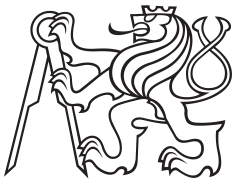
Bakalářská práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra kybernetiky

# Bezkolizní řízení pro robotickou helikoptéru

Michal Bahník

Vedoucí práce: Ing. Jan Chudoba
Obor: Kybernetika a robotika
Květen 2018

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Bahník** Jméno: **Michal** Osobní číslo: **434824**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra kybernetiky**

Studijní program: **Kybernetika a robotika**

Studijní obor: **Robotika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Bezkolizní řízení pro robotickou helikoptéru**

Název bakalářské práce anglicky:

**Collision Avoidance for a Multirotor UAV**

Pokyny pro vypracování:

Cílem práce je návrh systému umožňujícího bezkolizní pohyb malé robotické helikoptéry v prostředí s překážkami.
1. Vyhledejte relevantní zdroje a seznamte se s metodami pro bezkolizní navigaci bezpilotních helikoptér se zaměřením na operaci v omezeném prostoru a vypracujte rešerši popisující používané metody.
2. Navrhněte metodu mapování překážek v okolním prostředí vhodnou pro malý model vícerotorové helikoptéry s cílem bezkolizního průletu do zadané cílové pozice ve zvoleném souřadném systému. Po konzultaci s vedoucím práce zvolte vhodné omezující podmínky tohoto složitého problému s ohledem na řešitelnost úlohy.
3. S využitím vhodného simulačního nástroje vytvořte modelové prostředí pro simulaci chování helikoptéry, umožňující ověření funkce navržené navigační metody.
4. Navrženou navigační metodu implementujte, ověřte její funkci experimentálně na modelu a vyhodnoťte její chování v závislosti na struktuře prostředí.

Seznam doporučené literatury:

[1] Thrun, S., Burgard, W. and Fox, D., Probabilistic Robotics. Cambridge, Mass: MIT Press. 2005.
[2] Roland Siegwart, Illah Reza Nourbakhsh and Davide Scaramuzza, Introduction to Autonomous Mobile Robots, MIT Press, 2011.
[3] Tomáš Báča - Řízení vzájemně lokalizovaných bezpilotních helikoptér, diplomová práce - Praha, 2013.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Jan Chudoba, inteligentní a mobilní robotika CIIRC**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **12.02.2018** Termín odevzdání bakalářské práce: **25.05.2018**

Platnost zadání bakalářské práce: **30.09.2019**

| Ing. Jan Chudoba | doc. Ing. Tomáš Svoboda, Ph.D. | prof. Ing. Pavel Ripka, CSc. |
|---|---|---|
| podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | podpis děkana(ky) |

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. Datum převzetí zadání

Podpis studenta

# Poděkování

Tímto děkuji svému vedoucímu panu Ing. Janu Chudobovi za jeho ochotný přístup a podnětné rady, které vedly k dokončení této práce.


# Prohlášení

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

.....

V Praze dne 25. května 2018

# Abstrakt

Práce se zabývá návrhem obecného prostředí pro vývoj a testování řídicích systémů bezpilotních helikoptér. Součástí tohoto prostředí je i nástroj pro simulaci modelových prostředí a situací. Nastíněna je problematika detekce překážek a předcházení kolizí těchto strojů. Dále je navrženo a implementováno konkrétní řešení bezkolizního řízení pro obecnou bezpilotní helikoptéru. Součástí práce je i testování navrženého řešení na modelových situacích v simulátoru.



**Klíčová slova:** UAV, multikoptéra, detekce překážek, předcházení kolizím



**Vedoucí práce:** Ing. Jan Chudoba
České vysoké učení technické v Praze
Český institut informatiky, robotiky a kybernetiky,
Jugoslávských partyzánů 1580/3,
160 00 Praha 6, Dejvice



# Abstract

The thesis deals with the design of a general environment for the development and testing of unmanned helicopter control systems. A tool for simulating model environments and situations is also part of this environment. The issue of obstacle detection and prevention of collisions of these machines is highlighted. In addition, a specific solution for collision control for a general unmanned helicopter is proposed and implemented. Part of the thesis is the testing of the proposed solution in model situations in the simulator.



**Keywords:** UAV, multirotor, obstacle detection, collision avoidance

**Title translation:** Collision avoidance for a multirotor UAV

# Obsah

# Obrázky



# Tabulky

# Kapitola 1
# Úvod

V posledních letech výrazně stoupá obliba bezpilotních letadel, především pak multikoptér. Dokazuje to strmě stoupající počet prodaných bezpilotních helikoptér[1]. Tento trend lze pozorovat v různých oblastnech lidské činnosti. V komerční sféře jsou helikoptéry nasazovány například k inspekcím nedostupných či rozlehlých míst (kontrola materiálové integrity větrných elektráren, nepřístupných rozvodů, ...) nebo profesionální pořizování fotografií a videí. Armáda pak bezpilotní stroje nasazuje na průzkumné i bojové mise. Ve výzkumné sféře se helikoptéry těší oblibě při různých průzkumech a mapováních. Obliba multikoptér také výrazně stoupla v oblasti rekreační, kde prim hrají především fotografie a pořizování videa a FPV závody. Konkrétních činností, ke kterým se helikoptéry využívají, je nepřeberné množství a další řada jich je ve fázi vývoje.

Obliba multikoptér (zvláště pak kvadrokoptér, viz 1.2) je dána několika faktory. Hlavní výhodou je však jednoduchá konstrukce a (díky levné a výkonné elektronice) snadné řízení. Oproti většině ostatních letounů jsou také helikoptéry méně náročné na operátorovy schopnosti a zkušenosti. V některých aplikacích je také nezanedbatelnou výhodou schopnost udržovat výšku a pozici nebo symetrická schopnost pohybu do všech stran.

V případě havárie stroje však může dojít k újmě na zdraví či majetku. Riziko havárie pak roste (nejen) s počtem užívaných letounů. S ohledem na tuto skutečnost je tedy žádoucí zajistit jejich bezpečný provoz.

Jedním z důvodů havárie je kolize s okolními objekty. V případě stroje dálkově ovládaného pilotem sice protikolizní systém pro provoz nezbytný není, výrazně však může omezit riziko kolize. Pro autonomní užití bezpilotních strojů je však bezkolizní řízení esenciální.

V současné době jsou některé helikoptéry vybaveny systémy pro detekci překážek s různou mírou spolehlivosti. Podobný problem také řeší některé automobilky při vývoji autonomních vozidel. Přestože je vývoj v této oblasti intenzivní a dosahuje zajímavých pokroků (autonomní vozidla jsou již testována v běžném provozu), žádné z řešení však v době vzniku této práce nepřináší dostatečnou spolehlivost pro bezpečný ryze autonomní provoz.

## 1.1 Vymezení problému a předpokladů

Tato práce si dává za cíl navrhnout nízkonákladové bezkolizní řízení pro bezpilotní helikoptéru v omezeném prostoru s překážkami. Modelovou situací může být pohyb ve vnitřních prostorách budov. Po konzultaci s vedoucím práce a s ohledem na náročnost problému byly přijaty některé zjednodušující předpoklady, ačkoliv návrh počítá s možností pozdějšího rozšíření navrženého řešení.

Bezkolizní řídicí systém předpokládá statické uzavřené prostředí a multikoptéru udržující konstantní výšku. Řízení a detekce překážek tedy bude pouze ve dvou dimenzích (horizontální rovina). Uvažovaná helikoptéra pak musí být schopna symetrické manévrovací schopnosti v horizontální rovině (typická vlastnost multikoptér). Jelikož jsou v uzavřených prostorech často omezené či žádné možnosti komunikace s okolním světem (mobilní sítě, GPS, ...), návrh je nevyužívá a není na nich závislý. Stejně tak se předpokládá, že není k dispozici externí výpočetní výkon (tzv. *groundstation*).

Vývoj takového systému může vyžadovat provedení velkého množství testů na skutečné helikoptéře, což může být z různých důvodů (legislativa, nedostatek strojů, velké množství testů, ...) komplikované. Z tohoto důvodu je součástí tohoto návrhu i propojení bezkolizního řízení s počítačovým simulátorem, ve kterém bude možno testy provádět.

## 1.2 Terminologie

Zvláště média a veřejnost někdy používají nepřesné, nejasné či zbytečně obecné termíny označující bezpilotní letouny a jejich podskupiny. Zde jsou uvedeny některé termíny a jejich stručné definice.

Pro jakýkoliv bezpilotní létající prostředek je zejména v anglicky psaném textu hojně používána zkratka **UAV** (z anglického **U**nmanned **A**erial **Vehicle**). Termín zahrnuje stroje řízené autopilotem i dálkově lidským operátorem.

V podobném kontextu je pak užívá i termín **dron**. Jeho definice však není jednotná a v této práci nebude užit.

Podskupinou bezpilotních letadel jsou **helikoptéry**, tedy letouny s horizontálními rotory. **Multikoptéry** se pak vymezují počtem rotorů (více než dva). Dle tohoto kritéria rozlišujeme například **kvadrokoptéry**, **hexakoptéry**, **oktakoptéry** a podobně.

