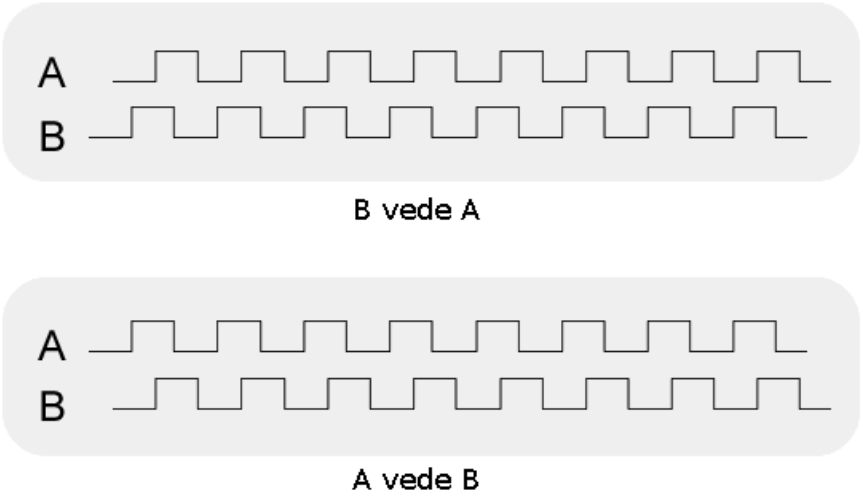
Řídící jednotka s modulem komunikuje jednosměrně po sériové lince, zjednodušeným sériovým protokolem. Zjednodušený sériový protokol používá jeden bajt pro určení směru otáčení a plnění motoru. Jelikož Sabertooth 2x5 ovládá dva motory, tak bajt v rozsahu 1-127 ovládá první motor, kde 1 je plné plnění zpět, 64 zastavení motoru a 127 plné plnění dopředu. Rozsah 128-255 ovládá motor druhý, kde 128 je plné plnění zpět, 192 zastavení motoru a 255 plné plnění dopředu. Bajt 0 má speciální význam vypnutí obou dvou motorů.

5.1.3 Kvadraturní enkodéry

Na oba motory kol jsou na zadní osičku přidělány permanentní magnety snímané čipem AS5040 (kvadraturní enkodér). Otáčení magnetu mění čipem snímané magnetické pole, na základě čehož čip generuje dva navzájem posunuté signály, zobrazené na obrázku 5.3. Změna signálu odpovídá otáčení kol a jedno úplné otočení magnetu na ose motoru vydá 1024 změn. Motory mají namontované převodovky 29:1, rozlišení enkodérů je tedy 0.01 stupně otočení kola. Směr otáčení kola se určuje podle vzájemného posunu signálu, jestli vede signál A nebo B.

5.1.4 Dekodér signálu z kvadraturních enkodérů

Signál z kvadraturních enkodérů zpracovává samostatný hw modul, založený na mikrokontroleru Atmel ATmega48, do dvou 16bitových čítačů. Čítače jsou inkrementovány nebo dekrementovány dle směru a změny otočení kol. Otáčení kol mění úrovně výstupních signálu z enkodérů. Tyto změny úrovní signálu vyvolávají externí přerušení na vstupních pinech mikrokontroleru. Jednotlivá přerušení jsou zpracována obslužnými funkcemi, implementující náhledovou tabulku 5.1, určující inkrementaci nebo dekrementaci adekvátního čítače.



Obrázek 5.3: Ukázka signálu z kvadraturního enkodéru.

	Aktuální stav	
Minulý stav	Inkrementace čítače	Dekrementace čítače
0	1	2
1	3	0
2	0	3
3	2	1

Tabulka 5.1: Závislost mezi změnou signálu z kvadraturních enkodérů a inkrementací nebo dekrementací čítače. Signál z enkodérů je převeden na 2bitovou hodnotu, kde spodní bit reprezentuje úroveň signálu B a horní bit úroveň signálu A.

