Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Inteligentní modul řídicího systému speciálních vozidel

Plzeň 2013 Lukáš Svoboda

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 22. srpna 2013

Lukáš Svoboda

Poděkování

Děkuji panu prof. Ing. Václavu Matouškovi, CSc. za konzultace a hlavně za umožnění realizace tohoto zajímavého tématu.

Abstract

Intelligent control module for special vehicles

This thesis proposes a simulation and methods how to plan and control any special robot vehicles. For track planning are implemented standard best search algorithm together with A* and also Dynamic programming (D*) to compare posibilities and usefullnes of each proposed solution. Points of generated track are reduced using Douglas-Peucker simplification method and than the path is smoothed for better representation of proposed track to control. PID controlling of the vehicle has been used for navigate on the track. Simulation of navigation control is supported with Linear algebra and also it is possible to add normally distributed noise to the robots movement. Simulation is possible to use for education purposes, because bring another visual way of understanding all implemented algorithm and PID controlling itself, but of course also for controling a real robotic vehicles.

Path generation has been also tested on real 2D maps of an environment which are generated by MRPT[1] libraries.

Proposed solution brings collision free smooth path and also way how to control a movement of a different wheel based vehicle using PID.

[1]Mobile Robot Programming Toolkit

Obsah

1 Úvod

1.1 Motivace

Žijeme v době rozmachu inteligentních zařízení. Takřka každým dnem se můžeme setkat s hlasovým ovládáním, vizuálním zpracováním informací, zpracováním psaného textu, různými formami strojového učení. Spousta z nás využívá v domácnosti robotický vysavač, či jistě viděla videa spojené s čistě autonomně řízeným autem v reálném provozu od společnosti Google. Každý si jistě pomyslí, že dostat se do okruhu robotiky vyžaduje buď velké štěstí, zkušenosti, nebo spoustu peněz. Ale ukažme si, že dnes již tomu není pravda. Díky zařízení Kinect a například platformě Kinbo[1], která vzniká na Západočeské univerzitě. Karel Čapek byl prvním člověkem, který slovo robot vymyslel a používal ho ve svých knihách. Autor chtěl toto slovo přivést na zmiňované univerzitě v život, nebo alespoň se na jeho „životě“ významnou měrou podílet. Vzhledem ke skutečnostem, že se již v minulosti podílel na vzniku platformy Kinbo zde na Univerzitě, kde měl na starosti právě hardwarovou část – vše sestavit a poskládat dohromady. Tudíž byla celá platforma „podomácku“ sestavena a mohl se pustit do studia problematiky navigace speciálních robotických vozidel.

1.2 Popis práce

Před vývojem problematiky navigačního systému byla důkladně prostudována literatura a nabytí potřebných teoretických znalostí v daném odvětví.

Na začátku této práce je velmi stručné teoretické shrnutí metod plánování a navigace cesty.

Dále se práce věnuje vlastním vývojem simulačního softwaru pro navigaci a plánování cesty. Na závěr také návrhu možného budoucího řešení lokalizace a integrace navrhovaného způsobu navigace do jednoho celku.

[1] (Kin)ect ro(bo)t

1.3 Problematika robotiky

Ještě než se ponoříme hlouběji do teorie, měl by mít čtenář na paměti pár základních postřehů v robotice.

K umožnění autonomní navigace robota v daném prostředí je nutná znalost prostředí. A to buď formou mapy, či manuálního měření prostředí a následné reprezentace takovéto mapy se spojitým prostředím v diskrétní podobě – v mřížce (více v sekci 2.1 na straně 6). Dalším problémem je skutečnost, že se robot může pohybovat ve stochastickém prostředí – prostředí náchylném na změny (přesun nábytku ...). To jsou jen základní problémy se kterými se musí vypořádat člověk, jež vyvíjí lokalizaci robota, či v našem případě navigaci a plánování cesty v prostředí.