# Kapitola 2
# Současný stav problematiky

V době psaní této práce vniklo množství více či méně komplexních systémů bezkolizního řízení. Jejich vývoj se však neustále pohybuje kupředu a tyto systémy se stávají čím dál tím spolehlivější. Nejvýraznějším příkladem jsou nepochybně autonomní vozidla. Několik výrobců automobilů již testuje autonomní vozidla v běžném provozu, v některých domácnostech narazíme na robotický vysavač a kvadrokoptéry s (do jisté míry) bezkolizním řízením jsou dostupné ve volném prodeji. Ačkoliv již tedy takové systémy existují a uvádějí se do praxe, stále ještě nedosahují požadované spolehlivosti.

Pro komplexní pohled na problematiku budou zmíněny i jiné pohyblivé stroje, které řeší detekce překážek a následné předcházení kolizí, neboť přístupy k detekci jsou obdobné a do velké míry přenositelné.

## 2.1 Autonomní vozidla

U autonomních automobilů, kde jsou extrémně vysoké nároky na spolehlivost (kvůli potenciální značné újmě na zdraví či majetku), se očekává, že detekce překážek a předcházení kolizím bude vysoce robustní. Některé návrhy a principy pak mohou být aplikovány i v systémech pro jiné stroje.

První vážné pokusy o autonomní řízení automobilu sice probíhaly již v 80. letech [2] minulého století, k testování v běžném provozu však došlo až v druhém tisíciletí. Počátkem druhého desetiletí pak již většina velkých automobilových společností buď vyvíjela, nebo dokonce testovala autonomní automobily.

Při podrobnějším zkoumání vyjde najevo skutečnost, že různé automobilky přistupují k problému často odlišným způsobem. Ačkoliv snad všechny systémy používají ultrazvukové dálkoměry a kamery, u lidarů (laserových scannerů) je situace opačná. Zatímco pro některé systémy jsou lidary nejdůležitějším sensorem (například Waymo od Google [3]), jiné nepoužívají žádné [4]. Přesto oba přístupy dosahují značné spolehlivosti. Některé systémy využivají i klasický radar.

Vzhledem ke komplexnosti situací, ve kterých se autonomní automobily mohou ocitnout, není překvapující počet i rozmanitost sensorů. Odpovídá tomu i nutnost značné výpočetní kapacity a pokročilých algoritmů.

## 2.2 Multikoptéry

Na poli volně prodejných dronů jasně kraluje společnost DJI. Vyšší řada jejich kvadrokoptér je vybavena sonary a optickými sensory, díky kterým je schopna detekovat překážky až v pěti směrech (vpředu, vzadu, vlevo, vpravo a dole). Kromě kvadrokoptér schopných detekovat překážky a vyhýbat se jim nabízí i modul Guidance[5]. Jedná se o integrovaný detekční systém založený na optických a zvukových sensorech pracující v pěti směrech. Tento modul lze použít na libovolném stroji a poskytuje kromě informací o překážkách i data z vlastní IMU[1]. Ostatně, sensory vytvářející hloubkové mapy používají i systémy ostatních výrobců. Zmiňme například Yuneec, který některé modely osadil přímo sensory navržené k tvorbě hloubkových map [6].

S autonomními helikoptérami také experimentuje firma Amazon, která s jejich pomocí plánuje doručovat zásilky. Přestože společnost zveřejnila několik propagačních videí, konkrétní informace o použitém hardwaru nebo technologiích nejsou dostupné.

Velmi zajímavý systém autonomního řízení představila společnost Skydio [7]. Na oficiálních stránkách výrobce uvádí, že stroj je osazen třinácti (!) monokamerami (umístěnými ve stereo konfiguracích), čtyřmi IMU, 256jádrovou grafickou kartou Nvidia TX1 a čtyřjádrovým ARM procesorem. Mapa okolního prostředí je generována SLAM[2] algoritmem na základě vstupů z videokamer. Použití sonarů, IR sensorů nebo laserových scannererů není zmíněno a tedy nejsou pravděpodobně vůbec použity. Přesto je kvadrokoptéra relativně spolehlivě schopna sledovat objekt i v prostředí s větším počtem překážek (např. les). Pozoruhodný je především fakt, že detekce překážek spoléhá čistě na obrazovou analýzu vizuálních dat a nepotřebuje žádné dálkoměry. Vzhledem k použitému hardwaru však zpracování vyžaduje vysokou výpočetní kapacitu a vysoce optimalizovaný kód.

## 2.3 Vědecké práce

Tématu bezkolizního řízení multikoptér se intenzivně věnují komerční společnosti (viz výše). Přesto existuje i řada akademických prací zabývajících se stejnou problematikou. Některé z prací, které mohou mít přínos pro návrh řešení v této práci jsou níže zmíněny a stručně popsány.

Z hlediska detekce překážek pomocí nízkonákladových senzorů stojí za povšimnutí práce [8]. Stroj je osazen pouze 12 ultrazvukovými a 16 IR sensory rovnoměrně rozmístěnými tak, aby rovnoměrně pokryly horizontální rovinu. Tento prostor kolem helikoptéry je pak rozdělen na 12 sektorů po 30° a odpovídá rozložení sonarů. IR sensory jsou také rozloženy rovnoměrně, avšak vždy ve dvojici (tedy 8 dvojic) dvou sensorů s rozdílnými (avšak alespoň částečně překrývajícími se) vzdálenostmi detekce.

[1] Angl. *Inertial Measurement Unit*, tedy inerciální měřicí jednotka. Měří (mimo jiné) lineární zrychlení a angulární rychlosti.

[2] Z angl. *Simultaneous Localization And Mapping*, tedy současné lokalizace a mapování okolí.

Na data ze sensorů je aplikován vážený filtr, jehož výstupem jsou vzdálenosti k nejbližší překážce v příslušném sektoru. Díky tomuto filtru jsou do jisté míry eliminována chybná měření způsobená vlastnostmi konkrétního sensoru. K vyhodnocování zásahu do řízení zavádí skupina tříúrovňový systém nebezpečí: nebezpečná vzdálenost, blízká vzdálenost a volný prostor[3].

Dále je implementován stavový automat, který rozlišuje pro každou osu čtyři situace: v ose je překážka v kladném směru, v ose je překážka v záporném směru, v ose je detekována překážka v kladném i záporném směru a žádná překážka není v ose detekována. O regulaci se pak stará upravený PID regulátor. Ten se snaží udržet minimální povolenou vzdálenost k překážce nebo shodné vzdálenosti k protilehlým překážkám.

Práce autorů Mendese a Ventury [9] pak kombinuje přístup reaktivního řízení a SLAM algoritmu ve 3D. Jelikož jejich cílem je předcházet kolizím a nikoli vytvářet přesné mapy, jejich algoritmus založený na FastSlamu je pro tento účel dostatečně rychlý. Helikoptéra zde je osazena 6 sonary, přičemž ve vodorovné rovině snímaly čtyři.

Výstupem SLAM algoritmu je trojrozměrná mřížka obsazenosti, ze které je později možné spočítat vzdálenost k překážce. Ke každé buňce je pak přizazen jeden ze stavů `F` (*free*), `O` (*occupied*) nebo `U` (*unknown*). Stav `U` je pouze pro inicializační hodnotu 0.5. Buňkám s vyšší hodnotou je pak přiřazen stav `O`, ostatním (tedy menším než 0.5) `F`.

Na rozdíl od [8] v této práci nejsou zavedeny nebezpečné vzdálenosti, ale systém se rozhoduje dle *času do kolize* (TTC, *Time To Collision*. Ten představuje dobu, za kterou by stroj narazil při současné rychlosti do překážky. Zavedeny pak jsou pro snižující se TTC postupně čtyři různé reakce: žádná (*No action*), zpomalení (*SLOW*), zastavení (*STOP*) a úhybný manévr (*Evasion Maneuver*).

V práci [10] se autoři rozhodli zkombinovat sonary, laserové scannerry a duální kamery. Dvě dvojice kamer s širokým ohniskem umístili do přední a zadní části helikoptéry, čímž pokryli většinu okolního prostoru. V spodní části je multikopétra osazena jedním 2D laserovým scannerrem, jehož rotací pomocí servomotorku je dosaženo pokrytí velké části prostoru pod helikoptérou. Po obvodu je pak rozmístěno 8 sonarů. Autoři jejich použití odůvodňují detekcí úzkých překážek (větve, dráty, ...), které scannerr nebo kamery nemusí spolehlivě detekovat.

Všechny zmíněné senzory pak slouží k tvorbě 3D mřížky obsazenosti, která pak slouží k předcházení kolizí a plánování trasy. Stav helikoptéry je pak určen pomocí rozšířeného Kalmanova filtru[4] fúzí dalších sensorů (IMU, barometr, senzor optického toku). Ačkoliv senzory a následné zpracování dat jsou rozebrány podrobně, řízení však v této práci popsáno prakticky není.