S dekodérem komunikuje řídící jednotka po sběrnici I2C vlastním komunikačním protokolem. Řídící jednotka vždy inicializuje komunikaci posláním jednoho bajtu, obsahujícího typ požadované odpovědi. Hodnota 10 odpovídá požadavku na vrácení aktuálních hodnot obou čítačů. Odpověď dekodéru je pět bajtů dlouhá a obsahuje postupně horní a dolní bajt čítače levého kola, horní a dolní bajt čítače pravého kola a poslední bajt obsahuje kontrolní součet předchozích čtyř bajtů. Kontrolní součet je přidán pro ověření validnosti přenesených dat řídící jednotkou. Pokud kontrolní součet přijatých dat řídící jednotkou není nulový, tak došlo během přenosu k chybě a musí se opakovat.

Výpočet kontrolního součtu

Výpočet je založen na cyklickém redundantním součtu CRC, běžně používaným k detekci chyb během přenosu dat. Dobré matematické vlastnosti a efektivní implementace mu umožnili se stát velmi rozšířeným způsobem realizace kontrolního součtu. Dekodér používá 8bitový mikrokontroler, na základě čehož byl vybrán 8-bitový CRC polynom $x^8 + x^5 + x^4 + 1$ s iniciální hodnotou součtu 0. Výpočet kontrolního součtu s tímto polynomem je již obsažen v AVR Libc knihovnách [42] pro zmiňovaný mikrokontroler.

5.1.5 Laserový dálkoměr

Na robota byl připevněn laserový dálkoměr firmy SICK Tim310. Komunikace je řešena přes standardní USB kabel, připojený k řídící jednotce. Dálkoměr vrací informaci o vzdálenostech v rozsahu 270 stupňů po jednom stupni, s maximálním dosahem 4m a frekvencí měření 15Hz.

5.2 Srovnání s jinými robotickými platformami

Vzniklou robotickou platformu je možné doplňovat o další senzory a příslušenství obdobně jako jiné komerční roboty, kupříkladu Pioneer P3-DX. Platforma se hlavně hodí na vnitřní použití, s možností venkovní jízdy po zpevněných cestách, chodnících nebo cestách v parku. Nízká výška podvozku od země neumožňuje nasazení v prostředích s nerovným povrchem, s velkými změnami výšky podloží na malé ploše, kde jsou lepší tankové podvozky, jako od firmy iRobot. Stejně tak není nijak chráněn proti dešti či prachu, což pro robota se zaměřením na vnitřní použití a akademické účely není podstatné kritérium.

Robot je schopný uvést ještě další až cca 4kg v podobě notebooku, senzorů a jiného vybavení. Na rozdíl od malých robotů, postavených na Arduino deskách, jako je BoeBot, který uvezou přídavnou zátěž v desetinách kilogramu.

Hlavní řídící jednotka je dostatečně dimenzovaná pro řešení velké škály problémů, ale není tak výkoná, jako u jiných robotů. Dodatečný výkon může poskytnou notebook, položený na robota, připojený pomocí ethernetového kabelu nebo wi-fi. Případná výměna řídící jednotky za výkonnější není problém, jelikož veškerá komunikace s ostatními moduly probíhá přes standardní komunikační prostředky a ovládací software je psán tak, aby ho bylo možné přenést na libovolnou řídící jednotku s Linuxem, opatřenou I2C a sériovou linkou.

Celková doba provozu robota na baterie se při běžném používání pohybuje okolo 2 hodin. Tato doba je velmi závislá na připojených senzorech a rychlosti jízdy robota. Dala by se prodloužit výměnou olověných baterií za Li-pol baterie s vyšší kapacitou a nižší váhou. Olověné baterie byli zvoleny pro snadnou údržbu, nízkou cenu a bezpečnost proti požáru nebo výbuchu.

Závěr

V této práci je popsána přestavba školní robotické platformy postavené na podvozku MOB-2 a řídící systém pro tuto platformu, ovládající hardware robota tak, aby projel uživatelem definovanými body s vysokou přesností, bez kolizí s překážkami.