Dalším problémem, se kterým se musíme vypořádat při návrhu navigačního systému robota, je, že i samotný pohyb robota není přesný (deterministický). Kola robota mohou podklouznout, rychlost a tedy i vzdálenosti robota nemusí proto odpovídat *odometrii*[2] (pokud ji má) robota, navíc kola nemusí mít správnou geometrii a mohou se stále stáčet jedním směrem. Toto všechno jsou komplikace, které je třeba brát v úvahu a mohou se v reálné situaci projevit.

1.4 Platforma Kinbo

Při vývoji navrženého systému řízení bylo přihlíženo k vlastnostem platformy Kinbo[3], na jejímž vzniku se autor v rámci semestrální týmové práce na Západočeské Univerzitě podílel.

Bohužel původní hotová platforma byla odcizena, proto byl znovu celý projekt samostatně vytvořen a sestaven (elektronika, díly, návrh podoby a dílenské zpracování vozidla) a aplikován již hotový software z předchozího řešení pro komunikaci s elektronikou vozidla. Sestavení a zprovoznění projektu zabralo pár měsíců práce, kdy byla několikrát předělávána elektronika.

Samozřejmě navržený systém potřebuje ke své funkčnosti hotový sys-

[2]Základem odometrie je znalost geometrického modelu robota. Je to proces, který popisuje transformaci dat poskytnutých enkodéry na změnu pozice a orientace robota.

[3]Kin-ect ro-bo-t

tém lokalizace a zatím tedy bez patřičné lokalizace není možno testovat, ale snahou bylo také přihlížet ke skutečnostem a vlastnostem tohoto reálného řešení, navrhnout systém řízení podle těchto zkušeností a částečně platformu přizpůsobit k jednoduché budoucí integraci řešení. Dalším důvodem byla také snaha o oživení daného projektu na Západočeské univerzitě. Nová platforma sice byla přislíbena spoluzakladatelem projektu Ing. Petrem Altmanem, ale hardware byl dodělán až těsně před koncem tohoto letního semestru a to by již na nějaké nabytí zkušeností a pokusů s platformou bylo pozdě.

1.4.1 Popis platformy

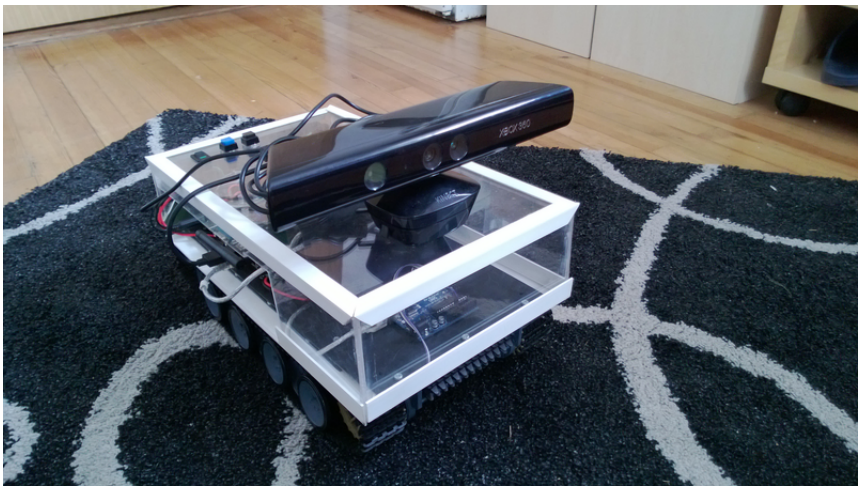
V této kapitole čtenáři představíme výhody/nevýhody použité platformy, její uplatnění a softwarovou výbavu. Původní nápad na vznik platformy se zrodil právě v hlavě kolegy z ročníku *Ing. Petra Altmana*. Cílem projektu bylo umožnit v rámci systému platformy svým uživatelům snadno umístit své programové řešení problému na počítač robota, na centrální počítač a na počítač uživatele modul řešící problém související s řízením robota. To znamená, že uživatel za využití námi vyvinutého programového rozhraní (*API*[4]) napíše tři programy, každý cílený pro jeden z počítačů. Tyto programy za ideálních podmínek budou moci vzájemně komunikovat právě prostřednictvím zmíněného *API* a umožní tak skrze počítačovou síť a WiFi robota ovládat. Na každém z počítačů tedy poběží samostatné programy, kde každý může využít potenciálu počítače. Například program běžící na centrálním počítači může využít vysoký výpočtový výkon grafické karty či program umístěný na robotu může využít minimální časové zpoždění při získávání obrazu z Kinect kamery a obsluhovat tak například autonomní pohyb robota.