[3] V originále *Danger Area*, *Near Area* a *Free Area*

[4] Angl. EKF - Extended Kalman Filter.

## 2.4 Shrnutí

Pokročilé systémy detekce překážek a předcházení kolizím vesměs používají hloubkové mapy tvořené komplexními sensory. Z hlediska stanovených požadavku a omezení v této práci je však použití těchto senzorů (především kvůli jejich ceně) omezené až nemožné. S ohledem na tuto skutečnost je žádoucí použít sonary a IR sensory jako hlavní sensory pro detekci překážek.

Většina systémů také implementuje nějakou formu SLAM algoritmu, čímž do vyhodnocovacího procesu zahrnuje také data z minulosti a tím zvyšuje robustnost systému. Implementací SLAM algoritmu do bezkolizního řízení je možno docílit přesnější detekce překážek, čímž se zlepší možnosti regulace řízení. Zejména práce [9] je ukázkou toho, že koncept levných sensorů a běžného výpočetního výkonu dovoluje implementaci protikolizního systému podpořeného SLAM algoritmem.

Pro spolehlivý bezkolizní řídicí systém v prostoru se však použití stereokamer zdá být nezbytné. Pokud se uvažuje možnost rozšíření detekce překážek do 3D, je nutné počítat s tímto faktem při návrhu.

# Kapitola 3

# Návrh řídicího systému

Cílem této práce je návrh obecných abstraktních tříd pro simulaci a testování řídicích systémů multikoptér a také implementace funkčního bezkoliziního řízení. Tato kapitola se bude věnovat především konkrétnímu protikoliznímu systému, který však bude zároveň ilustrovat funkčnost obecného návrhu. Abstraktní třídy jsou pak podrobněji popsány v kapitole 5.

Kromě omezení a požadavků v sekci 1.1 bude dále při návrhu systému uvažována multikoptéra s již implementovaným nízkoúrovňovým řízením, jehož vstupem jsou požadované lineární a angulární rychlosti. Díky tomuto předpokladu je možné navrhnout bezkolizní řízení bez ohledu na typ multikoptéry[1].

Je nutné podotknout, že při návrhu je brán ohled na možnost budoucí eliminace některých přijatých omezení a požadavků.

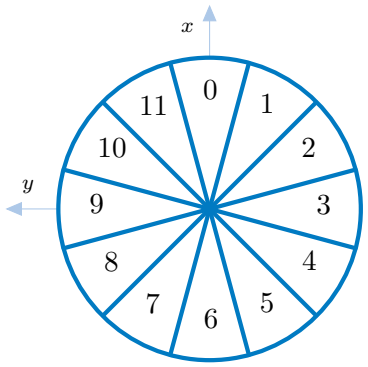
## 3.1 Sensory a detekce překážek

Uvažujmě rovinu rovnoměrně rozdělenou do sektorů (podobně jako v [8]). Vzhledem k pozorovacímu úhlu ultrazvukových dálkoměrů SRF-08 (který je přibližně 30°) je možné pokrýt každý sektor jedním senzorem, celkem tedy 12 sonarů. Rozložení sektorů je včetně číslování užitého v kódu a os simulátoru MORSE na obrázku 3.1. Tento konkrétní přístup také předpokládá umístění sensorů ve středu helikoptéry. Dostatečné pokrytí všech sektorů lze dosáhnout i s jinou konfigurací a počtem senzorů. Volbou většího množství (dostatečně pokrytých) sektorů pak lze docílit vyšší přesnosti detekce překážek.

Pokud se předpokládá prostředí s překážkami, jejichž detekce je pro sonary problematická, můžou být US senzory být doplněny i o infračervené sensory. Zde bude předpokládáno prostředí, ve kterém je detekce překážek sonary dostatečně spolehlivá. IR senzory však jsou v simulaci také implementovány.

Rozmístění senzorů umožňuje detekovat překážky prakticky v celé rovině. Sensory však mají přesnost jednotek centimetrů a čas od času vrátí i zcela chybné měření. Tyto nedostatky z části eliminuje filtrace hodnot mediánem. Pokud je dostupná mapa obsazenosti generovaná SLAM aloritmem, je možné

[1] Ačkoli je dále v textu místy použit obecnější termín *helikoptéra*, jedná se vždy o multikoptéru.




**Obrázek 3.1:** Rozdělení roviny na sektory

ji použít k zpřesnění aktuálních měření senzorů, případně korekci chybných nebo dokonce chybějících měření. Systém tedy bude schopen z mřížky (mapy) obsazenosti odhadnout vzdálenost k překážkám.

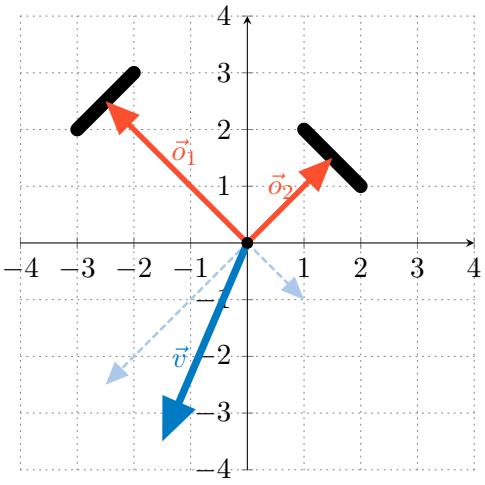
## 3.2 Předcházení kolizím

V každém sektoru je vyhodnoceno, v jaké vzdálenosti se nachází nejbližší překážka. Pokud je helikoptéra dostatečně vzdálena od všech překážek, je jí umožněn volný pohyb. V případě, že je ve směru pohybu detekována překážka v *blízké* vzdálenosti, je rychlost v tomto směru omezována až do uplného zastavení v *kritické* vzdálenosti, kterou helikoptéra udržuje. Překážky v ostatních směrech jsou také detekovány a pokud je jejich vzdálenost *kritická*, algoritmus na základě směru a vzdálenosti všech překážek vypočítá velikost a směr rychlosti, která zabrání kolizi.

Výše uvedený postup byl implementován v jazyce C++ a přepsán do pseudokódu, který je uveden v Algoritmu 1. Poznamenejme, že zde uvedený kód je oproti skutečné implementaci zjednodušený, z funkčního a principiálního hlediska jsou však ekvivalentní.

### 3.2.1 Popis pseudokódu

Nejprve proběhne inicializace proměnných. Proměnné $v\_x$ a $v\_y$ jsou řídicí povely (rychlosti ve směru os x a y robotu), které jsou po skončení algoritmu předány jako návratová hodnota. Proměnná $v\_angle$ zde představuje úhel pohybu daný řídicím povel vzhledem k ose x. K němu příslušný sektor pak reprezentuje $sector\_v$, který vrátí funkce `get_sector_from_angle()`. Proměnné *crit* a *near* pak informují, zda je alespoň v jednom sektoru detekována překážka v kritické resp. blízké vzdálenosti. Vektory $crit\_dists[]$ a $near\_dists[]$ pak uchovávají přímo jednotlivé vzdálenosti. Pokud v sektoru překážka v dané vzdálenosti detekována není, prvek nabývá záporné hodnoty (-1).




**Obrázek 3.2:** Ilustrační příklad výpočtu řídicího povelu

Algoritmus nejprve funkcí `udpate()` přiřadí vektorům *crit_dists*[] a *near_dists*[] naměřené hodnoty ze senzorů[2] dle Algoritmu 2. Pokud není v sektoru detekována překážka v příslušné vzdálenosti, hodnota je nastavena na -1. Následně se vyhodnotí, jestli řídicí povel směřuje do sektoru, ve kterém se nachází překážka v kritické vzdálenosti.

Pokud ano a zároveň není aktivní PID regulátor, je PID regulace aktivována (samotná regulace proběhne později, viz Algoritmus 3). PID regulace je pak deaktivována, pokud se robot vzdálí od překážky o kritickou vzdálenost a navíc o práh $PID\_TH$. Tato hystereze zabrání střídavé aktivaci a deaktivaci PID regulace v kritické vzdálenosti.

Dále může nastat případ, kdy řídicí povel nesměřuje do kritického sektoru, ale existuje alespoň jeden sektor, kde je překážka v kritické vzdálenosti detekována. Algoritmus pak vypočte nový řídicí povel, který zamezí kolizi. Výpočet probíhá následovně: Pro každý sektor je určen vektor, jehož směr je shodný s osou sektoru a jehož velikost je dána součinem nejbližší detekované (*kritické*) vzdálenosti a regulačního koeficientu. Regulační koeficient $k_c$ je dán exponenciální funkcí s časovou konstantou $T_c$ a roste se zmenšující se vzdáleností k překážce. Otočením tohoto vektoru pak získáme vektor pohybu, který směřuje od překážky a jehož velikost je úměrná vzdálenosti k překážce. Posčítáním vektorů ze všech sektorů dostáváme výsledný řídicí povel. Tento postup ilustruje obrázek 3.2, kde $\vec{o}_1$, $\vec{o}_2$ představují vektory relativní polohy detekovaných překážek a $\vec{v}$ je vektor rychlosti výsledného řídicího povelu. Výhoda této metody vektorových přírůstků spočívá především v univerzálnosti, neboť není nutné implementovat stavový automat pro různé situace (např. pohyb mezi vertikálními paralelními zdmi).