Vzniklá nová školní platforma je navrhnuta tak, že má dostatečný výpočetní výkon a schopnosti, aby jí bylo možné použít pro běžnou výuku základů mobilní robotiky. Zároveň jí lze rozšiřovat o nové druhy senzorů, jako například videokamera, senzory osvětlení nebo mikrofony: k řídící jednotce je možné připojit tyto senzory například přes USB sběrnici, I2C nebo prostřednictvím různých protokolů na volné vstupně/výstupní piny řídící jednotky. Pokud by bylo v budoucnosti nutné vyměnit řídící jednotku za výkonnější, je to možné víceméně beze změn, pokud na nové řídící jednotce bude fungovat Linux a bude moci komunikovat s ostatními hardwarovými moduly po sériové lince a I2C sběrnici. Celý robot se dá snadno rozebrat na jednotlivé díly a znovu složit, při potřebě opravy nebo výměny některého z hardwarových modulů nebo pro demonstraci zapojení v rámci výuky.

Robot je pomocí nového řídícího systému schopen plynulou jízdou projet uživatelem definovanými cílovými body bez kolizí s překážkami a průběžně vracet mapu neznámého prostředí. Řídící systém správně reaguje i na nečekané změny prostředí, jako je umístění nové překážky před robota.

Moduly řídícího systému byly rozděleny do dvou skupin, kde každá používá jiný souřadnicový systém, až na řídící modul, který využívá oba souřadnicové systémy. To se ukázalo jako správné řešení, neboť to významně zjednodušuje celou implementaci. Řídící modul je schopný prostřednictvím transformačních matic, převádějících body z jednoho souřadnicového systému do druhého, průběžně kompenzovat nashromážděnou chybu relativní polohy.

S tím souvisí dynamické předávání části cesty, kdy modul ovládající jízdu robota má předanou pouze část cesty a zbylou část zatím nezná. Postupné předávání cesty umožňuje dynamicky reagovat na nové informace o prostředí ze senzorů. Řídící systém může dynamicky měnit nepředanou část cesty tak, že to vůbec neovlivní současný pohyb robota.

Pomocí sdílení stejného souřadnicového systému mezi moduly ovládajícími jízdu a hardware robota bylo možné dosáhnou vysoké přesnosti a plynulosti sledování hermitovské křivky. Robot proložením hermitovské křivky naplánovanými body zvládá danými body projet plynule a tedy častokrát i rychleji než při samostatném otáčení na místě a pohybu přímo.

Volba nasazení operačního systému Linux s přerušitelným jádrem na řídící jednotku a důraz na přenositelnost během implementace umožňují vzniklý řídící systém bez žádných změn (případně pouze s minimálními změnami) nasadit na libovolného robota disponujícího řídící jednotkou s Linuxem a odpovídajícími hardwarovými moduly. Moduly řídícího systému, ovládající jízdu a hardware robota, byly samostatně úspěšně použity a odladěny na jiném diferenciálním robotu, použitém na soutěži SICK robot day 2014. Nasazení na jiného robota pomohlo nalézt a odladit chyby v implementaci modulů, které se s přestavěnou školní platformou neprojevovaly.

Systém v současné podobě splňuje požadavky, které na něj byly kladené. Do budoucna je samozřejmě možné vzniklý řídící systém rozšiřovat například o následující změny:

- Vytvořit vlastní rozhodovací modul, určující cílové body pro prozkoumání neznámého prostředí. Robot by poté mohl sloužit jako průzkumník;
- Umožnit zvětšení velikosti mapy používané moduly, případně moduly pracující s mapou zaměnit za moduly implementující jiné sofistikovanější algoritmy;
- Lokalizační modul by bylo možné snadno rozšířit o další typy senzorů, na základě implementačního rozhodnutí použít návrhový vzor visitor na částice lokalizace;
- Robot v současné implementaci projíždí čelem vloženými průjezdními body. Bylo by zajímavé dovolit definovat průjezdním bodům, zda jimi robot projede čelně nebo pozpátku.