1.4.2 Technika vozidla

Abychom usnadnili práci programátorům (snadnější plánování cesty), postavili jsme robota na tankovém (pásovém) podvozku, který umožňuje otočení robota kolem osy na místě (viz obr. 1.1).
Přehled robota:

- Kinect kamera—originální kamera, která pochází z příslušenství hrací konzole Xbox 360.

[4]Aplication Programming Interface



Obrázek 1.1: Má vlastnoručně vytvořená platforma Kinbo.

Specifikace kamery:

- VGA kamera – rozlišení 640 × 480.
- Infra senzor – zachycuje hloubkovou informaci obrazu, hw rozlišení kamery je v tomto případě 320 × 480 a je interpolováno na rozlišení 640 × 480. Hloubková informace je uložena v 11 bitech, čili umožňuje rozlišit 2048 hloubkových úrovní.
- Rozsah senzoru je s použitím *opensource* ovladače *Freenect* $\sim$ 0,5 až 4m.

- MiniITX Jetway NF2900 s Intel Atom druhé generace -– 1.6Ghz Dual-Core, 2GB DDR3 RAM, paměť – 16GB CompactFlash.
- Podvozkové šasi tanku Panzer ve velikosti 1:16.
- PWM regulátory motorů.
- Arduino pro kontrolu regulátorů [13].

1.4.3 Architektura softwaru a výhody/nevýhody platformy

Realizované programy je nutno nějakým způsobem do jednotlivých počítačů implementovat, a proto jsme výsledný systém postavili na velice propracované architektuře, jejíž myšlenkou bylo nejen umožnit pohodlně umístit programy

na jednotlivé počítače, ale také i jednotlivé programy snadno obsluhovat. Především je i sledovat pro případ výskytu chyby ve spuštěném programu uživatele.

Bohužel celý systém byl programován před dvěma lety, přičemž se zjistilo, že dokumentace k softwaru přístupná pouze na serverech Origo již neexistuje.

S přibývajícími znalosti problematiky se také objevují další problémy a úskalí zvoleného řešení, vyjmenuji zde některá technická/softwarová omezení dané platformy, rozsáhlý popis problematiky a řešení konkrétních problémů je uveden v příslušné sekci 5.1 na straně 53.

1. Kinect
2. Absence *odometrie* na navržené platformě. Do budoucna silně doporučuji dodělat odometrické senzory na platformě, neboť se podstatně rozšíří portfolio možných řešení lokalizace, které jsou také na mnohem kvalitnější úrovni.
3. Slabý *framerate* (WiFi). Navržená platforma je postavena na přenášení surových dat přes wifi, sice jsou data komprimována, ale komprimace je velice nedostatečná a výsledný framerate je velmi špatný, což se nakonec také projevilo při testování jako silně nevyhovující řešení pro jediné možnosti lokalizace.
4. Dosavadní způsob ovládání motorů – bez využití *odometrie*.
5. Výběr zcela nevhodného programového vybavení pro oblast použití v robotice.
6. Výběr ovladačů pro zařízení Kinect a použitý operační systém.

2 Základní teorie navigace a plánování

V této kapitole postupně nabudeme znalosti potřebné k sestavení robustního řešení plánování a lokalizace. Dále se také stručně podíváme na teoretické znalosti lokalizace pohybu robota a mapování prostředí, neboť mimo rámec zadání byla navrhnuta možnost lokalizace a další postup práce.