Pokud není v žádném sektoru překážka v kritické vzdálenosti, avšak řídicí povel směřuje do sektoru s detekovanou překážkou v blízké vzdálenosti, je povel regulován exponenciální funkcí (s časovou konstantou $T_n$ určující

[2]Ve skutečnosti se jedná o medián z posledních tří naměřených hodnot.

strmost).

**Algoritmus 1** Řídicí algoritmus

**Require:** $v\_x$, $v\_y$, $dists[]$, $CRIT\_DIST$, $NEAR\_DIST$, $PID\_TH$, $k$, $T\_c$, $T\_n$
**Ensure:** $v\_x$, $v\_y$

$v\_angle$ = atan2($v\_y$, $v\_x$)
$sector\_v$ = get_sector_from_angle($v\_angle$)
$crit$, $near$ = false
$PID\_active$ = false
update($dists[]$) ▷ Fills $crit\_dists[]$ and $near\_dists[]$
**if** $crit\_dists[sector\_v] > 0$ **then**
    $v\_x$, $v\_y$ = 0
    **if** not $PID\_active$ **then**
        $PID\_active$ = true ▷ PID
    **end if**
**else if** $PID\_active$ and $near\_dists[sector\_v] > (CRIT\_DIST + PID\_TH)$ **then**
    $PID\_active$ = false
**end if**
**if** $PID\_active$ **then**
    PID_control()
**else if** $crit$ **then** ▷ $crit$
    **for** $i$ **in** $crit\_dists[]$ **do**
        **if** $crit\_dists[i] > 0$ **then**
            $error = CRIT\_DIST$ - $crit\_dists[i]$
            $k\_c$ = 1.1 - exp($-T\_c$ * ($error$))
            $v\_x$ -= $error$ * $k\_c$ * cos($sector\_angle$)
            $v\_y$ -= $error$ * $k\_c$ * sin($sector\_angle$)
        **end if**
    **end for**
    **if** $near\_dists[sector\_v] < 0$ **then** ▷ Allow to move to free sector
        $v\_x$ += $v\_x$ * cos($sector\_angle$)
        $v\_y$ += $v\_y$ * sin($sector\_angle$)
    **end if**
**else if** $near\_dists[sector\_v] > 0$ **then** ▷ $near$
    $k\_n$ = 1 - exp($-T\_n$ * ($near\_dists[sector\_v]$ - $CRIT\_DIST$))
    $k\_vx$ = $k\_n$ * cos( $sector\_angle$);
    $k\_vy$ = $k\_n$ * sin( $sector\_angle$);
    $v\_x$ *= $k\_vx$
    $v\_y$ *= $k\_vy$
**else**
    do nothing
**end if**

**Algoritmus 2** update

**Require:** $dists[]$, $CRIT\_DIST$, $NEAR\_DIST$
**Ensure:** $crit\_dists$, $near\_dists$, $crit$, $near$

```
1: crit_dists.fill(-1)
2: near_dists.fill(-1)
3: for i in dists[] do
4:     if dist[i] < CRIT_DIST then
5:         crit_dists[i] = dist
6:         crit = true
7:     else if dist < NEAR_DIST then
8:         near_dists[] = dist
9:         near = true
10:    end if
11: end for
```

**Algoritmus 3** PID regulátor

**Require:** $dists[]$, $_i\_error$, $_d\_error$, $last\_error$, $sector\_v$, $CRIT\_DIST$, $d\_t$, $P\_GAIN$, $I\_GAIN$, $D\_GAIN$
**Ensure:** $out$, $_i\_error$, $_d\_error$, $last\_error$

```
1: error = dists[sector_v] - CRIT_DIST
2: i_error += error * d_t
3: d_error += (error - last_error) / d_t
4: out = (P_GAIN* error) + (I_GAIN * i_error) + (D_GAIN *
   d_error);
5: last_errror = error
```

# Kapitola 4
# Simulace

Simulace má oproti testům v reálném prostředí několik výhod. Obecně jsou časově i finančně méně náročné, odpadá riziko poškození majetku či zranění a simulace poskytuje možnost provádění experimentů, jejichž realizace by z různých důvodů byla nemožná. Také není nutné robot v době simulace fyzicky vlastnit, což může být výhodné v situaci, kdy se robot teprve staví či simulace jsou součástí návrhu konstrukce. V neposlední řadě take simulace poskytuje možnost jednoduše modelovat diametrálně odlišné situace s různými roboty.

Simulace však není přesně schopna postihnout reálnou situaci v celé její komplexnosti, ačkoli se při vhodném návrhu může dostatečně přiblížit.

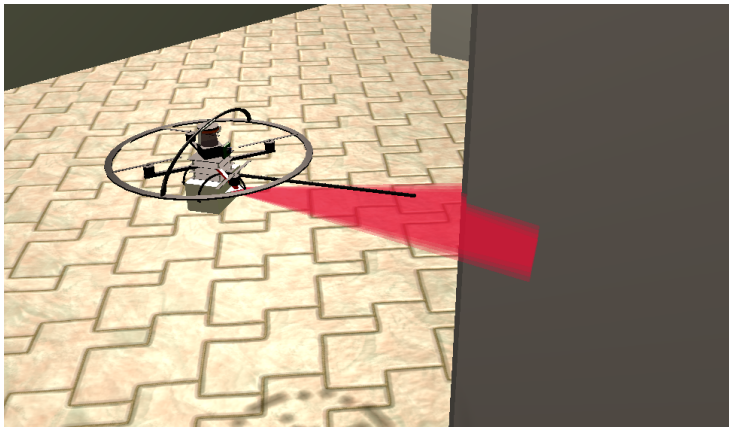
## 4.1 MORSE simulátor

K simulaci různých testovacích scénářu byl vybrán MORSE (Modular OpenRobots Simulator Engine)[11]. Jedná se o otevřený akademický simulační software. O vykreslování se stará Blender Game Engine a fyzikální výpočty zajišťuje knihovna Bullet. Samotné simulace a definice komponentů jsou napsány v Pythonu. V tom je také definovaný interface s MORSE zvaný *pymorse*.

Simulátor stojí na třech základních komponentách: **robotech**, **sensorech** a **akčních členech**.

Robot zdev podstatě tvoří pouze kontejner na sensory a akční členy. Sensory zde plní funkci robotických smyslů, neboť generují informace o okolí a poskytují je robotu. Aktuátory pak na základě vstupů ovlivňují kinematiku robotu. Simulátor již obsahuje knihovnu předimplementovaných komponent, systém je však navržen tak, aby bylo možné snadno přidávat vlastní komponenty. Každá komponenta musí být „upevněna“ na právě jednoho robota.

*Poznámka:* Jelikož v MORSE knihovně komponent často užívají názvy, které nejsou shodné s názvem implementované třídy, bude v textu vždy tučně uveden název třídy a název z knihovny komponentů bude případně uveden v závorce normálním stylem).



**Obrázek 4.1:** Screenshot prostředí MORSE - multikoptéra s vizualizací senzoru

### 4.1.1 Robot

MORSE nabízí dokonce dva typy robotické kvadrokoptéry, a sice model kinematický a model dynamický. Za účelem vyšší věrohodnosti byl vybrán dynamický model, tedy **Quadrotor**. Jelikož je na rozdíl od druhého modelu objekt typu *Rygid body*, podporuje více možností fyzikálních simulací.

### 4.1.2 Sensory

Knihovna sensorů již obsahuje obecnou třídu **LaserScanner**[1]. Ta poskytuje základ pro konkrétní implementaci rozmítaných laserových dálkoměrů (někdy také nazývány lidary nebo laserové scannerery). V knihovně je uveden i příklad infračerveného dálkoměru, který vznikl změnou parametrů rozmítaného laseru.

Pro potřeby této práce jsou tyto komponenty modifikovány tak, aby dostatečně věrohodným způsobem imitovaly ostatní použité sensory (IR sensory a sonary). Při imitaci jednorozměrných sensorů se postupovalo podobně jako u předimplementovaného IR sensoru **Infrared** změnou parametrů (viz tabulka 4.1). Paprsek obou sensorů je zde modelován několika vrstvami kruhových výsečí s totožným počátkem. Tím lze přibližně aproximovat tvar snímacího paprsku reálných sensorů. Jelikož laserové scannery vrací vektor bodů, sensory jsou ještě dodatečně modifikovány tak, aby vracely pouze nejmenší hodnotu a více se tak chováním přiblížili chování skutečných sensorů.

Reálné sensory však nevrací přesné hodnoty a v některých případech jsou měření (například nezachycený odraz) zcela chybná. Proto má každý sensor parametr *real_sensor*, který v případě nastavené hodnoty `true` zatíží měření Gaussovským šumem a určité procento měření vrátí maximální vzdálenost. Vlastnosti obou změn lze také nastavit příslušnými parametry.