Seznam použité literatury

[1] FreeRTOS: *homepage* [online]. [cit. 2014-01-8]. Dostupné z: http://www.freertos.org/

[2] Microsoft .NET Gadgeteer: *homepage* [online]. [cit. 2014-01-20]. Dostupné z: http://www.netmf.com/gadgeteer/

[3] Arduino: *homepage* [online]. [cit. 2014-01-20]. Dostupné z: http://arduino.cc/

[4] Atmel: *microcontrollers* [online]. [cit. 2014-01-20]. Dostupné z: http://www.atmel.com/products/microcontrollers/

[5] Pixace: *homepage* [online]. [cit. 2014-01-20]. Dostupné z: http://www.picaxe.com/

[6] ST: *STM32 MCUs* [online]. [cit. 2014-01-20]. Dostupné z: http://www.st.com/web/en/catalog/mmc/FM141/SC1169

[7] Tinkerforge: *homepage* [online]. [cit. 2014-01-20]. Dostupné z: http://www.tinkerforge.com/en

[8] BeagleBone: *Products,BeagleBone* [online]. [cit. 2014-01-25]. Dostupné z: http://beagleboard.org/Products/BeagleBone

[9] Gumstix: *homepage* [online]. [cit. 2014-01-25]. Dostupné z: https://www.gumstix.com/

[10] Intel: *support,galileo* [online]. [cit. 2014-01-25]. Dostupné z: http://www.intel.com/support/galileo/index.htm

[11] Rasberry PI: *homepage* [online]. [cit. 2014-01-25]. Dostupné z: http://www.raspberrypi.org/

[12] National Instruments: *LabView* [online]. [cit. 2014-01-25]. Dostupné z: http://sine.ni.com/np/app/main/p/docid/nav-104/lang/cs/

[13] MathWorks: *MATLAB* [online]. [cit. 2014-01-25]. Dostupné z: http://www.mathworks.com/products/matlab/

[14] MathWorks: *Simulink* [online]. [cit. 2014-01-25]. Dostupné z: http://www.mathworks.com/products/simulink/

[15] National Instruments: *myRIO* [online]. [cit. 2014-01-26]. Dostupné z: http://sine.ni.com/np/app/main/p/ap/academic/lang/cs/pg/1/sn/n17:academic,n21:18368/

[16] Terasic: *SoCKit* [online]. [cit. 2014-01-26]. Dostupné z: http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=167&No=816

[17] Fischer technik: *homepage* [online]. [cit. 2014-01-26]. Dostupné z: http://www.fischertechnik.de/

[18] Lego: *Mindstorms* [online]. [cit. 2014-01-26]. Dostupné z: http://www.lego.com/en-us/mindstorms/

[19] Vex Robotics: *homepage* [online]. [cit. 2014-01-26]. Dostupné z: http://www.vexrobotics.com/

[20] ABBOTT, Doug. *Linux for embedded and real-time appplications* [online]. 2nd ed. Burlington, MA: Newnes, c2006, xviii, 300 p.

[21] ADEOS: *homepage* [online]. [cit. 2014-02-19]. Dostupné z: http://home.gna.org/adeos/

[22] RTAI: *homepage* [online]. [cit. 2014-02-20]. Dostupné z: https://www.rtai.org/

[23] Xenomai: *homepage* [online]. [cit. 2014-02-21]. Dostupné z: https://www.xenomai.org/

[24] EtherCat: *homepage* [online]. [cit. 2014-02-27]. Dostupné z: http://www.ethercat.org/

[25] tenAsys: *homepage* [online]. [cit. 2014-02-27]. Dostupné z: http://www.tenasys.com/

[26] ItervalZero: *homepage* [online]. [cit. 2014-02-27]. Dostupné z: http://www.intervalzero.com/

[27] ROBOTC: *homepage* [online]. [cit. 2014-02-28]. Dostupné z: http://www.robotc.net/