2.1 Plánování cesty

Protože plánování cesty přináší často komplikované a výpočetně náročné řešení, však si vzpomeňme na známý problém obchodního cestujícího [2], jež se řadí mezi NP-úplné problémy, omezíme se i vzhledem k platformě na kterou je dané řešení mířené na algoritmy, jež vyhovují podmínkám a velikosti prostředí, ve kterém se bude platforma Kinbo potenciálně pohybovat.

K plánování cesty se ve skutečnosti využívá řada algoritmů, některé z nich mají podobné vlastnosti, některé se výrazněji odlišují. Obecně dva nejvíce používané způsoby hledání cesty jsou deterministické a čistě heuristické metody. Zde se zaměříme na deterministické metody, které mají poměrně jasně ohraničenou kvalitu výsledné cesty. Více o pravděpodobnostních technikách hledání cest naleznete v literatuře [22]. Setkali jsme se s řadou teoretických článků a se spoustou řešení různých univerzit, některé byly zajímavé, nicméně často jsou tato řešení příliš „znalostně překompenzovaná“. Proto jsme sáhli po osvědčené knize, jejíž spoluzakladatel Sebastian Thrun je největší kapacita v dané problematice a významnou měrou se podílí na vzniku autonomního vozidla společnosti Google [5]. Vzhledem k tomu, jaké má reference a s přihlédnutím k reálným výsledků věřím, že i způsob řešení, které jsem z daných literatur převzal a další elegantní poznatky z jiných materiálů společně s úpravou algoritmů podle mého vlastního selského rozumu a uvážení, je dostačující pro navigaci v jednoduchém prostředí platformy Kinbo, ale nejen ji.

Každý algoritmus nabízí řadu výhod, ale také nevýhod. Zde v práci se zaměříme na čtyři (respektive dva) nejpoužívanější algoritmy hledání cesty. Také předpokládáme, jak je obvyklé při plánování tras robotických vozidel s 2D reprezentací mapy prostředí. Náš problém spočívá v nalezení

nejkratší cesty na této mapě prostředí.

Nejprve představíme pozadí strategií plánování cesty ve statickém prostředí s využitím klasických algoritmů používaných ke generování plánu v této doméně. Dále si pak představíme již méně známé algoritmy, které se hodí pro plánování v dynamickém prostředí.

Metody plánování zahrnují nalezení sekvencí stavů, kterých je třeba dosáhnout, abychom se dostali z výchozího stavu do stavu koncového. Při obecném plánování cesty robota jsou stavy robota jeho pozicemi a přechody mezi stavy reprezentují akce robota, které je schopen vykonat, aby dosáhl patřičného stavu. Každá taková akce má nastavenu svou cenu akce přechodu mezi stavy. Cesta je optimální za předpokladu, že suma ceny akcí přechodu robota mezi pozicemi je nejmenší mezi sumami všech možných cest.

Technika navržených algoritmů pro plánování cesty robotů spočívá v reprezentaci prostředí, ve kterém se pohybuje, jako graf $G \subseteq (S, A)$, kde S je konečná množina všech pozic mapy a A je množina akcí, jež musí vykonat, aby se dostal z jedné pozice do druhé. Každá taková akce má svou již zmíněnou cenu přechodu. Dále budeme rozlišovat $S_{free} \subseteq s$, což je množina všech volných pozic mapy, kde se robot může vyskytovat a G je poté výsledná cesta, čili bude se jednat o podmnožinu pozic na mapě a přechodů mezi nimi.

2.1.1 Dijkstrův algoritmus

Vznik Dijkstrova algoritmu se datuje již k roku 1959 [18]. Algoritmus opakovaně prohledává nejbližší ještě neprozkoumané pozice a přidává je do seznamu pozic k další expanzi. V každém kroku vybere ze seznamu pozici s nejnižším ohodnocením $g(s_i)$. Tento postup je opakován do té doby, dokud nenajde cílovou pozici na mapě. To, jaké body k expanzi algoritmus vybere, určuje rozhodovací funkce popsaná níže.