[1] Ve skutečnosti odvozené implementace dědí od **LaserSensorWithArc**, která navíc vizualizuje paprsky sensorů.

| Sensor | Rozsah | Absoltní přesnost | Snímací úhel | Rozlišení[2] |
|---|---|---|---|---|
| SRF-08 | 3-600 cm | 5 cm | 30° | 1° |
| SRF-10 | 6-600 cm | 5 cm | 30° | 1° |
| Sharp GP2Y0A21 | 10-80 cm | 3 cm | 6° | 1° |
| Sharp GP2Y0A710 | 92-550 cm | 3 cm | 6° | 1° |

**Tabulka 4.1:** Tabulka parametrů sensorů vzdálenosti

### 4.1.3 Akční členy

#### Kontrolér pohybu

Několik akčních členů pro ovládání kvadrokoptéry již je předimplementovaných v MORSE knihovně. Byl vybrán **RotorcraftVelocity**, který z požadovaných lineárních rychlostí $v_x$, $v_y$ a $v_z$ a angulární rychlosti $v_{yaw}$ spočte adekvátní silové účinky, které aplikuje na robot. K tomuto výpočtu je užit PID regulátor, jehož parametry lze případně upravit. Regulátor pak udržuje zadanou rychlost. V případně zadaných nulových rychlostí se robot zastaví. K plné funkčnosti akčního členu byl nicméně třeba jistý zásah do kódu. Upravený akční člen nese název **MyRotorcraftVelocity**.

#### Odpor vzduchu

Pro vyšší reálnost simulace byl doimplementován i akční člen **AirResistance** (knihovna MORSE jej neobsahuje), který k dynamice robotu přidává odpor vzduchu (resp. odporovou sílu). Pro výpočet síly je použit vztah

$$F_o = kv^2, \tag{4.1}$$

vycházející z Newtonova zákonu odporu, kde $F_o$ je výsledná odporová síla, $k$ je celkový třecí koeficient a $v$ je absolutní rychlost ve směru pohybu. Pro určení $k$ pak lze použít vztah

$$k = \frac{1}{2} C_x \rho S, \tag{4.2}$$

kde $C_x$ představuje činitel odporu, $\rho$ je hustota prostředí a S je plocha tělesa. Výchozí hodnota pro celkový třecí koeficient je stanovena na $k = 0.06$ (pro $C_x = 1$, $\rho = 1.2$ a $S = 0.1$).

#### Poryvy

Dalším faktorem, který byl pro větší reálnost simulace zaveden (a také není součástí MORSE knihovny), jsou síly nepředvídatelné velikosti i směru neustále působící na stroj. Tyto síly vznikají jak poryvy větru, tak turbulencemi vzniklými rotory i změnami tlaku a jinými jevy. Příslušný nově implementovaný akční člen nese název **Disturbances**.

Tyto poruchy jsou simulovány náhodně se měnící silou (Gaussovské rozložení) ve směrech os $x$, $y$ a $z$. Akční člen umožňuje nastavit četnost změn, jejich

maximální velikost a standardní odchylku tak, aby co nejvíce odpovídala modelované situaci.

### 4.1.4 Scénář simulace

Scénář simulace (angl. *Builder Script* nebo *simulation scenario*) je skript v jazyce Python, obsahující informace o počátečním stavu simulace. Je uložen jako soubor *default.py* v kořenovém adresáři MORSE projektu. Vytvářejí se zde jednotlivé komponenty, upevňují se na roboty a nastavují se jejich parametry. Take se zde vytváří model prostředí, ve kterém simulace bude probíhat a definují se způsoby komunikace s okolím. V našem případě jsou to síťové sockety.

V tomto skriptu je také implementováno načtení konfigurace senzorů z INI souboru (viz 5.3). Aby skript osadil helikoptéru sensory přesně tak, jak očekává řídicí algoritmus, je nutné použít stejný inicializační soubor, jaký načítá **Robot**.

Ukázový scénář simulace s použitím základních funkcí a konstrukcí je na Výpisu 4.1.

**Výpis 4.1:** Ukázkový scénář simulace

```
from morse.builder import *

quad = Quadrotor()
quad.GroundRobot = False
quad.translate(x = 1, y = 1, z = 1)

ctrl = MyRotorcraftVelocity()
quad.append(ctrl)

rfinder = SRF08()
quad.append(rfinder)

ctrl.add_service("socket")
rfinder.add_stream("socket")

env = Environment('environments/indoor_1.blend')
```

### 4.1.5 Komunikace

MORSE umožňuje komunikaci pomocí několika standardů. Patří mezi ně TCP/IP sockety, ROS, YARP a další. Z důvodu jednoduché implementace (standardní knihovna umožňuje základní obsluhu) jsou pro komunikaci mezi MORSE a kódem použity sockety.

Základním komunikačním portem s MORSE je port 4000. Tento port obsluhuje obecné požadavky týkající se simulace a zároveň umožňuje zasílat příkazy akčním členům. Datové proudy (typicky data ze sensorů) pak začínají

na portu 60000 a pokračují výše (tedy porty 60001, 60002, ...). Každému proudu přísluší jeden port, přiřazení však proběhne až po spuštění simulace a nelze jej předem spolehlivě ovlivnit[3]. Proudové sockety pak představují textové zprávy ve formátu JSON (viz dále) následované koncem řádku (znak `'\n'`. MORSE vždy posílá jen jednu JSON zprávu.

Komunikace s hlavním portem probíhá pomocí příkazů. Ty vždy začínají textovým identifikátorem požadavku (např. `id1`). Následuje specifikace poskytovatele služby. Pro obecné požadavky týkající se simulace je určen specifikátor `simulation`, pro konkrétní roboty je pak třeba tento robot a příslušnou komponentu specifikovat, například `quadrotor.controller`. Následuje název služby (shodný s názvem funkce v kódu) doplněný o parametry, vetšinou vektor hodnot v hranatých závorkách. Příkladem celého příkazu zaslaného hlavnímu portu může být `id0 robot.controller move [0.0, 0.1]`. Pokud je tento příkaz přijat a zpracován, následuje odpověď obsahující identifikátor příkazu, status zpracování (`SUCCESS` nebo `FAILED`) a případná návratová hodnota nebo popis chyby. Příkladem může být `id0 SUCCESS`.

Senzory komunikují s okolním světem pomocí datových proudů. Nejjednodušší formou této komunikace je využití socketů. Zprávy jsou zasílány v JSON (JavaScript Object Notation) formátu. Senzory pak s předem definovanou frekvencí posílají na příslušný port zprávy obsahující výstup sensoru. Příkladem takovéto zprávy může být `{"timestamp": 1502546846.5060325, "range": 1.0}`.

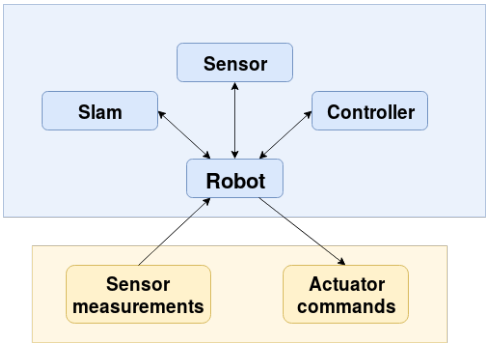
[3]MORSE nabízí možnosti volby portu pro každý proud, jedná se však o doporučení, na které se nelze v žádném případě spolehnout.

# Kapitola 5
# Implementace

Cílem této práce je návrh a implementace takového systému, který umožní co největší variabilitu využití a reflektuje rozmanitost helikoptér i sensorů. Podobné požadavky jsou kladeny i na simulaci. Základem návrhu je myšlenka, že řídicí algoritmus je nezávislý na platformě. Jeho chování je tedy totožné (při identických vstupech) v simulaci i na skutečném stroji a bude nutná minimální modifikace před nahráním do fyzické helikoptéry. Dalším požadavkem je, aby byla simulace i řídicí algoritmus inicializovány z (jednoho) INI souboru, což dále zjednoduší testování různých konfigurací.

## 5.1 Struktura

Základní struktura je naznačena na obrázku 5.1. Modrou barvou jsou vyznačeny součásti programu, žlutou barvou pak externí součásti (ať už skutečné, či simulované komponenty). Třída **Robot** (respektive z ní odvozené třídy) zde představuje společný prvek pro všechny ostatní třídy. Naprostá většina komunikace mezi ostatními třídami probíhá přes tuto třídu, aby se v co největší míře omezily závislosti na jednotlivých implementacích. Pak je možné jednoduše a s minimálními úpravami kódu měnit jednotlivé implementace. Pokud například vznikne nová implementace třídy **Controller**, kterou chceme nahradit kontrolér původní, nebude nutný zásah do kódu ostatních tříd, jelikož interface bude stále stejný.




**Obrázek 5.1:** Základní struktura programu

## 5.2 Vlákna

Přestože třídy nejsou kompletně thread-safe[1], návrh počítá s existencí více vláken a proměnné, u kterých se předpokládá přístup z různých vláken, mají vždy implementováné vláknově bezpečné přístupové funkce (tzv. *gettery* a *settery*). Jedná se například o funkce související s polohou a rychlostí robota, ale i přístup k sensorům a jejich datům.