[28] ROS: *homepage* [online]. [cit. 2014-02-28]. Dostupné z: http://ros.org/

[29] URBI: *homepage* [online]. [cit. 2014-02-28]. Dostupné z: http://www.urbiforge.org/

[30] Lenovo: *x240* [online]. [cit. 2014-04-3]. Dostupné z: http://shop.lenovo.com/us/en/laptops/thinkpad/x-series/x240/

[31] Dell: *latitude 3330* [online]. [cit. 2014-04-3]. Dostupné z: http://www.dell.com/

[32] Google: *Nexus 5* [online]. [cit. 2014-11-14]. Dostupné z: http://www.google.com/nexus/5/

[33] Evga: *Tegra NOTE 7* [online]. [cit. 2014-11-14]. Dostupné z: http://www.evga.com/Products/Product.aspx?pn=016-TN-0701-B1

[34] NVIDIA: *Shield Tablet* [online]. [cit. 2014-11-14]. Dostupné z: http://shield.nvidia.com/gaming-tablet/

[35] WINDOWS EMBEDDED COMPACT 2013: [online]. [cit. 2014-09-29] Dostupné z: http://www.microsoft.com/windowsembedded/en-us/windows-embedded-compact-2013.aspx

[36] WINDOWS EMBEDDED AUTOMOTIVE 7: [online]. [cit. 2014-09-29] Dostupné z: http://www.microsoft.com/windowsembedded/en-us/windows-embedded-automotive-7.aspx

[37] NISE, Norman S. *Control systems engineering*. New York: John Wiley, 2004, 983 s. ISBN 04-714-4577-0.

[38] DIMENSIONENGINEERING: *sabertooth 2x5* [online]. [cit. 2014-10-14]. Dostupné z: http://www.dimensionengineering.com/products/sabertooth2x5

[39] LINUX: *pthread_ getschedparam* [online]. [cit. 2014-10-15]. Dostupné z: http://man7.org/linux/man-pages/man3/pthread_getschedparam.3.html

[40] ŽÁRA, Jiří, Bedřich BENEŠ, Jiří SOCHOR a Petr FELKEL. *Moderní počítačová grafika. Vyd 1.* Brno: Computer Press, 2004, 609 s. ISBN 80-251-0454-0.

[41] THRUN, Sebastian. *Probabilistic robotics*. Massachusetts: MIT Press, c2006, xx, 647 s. ISBN 02-622-0162-3.

[42] AVR LIBC: *homepage* [online]. [cit. 2014-11-12]. Dostupné z: http://www.nongnu.org/avr-libc/

Seznam tabulek

Seznam použitých zkratek

hw platformy - hardwarové platformy
RTOS (real-time operating system) - real-time operační systémy
SoC (system on a chip) - systém na chipu. Integrovaný čip, který obsahuje všechny komponenty počítače.
FPGA (field-programmable gate array) - programovatelné hradlové pole
NI - firma National Instruments
RTX - realtime rozšíření
RTSS (real-time subsystem) - realtime subsystém
POMDP (partially observable Markov decision process) - částečně pozorovatelné markovské rozhodovací procesy
PRM (probabilistic roadmap) - pravděpodobnostní silniční mapy
SLAM (simultaneous localization and mapping) - simultánní lokalizace a mapování
EKF (extended Kalman filter) - rozšířený kalmanův filtr
SEIF (sparse extended information filter) - řídký rozšířený informační filtr
STL (standard template library) - standardní šablonová knihovna pro C++
CRC (cyclic redundancy check) - cyklický redundatní součet

Přílohy

Appendix 1 - Obsah DVD

Na přiloženém DVD se nalézají:

- tato práce ve formátu pdf
- uživatelská a vývojová dokumentace projektu
- dokumentace napájecí a dekodérové desky
- zdrojový kód projektu
- obrázky použité v této práci
- podkladové soubory pro stavbu vzniklé robotické platformy
- obraz 16GB SD karty robota, obsahující Linux Raspbian s nainstalovaným ROS, řídícím systémem a zdrojovými soubory