Dijkstrův algoritmus garantuje nalezení nejkratší možné cesty (případně alternativní ekvivalentní cestu).

Rozhodovací funkce pro výběr cesty vypadá následovně:

$$f(s_i) = g(s_i) + cena, \tag{2.1}$$

kde $s \in s$ je pozice mapy a funkce $g(s)$ je cena cesty od počáteční pozice s_{start}, *cena* je konstanta a udává „cenu“ přechodu mezi sousedními pozicemi. Funkce *f(s)* potom udává celkovou cenu dané konkrétní pozice, včetně ceny přesunu na pozici.

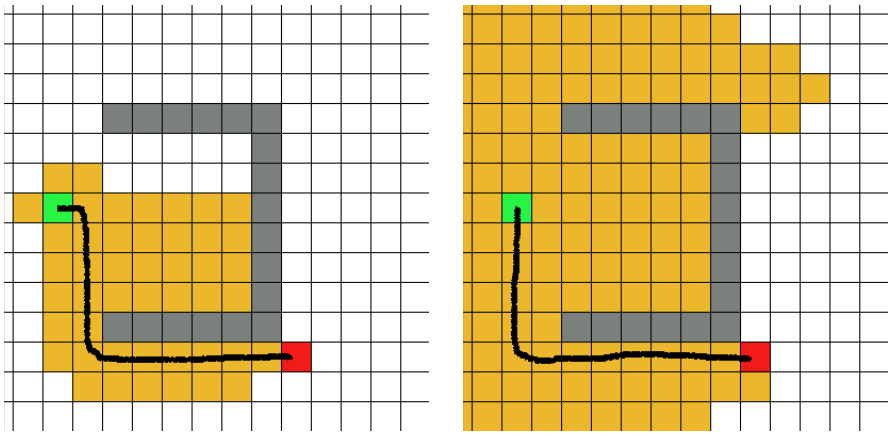
2.1.2 $A^\star$

Algoritmus $A^\star$ nabízí zajímavé rozšíření Dijkstrova algoritmu. Tento algoritmus vznikl na Stanfordské Univerzitě a poprvé byl popsán v roce 1968 [4]. Stejně jako Dijkstrův algoritmus vrací optimální cestu. Jedná se dnes o nejpoužívanější způsob hledání cesty, neboť dokáže najít cestu oproti Dijkstrovu algoritmu díky heuristické funkci za určitých podmínek podstatně rychleji.

Princip je velice podobný Dijskrovu algoritmu hledání nejlepší cesty, proto se vybízí vcelku jednoduchá úprava na algoritmus $A^\star$. Jeho kouzlo tkví právě v použití zmiňované heuristické funkce[4]. Jak si můžete všimnout (viz obr. 2.1), vliv heuristické funkce je značný a dokáže ušetřit spoustu pozic při vyhledávání. Dále jsem pouze upravil zkoumání bodů v okolí s respektování šířky robota. Abychom neplánovali cestu těsně podél překážek, ale také brali v potaz právě šířku, pokud se tedy v okolí o velikosti šířky robota objeví překážka, výsledek plánování nepovede touto cestou, neboť daný bod je brán ve smyslu překážky a tudíž, povede-li cesta robota kolem překážky, povede od překážky v minimální vzdálenosti odpovídající šířce robota.
Rozhodovací funkce pro výběr cesty vypadá následovně:

$$f(s_i) = h(s_i) + g(s_i) + cena, \tag{2.2}$$

na rozhodovací funkci je vidět, že se výběr expanze nerozhoduje pouze podle aktuální ceny buňky od počátku a ceny přechodu mezi buňkami, ale také heuristikou $h(s_i)$ (nejlepším odhadem ceny cesty k cílové destinaci od pozice s_i). Čili při rozhodování je také zahrnuta pozice cílové destinace a její relativní vzdálenost. Toto zdánlivě jednoduché, nýbrž velmi efektivní rozhodování dělá z algoritmu $A^\star$ právě jeden z nejpoužívanějších algoritmů v plánování cesty této doby.