## 5.3 INI soubory

Na začátku program načte data ze souboru *ancos_init.INI*. Ten zatím obsahuje jen relativní cestu k inicializačnímu souboru robota (například *star_12_ini.INI*), je však možné snadno přidat další parametry. Z inicializačního souboru robota je třída **Robot** schopná načíst sensory se všemi parametry tak, aby odpovídaly sensorům v simulaci (pokud simulace načte stejný inicializační soubor pro robota). Nezbytnými parametry jsou id, typ sensoru, počet uložených měření, standardní odchylka šumu a jeho pozice upevnění na robotu.

Ke čtení z INI souborů v C++ kódu je použita jednohlavičková knihovna *INIReader* od Bena Hoyta [12].

## 5.4 Třída Robot

**Robot** je abstraktní třída (interface) popisující požadavky obecného robotu a implementující některé platformě nespecifické funkce. Mezi tyto nespecifické funkce patří například funkce `init_sensors`, která z inicializačního souboru načte data potřebná pro inicializaci instancí třídy **Sensor**. Funkce `set_velocity_command()` pak nastaví řídicí povel, který je dále předán přímo regulátoru pohybu. Dále poskytuje přístupové funkce k poloze a rychlosti robotu nebo zápisu a získání dat naměřených sensory. Také obsahuje funkci `start_controller()` která spustí nekonečnou smyčku vypočítávající akční zásahy do řízení. O zápis měření do senzoru se stará funkce *write_meas_to_sensor()*.

### 5.4.1 Třída MorseRobot

Jedná se již o konkrétní implementaci třídy **Robot**. Implementace slouží ke komunikaci se simulátorem MORSE, zajišťuje tedy získávání dat ze sensorů a zasílání řídicích povelů. Jelikož komunikace probíhá přes sockety ve formátu JSON (vit 4.1.5), byla implementována třída **MorseParser**, která zajišťuje přijímání a odesílání socketů a syntaktickou analýzu (angl. *parsing*). K tomu je využita jednohlavičková knihovna JSON od Nielse Lohmanna [13]. Čtení

[1] Zabezpečený přístup ke zdrojům sdíleným různými vlákny programu.

dat z každého sensoru probíhá v samostatném vláknu a ihned po zpracování je prostřednictvím třídy **Robot** zapsáno do **Sensoru**.

## 5.5 Sensor

Tato třída slouží převážně jako kontejner na uchovávání měření ze senzorů a poskytuje informaci o poloze uchycení na robotu. Každý sensor má své id (dáno inicializačním souborem), kterým jej lze jednoznačně identifikovat. Třída také podporuje také uchování několika posledních měření. Předpokládá se, že třída může být rozšířena o jednoduché zpracování dat (průměrování, medián, ...) nebo poskytování dodatečných informací o sensoru (například standardní odchylka chyby měření).

## 5.6 Třída Controller

Třída **Controller** má za úkol zapouzdřit algoritmy vyhodnocující získaná data a vyvodit z nich akční zásah do řídicího povelu. K tomu slouží především dvě funkce: `update()` a `eval()`.

Funkce `update` má za úkol vyhodnotit dostupná měření ze sensorů (dálkoměry, IMU, ...) a ostatních dat (zkonstruované mapy okolí, ...) a aktualizovat informaci o okolních překážkách, kterou **Controller** nese.

Voláním funkce `eval()` pak dojde k vyhodnocení informace o okolních překážkách a spočtení akčního zásahu do řídicího povelu. V ideálním případě by mělo před touto funkcí dojít k volání funkce `update()`. Pokud však mezi voláními `eval()` nedojde k volání `update()` algoritmus přesto znovu provede výpočet s dostupnými hodnotami. Tento přístup může být výhodný v situaci, kdy je volání algoritmu častější, než frekvence sensorů.

### 5.6.1 Třída AlphaController

Z této třídy stojí za zmínku především implementace metody `eval()`. Ta určuje zda, a případně jakým způsobem je třeba zasáhnout do řídicího příkazu. Tento proces podrobněji popisuje Algoritmus 1 (viz sekce 3.2). V této třídě je také implementována funkce `fuse_dists()`, která spočte vzdálenosti k překážkám z mapy generované SLAM algoritmem a sloučí je s aktuálními měřeními ze sensorů. Vzhledem k aktuální absenci funkčního SLAM algoritmu však tento kód není aktivní.

Algoritmus byl v prostředí MORSE testován v několika typových situacích. Popis těchto testů, jejich výsledky a získaná data prezentuje Kapitola 6.

## 5.7 Třída Slam

Pro pohyb v neznámém prostředí je užitečné znát informaci o poloze robota i o svém okolí. SLAM algoritmus (angl. *Simultaneous Localization And Mapping*) na základě pravděpodobnostního přístupu pomocí měření ze sensorů sestaví

mapu, ve které zároveň lokalizuje robota. V této práci je jako jeho potenciální přínos uvažováno především zpřesnění měření ze senzorů a jejich nahrazení v případě chybných dat.

Tento interface zajišťuje především implementaci funkce `get_map()` a `update_map()`. První zmiňovaná vrací doposud vytvořenou mapu, zatímco druhá tuto mapu aktualizuje. V současném stavu se počítá s aktualizací mapy pro každý senzor zvlášť.

### 5.7.1 Třída BreezySLAM

Třída BreezySLAM je nástavbou knihovny BreezySLAM[14] představené v [15]. Knihovna však počítá pouze s jedním sensorem (laserovým scannerrem). Pro účel této práce byla tato knihovna modifikována tak, aby bylo možné algoritmus použít s větším množstvím senzorů, převážně sonarů a IR dálkoměrů. V současném stavu není funkční fáze lokalizace a SLAM algoritmus tedy v této práci použit k řízení není. Nicméně, návrh je připraven jeho budoucí začlenění.

## 5.8 Průběh kódu se simulací v MORSE

Program se z počátku pokusí spojit s hlavním portem MORSE simulátoru. Dokud toto spojení není úspěšné, program dál nepokračuje. Následně proběhne načtení inicializačního souboru *ancos_init.ini* a je vytvořena instance třídy **MorseRobot**. Ta inicializuje podle inicializačního souboru sensory a skrze třídu **MorseParser** se spojí se simulací. Po získání portů datových proudů v simulaci je pro každý tento proud spuštené samostatné vlákno. Úkolem každého vlákna pak je každý zachycený socket zpracovat a uložit na příslušné místo (například měření ze senzorů skrze **Robota** do **Sensoru**, pozici a rychlosti přímo **Robotu**). SLAM algoritmus pak může být spuštěn po každé aktualizaci sensoru nebo až po zvolené dávce měření. Tato vlákna pak běží až do ukončení simulace. Jakmile je robot úspěšně inicializován, čeká na spuštění kontroléru.

Pokud je spuštěn **AlphaController**, bude s danou periodou (výchozí hodnota je 100 ms) nejprve aktualizována informace o překážkách a následně je vyhodnocen akční zásah do řízení. Upravený řídicí povel je pak zaslán hlavním portem do MORSE.

Po celou dobu běhu programu jsou v terminálu zobrazovány nejdůležitější informace o průběhu simulace.

# Kapitola 6

# Testování

Pro ověření funkčnosti výše uvedeného bezkolizního řízení je navrženo několik scénářů s typickými situacemi, do kterých se helikoptéra může dostat. Testy jsou provedeny v simulátoru MORSE a byl pro ně vytvořen model vnitřního prostředí (viz Obrázek 6.1). Robot je osazen 12 sensory typu SRF-08 v konfiguraci typu hvězda. K dosažení vyšší věrohodnosti byly použity i akční členy **Disturbances** a **AirResistance** s výchozími parametry. U sensorů pak bylo povoleno chování reálného sensoru (viz sekce 4.1.2). Pokud není zmíněno jinak, testy jsou prováděny s parametry uvedenými v Tabulce 6.1.

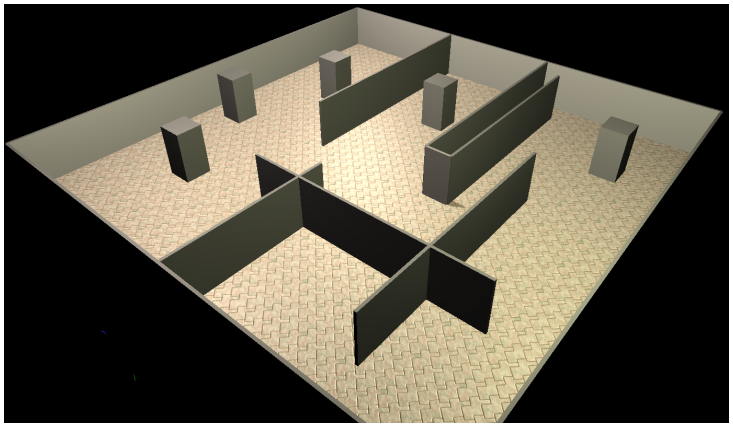
| parametr | hodnota |
|---|---|
| perioda řídicího algoritmu | 100 ms |
| poloměr helikoptéry | 0.4 m |
| $CRIT\_DIST$ | 0.7 m |
| $NEAR\_DIST$ | 1.4 m |
| $T_c$ | 1.5 |
| $T_n$ | 3 |
| $K_p$ | 0.3 |
| $K_i$ | 0.15 |
| $K_d$ | 0.01 |

**Tabulka 6.1:** Hodnoty parametrů pro testy

## Test 1: Let proti zdi

V prvním testu dostává helikoptéra po celou dobu povel k rovnoměrnému pohybu kolmo ke stěně rychlostí $0.2\,\mathrm{m\,s^{-1}}$ v ose x.