(a) $A^\star$ (b) Dijkstrův algoritmus

Obrázek 2.1: Zde je vidět znázornění expanze buněk u algoritmu $A^\star$ a Dijkstrova algoritmu. Povšimněte si zejména o polovinu kratšího a tedy značně rychlejšího průchodu s výslednou optimální cestou k cíli (neboť algoritmus stále bere v potaz cestu od počátečního bodu). Oranžově jsou označeny všechny navštívené body, zeleně potom výchozí pozice a červeně cílová stanice. Černě je vyznačena výsledná cesta.

2.1.3 Dynamické programování - $D^\star$

Předchozí zmiňované algoritmy mají zásadní problém. Algoritmy byly a dodnes jsou široce využívány v plánování cest v robotice, ale jejich nevýhoda spočívá v tom, že předpokládají statické a předem známé překážky na mapě. Ve skutečnosti je však prostředí stochastické, obzvlášť v robotice se může např. objevit nová „živá“, či „neživá“ překážka v předem naplánované trase a tyto algoritmy neumožňují alternativní trasu. Je tedy nutné trasu znovu vypočítat, což může vést ke ztrátě kontinuity průjezdu robota danou oblastí. Navíc toto není jediný problém předchozích řešení, neboť jak již bylo zmiňováno i samotný pohyb robota je stochastický a dohromady s nepřesným měřením a odhadem pozice se může stát, že se robot ocitne v prostředí odkud nemá naplánovánu cestu a je tudíž nutné cestu z daného bodu opět spočítat, což může být opět výpočetně náročné a vede k dočasnému zastavení akcí robota.

Proto byl publikován v roce 1993 [7] algoritmus dynamického plánování cest. Tento algoritmus je často nazýván jako dynamický $D^\star$. Představuje evoluci v nabízených algoritmech plánování od dob vzniku algoritmu $A^\star$.

Algoritmus na začátku vypočte hodnoty pro všechny buňky mapy, jež je možné navštívit. Směr cesty se následně vybírá pomocí sousedů s nejnižším ohodnocením. Posléze je možné nalézt cestu z jakéhokoliv umístění na mapě bez dodatečných výpočtů do předem dané cílové destinace. Nalezne-li se robot na nechtěné pozici, ze které nemá plán cesty vlivem např. nepřesností pohybu, či zmiňované nutnosti objet blízkou překážku, jednoduše se bez následných zdlouhavých výpočtů vybere nová cesta. Pokud se vyskytne v okruhu nová překážka, je nutné pouze přepočet pár bodů v okolí, aby byl robot naveden opět na správnou, již vypočtenou cestu. Pro lepší představu čtenáři doporučuji podívat se na grafické znázornění plánů cesty k cílové destinaci (viz obr. 2.2).

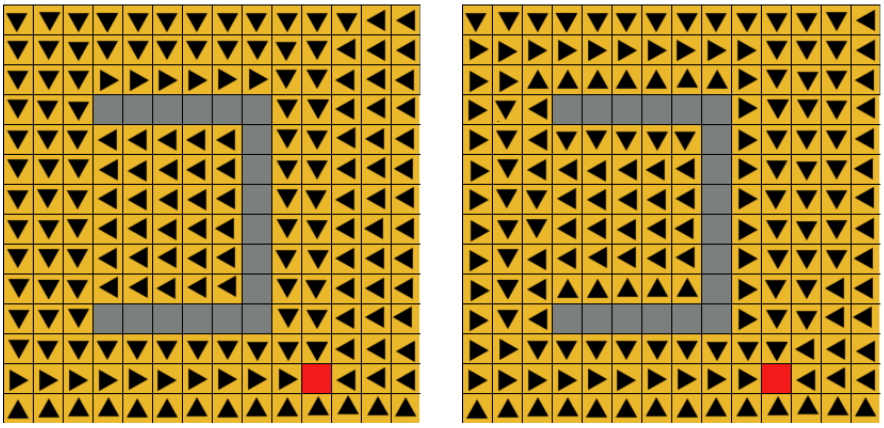
Má ovšem jednu podstatnou nevýhodu, neboť algoritmus je rekurzivní a musí procházet všechny dostupné pozice na mapě do doby konvergence algoritmu. Čili počáteční výpočet je velice časově náročný a daleko pomalejší než např. výpočet pomocí Dijkstrova algoritmu, natož pomocí algoritmu $A^\star$.