Na Grafu 6.2 je také patrný přechod do blízké vzdálenosti, kde je rychlost postupně snižována. Následně se helikoptéra dostane i do kritické vzdálenosti, kde se zastaví úplně. V grafu jsou také naměřené hodnoty vzdálenosti k překážce v sektoru 0.



**Obrázek 6.1:** Screenshot modelu prostředí MORSE pro testování

## Test 2: Let přímo do rohu

Rohová oblast je typickou situací, do které se helikoptéra ve vnitřním prostředí může dostat. V tomto testu je helikoptéře zadán povel rovnoměrného pohybu rychlostí $0.2\,\mathrm{m\,s^{-1}}$ vpřed (tedy osa x, která je zároveň osou sektoru 0) ve směru osy rohu.

Na Grafu 6.3 jsou vyneseny naměřené vzdálenosti překážek ve vybraných sektorech. Z jejich průběhu je patrné, že řídicí algoritmus zabránil kolizi a drží helikoptéru v kritické vzdálenosti od obou stěn.

## Test 3: Blízká překážka

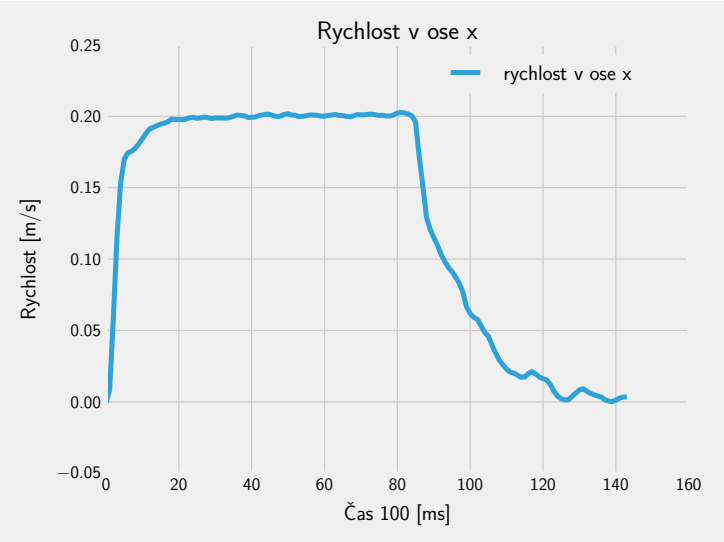
Helikoptéra se může dostat do situace, kdy se v kritické vzdálenosti objeví překážka. Tato situace může nastat v především v dynamických prostředích, ale také při nedokonalém pokrytí okolí sensory. Analýza chování řídicího algoritmu v této situaci může také pomoci k jeho vylepšení. Helikoptéra je tedy v tomto testu umístěna v těsné blízkosti překážky (cca 10 cm, kolmo na kladný směr osy x) a následně je spuštěn řídicí algoritmus.

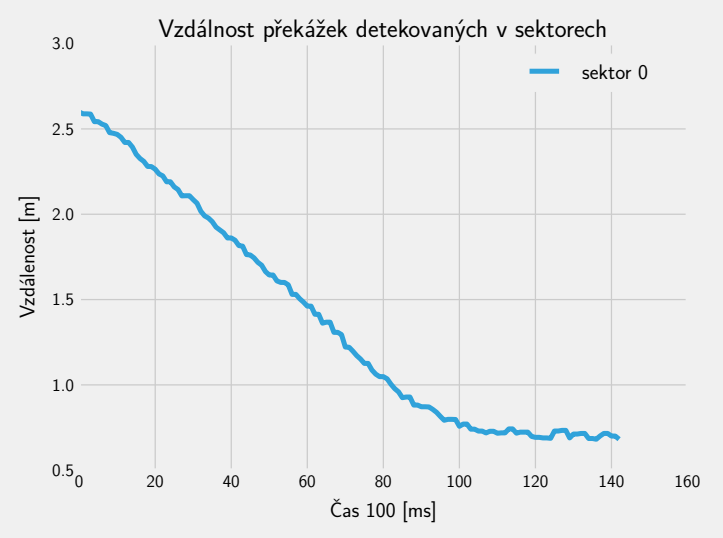
Graf 6.4 ukazuje přesnou polohu multikoptéry v globálních souřadnicích (zde jen osa x) MORSE. V prvních vteřinách je vidět razantnější změna polohy následovaná mírnějšími korekcemi k dosažení kritické vzdálenosti. Na grafu můžeme pozorovat i průběh rychlosti v ose x. Zde je již patrný vliv náhodných působících sil. V grafu vzdáleností v sektorech je také očividný zavedený šum sensorů i medián filtr, který nepropustil žádná falešná měření. Ačkoliv grafy rychlosti i měřené vzdálenosti vykazují šum, z grafu přesné polohy je patrné, že řídicí algoritmus dosáhl relativně hladkého pohybu.

## Test 4: Paralelní vertikální stěny

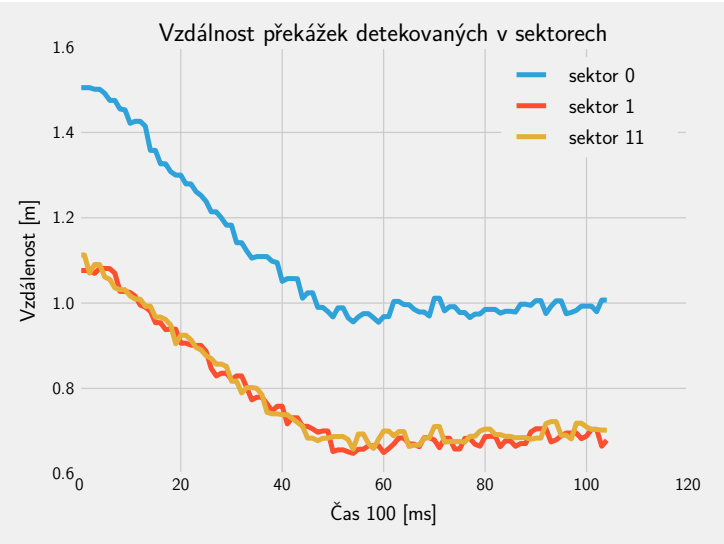
V uzavřených prostorách je vysoce pravděpodobné, že se helikoptéra dostane do situace, ve které se budou na obou stranách nacházet překážky v blízké nebo dokonce kritické vzdálenosti. Typickým příkladem je chodba. V tomto testu se helikoptéra nachází mezi dvěma stěnami (obě jsou ve vzdálenosti měnší, než kritické) a pohybuje se rovnoběžně s nimi rychlostí $0.15\,\mathrm{m\,s^{-1}}$. Po chvíli jedna stěna skončí.

Na Grafu 6.5 je možné pozorovat, že helikoptéra drží konstantní vzdálenost od obou stěn. Jakmile jedna ze stěn zmizí, rychlost vpřed se nemění, avšak helikoptéra zvýší vzdálenost od stěny na kritickou vzdálenost. Graf 6.5 také ukazuje průběh rychlostí.

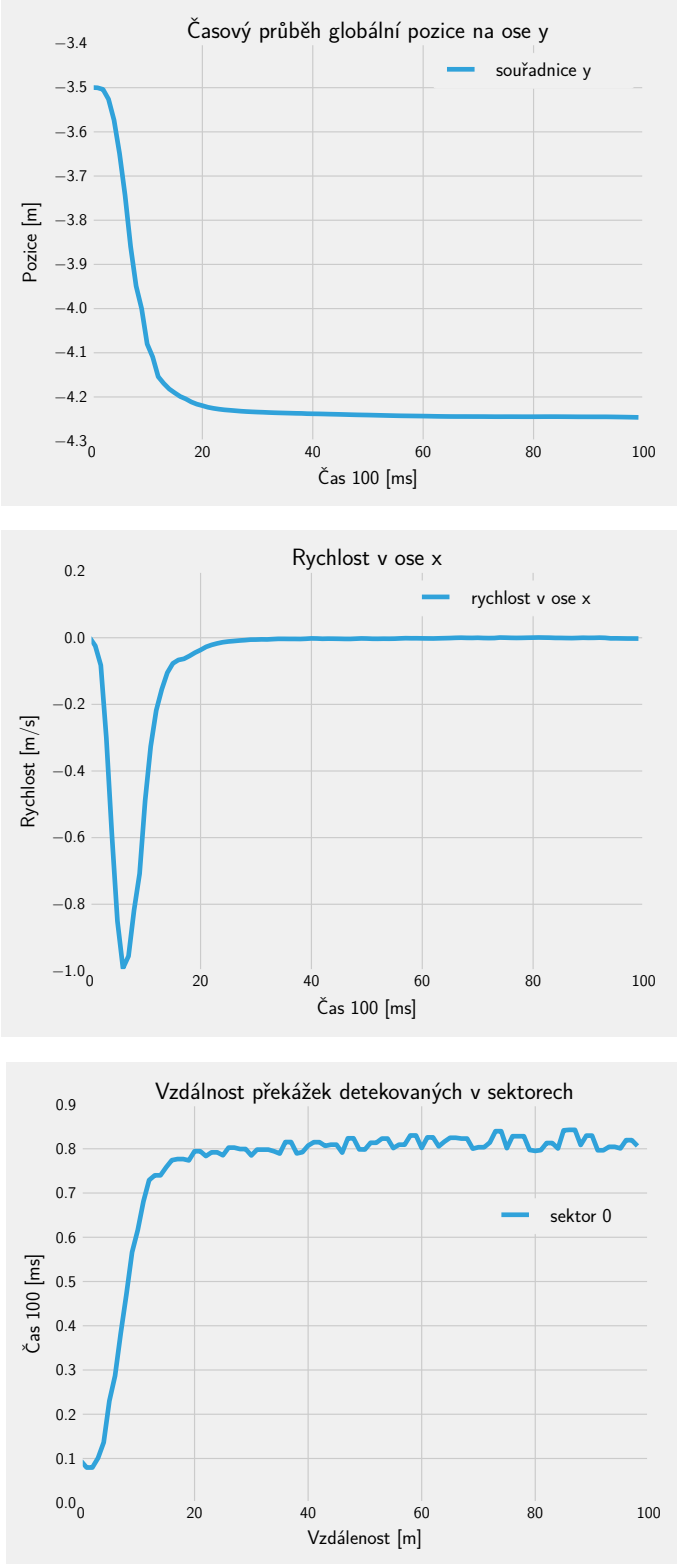


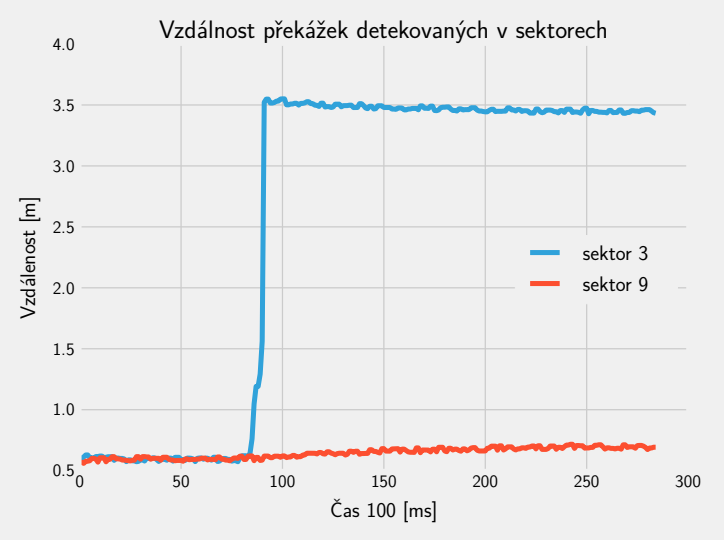

**Obrázek 6.2:** Test 1: Časový průběh naměřených vzdáleností

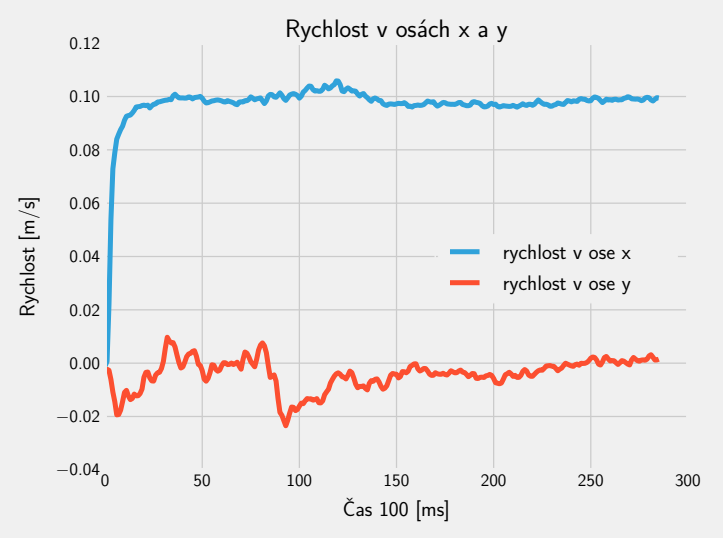



**Obrázek 6.3:** Test 2: Časový průběh naměřených vzdáleností




**Obrázek 6.4:** Test 3: Vzdálenost detekovaných překážek




**Obrázek 6.5:** Test 4: Rychlost v osách x a z

# Kapitola 7

# Závěr

Tato práce si dala za cíl navrhnout prostředí pro vývoj, testování a simulaci provozu bezpilotních helikoptér. Nejprve bylo bezkolizní řízení rozebráno po teoretické stránce a uvedeny některé z přístupů v komerční i akademické sféře. Dále byl představen konkrétní návrh detekce překážek a předcházení kolizí a pseudokód algoritmu určeného k bezkoliznímu řízení.

Dalším z cílů bylo simulování chování multikoptéry a bezkolizního systému v modelových situacích. K tomu byl využit open-sourcový simulátor MORSE. Pro potřeby této práce pak byl rozšířen o nové komponenty a bylo vytvořeno rozhraní pro komunikaci simulátoru se zbytkem kódu.

V jazyce C++ pak byl implementován abstraktní návrh řídicího systému a zároveň implementován konkrétní bezkolizní řízení. Tento návrh pak byl v simulátoru otestován v několika scénářích.

Do budoucna je možné v rámci dimplomové práce návrh rozšířit do třech dimenzí a zahrnout nějakou formu SLAM algoritmu.

# Bibliografie


[1] Business Insider. *Drone market shows positive outlook with strong industry growth and trends*. 2017. URL: http://www.businessinsider.com/drone-industry-analysis-market-trends-growth-forecasts-2017-7 (cit. 13. 07. 2018).

[2] Takeo Kanade, Chuck Thorpe a William Whittaker. "Autonomous Land Vehicle Project at CMU". In: *Proceedings of the 1986 ACM Fourteenth Annual Conference on Computer Science*. CSC '86. Cincinnati, Ohio, USA: ACM, 1986, s. 71–80. ISBN: 0-89791-177-6. DOI: 10.1145/324634.325197. URL: http://doi.acm.org/10.1145/324634.325197.

[3] Google LLC. *Waymo*. 2018. URL: https://waymo.com/ (cit. 21. 05. 2018).

[4] Inc. Tesla. *Tesla Autopilot*. 2018. URL: https://www.tesla.com/en_EU/autopilot (cit. 21. 05. 2018).

[5] DJI. *Guidance*. 2018. URL: https://www.dji.com/guidance/info#specs (cit. 21. 05. 2018).

[6] Intel Corporation. *Intel RealSense Technology*. 2018. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/realsense-overview.html (cit. 20. 05. 2018).

[7] Inc. Skydio. *Skydio Technology*. 2018. URL: https://www.skydio.com/technology/ (cit. 11. 05. 2018).

[8] N. Gageik, P. Benz a S. Montenegro. "Obstacle Detection and Collision Avoidance for a UAV With Complementary Low-Cost Sensors". In: *IEEE Access* 3 (2015), s. 599–609. DOI: 10.1109/ACCESS.2015.2432455.

[9] J. Mendes a R. Ventura. "Safe teleoperation of a quadrotor using FastSLAM". In: *2012 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. Lis. 2012, s. 1–6. DOI: 10.1109/SSRR.2012.6523878.

[10] D. Holz et al. “Towards Multimodal Omnidirectional Obstacle Detection for Autonomous Unmanned Aerial Vehicles”. In: *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 2 (srp. 2013), s. 201–206. DOI: 10.5194/isprsarchives-XL-1-W2-201-2013.

[11] OpenRobots. *MORSE*. 2018. URL: https://www.openrobots.org/wiki/morse (cit. 21. 05. 2018).

[12] Ben Hoyt. *INIReader*. [Online]. Available from: https://github.com/benhoyt/inih. 2013.

[13] Niels Lohmann. *JSON*. [Online]. Available from: https://github.com/nlohmann/json. 2013.

[14] Suraj Bajracharya et al. *BreezySLAM*. [Online]. Available from: https://github.com/simondlevy/BreezySLAM. 2014.

[15] Suraj Bajracharya a Simon D. Levy. *BreezySLAM: A Simple, efficient, cross-platform Python package for Simultaneous Localization and Mapping*. Tech. zpr. Honors Thesis. Washington a Lee University, dub. 2014.