2.1.4 Dynamické programování – nedeterministický pohyb

Výše zmíněné algoritmy sice všechny tvoří optimální cestu, nicméně mezi cílovou a počáteční stanicí se může vyskytovat řada překážek a výsledná cesta může vést těstě kolem těchto překážek (u řešení Dijkstrova algoritmu a algoritmu $A^\star$ je tento problém částečně řešen nastavením šířky robota a nutností vést cestu minimálně o vzdálenosti dané šířky). Protože je pohyb robota po prostředí nedeterministický, rozhodně není nejlepší volba vést cestu těsně vedle překážky, neboť se může stát, že bude agent mírně odkloněn od požadované cesty a koliduje s blízkou překážkou.

Existuje proto celá řada vylepšení zmíněných algoritmů. U dynamického programování, které bere v potaz hodnotu pozic ve svém okolí a navádění na nejbližší vrchol s nejmenší hodnotou, lze využít právě *stochasticitu* pohybu robota v daném prostředí. Od klasického algoritmu představeného výše v sekci 2.1.3 na straně 9 jeho úprava spočítá v daném pohybu robota. Místo pohybu vpřed – s pravděpodobností řekněme $P(a_i) = 0,8$, kde $a_i \in a$, je 20% šance, že robot sjede z cesty. Při definovaném pohybu vpřed je 10% šance, že se robot ocitne místo pozice vpřed na pozici vpravo/vlevo od aktuální pozice s_i. Pokud například vlivem podklouznutí kol zůstane na stejné pozici, tak se nic neděje, protože víme, že se robot vyskytuje na pozicích bez kolize. Při

celkovém výpočtu ohodnocení dané pozice s_i bereme tedy také hodnotu pozic na které můžeme náhodně vstoupit s pravděpodobností $P = 0,1$. Pokud se zde vyskytne překážka, jež má vysokou cenu, dojde k nárůstu ceny takovéto pozice a při výběru pohybu bude vybrán pohyb vedoucí směrem od případného směru kolize a tím bude robot následně navrácen na optimální cestu vedoucí k cíli. Pro lepší představu doporučuji grafické znázornění cest (viz obr. 2.2) a popis implementace v sekci 4.2.2 na straně 28.



(a) Dynamické programování

(b) Dynamické programování - stochasticita

Obrázek 2.2: Znázornění výsledných plánů cest algoritmu dynamického programování do cílové pozice.

2.2 Úprava navržené cesty

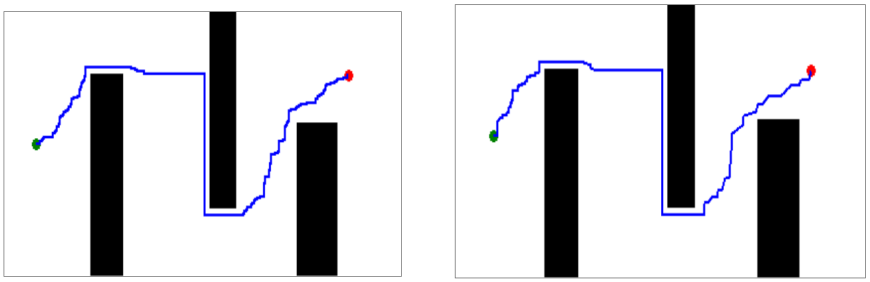
Máme-li naplánovánu cestu do naší cílové destinace, je třeba tuto cestu dále upravit, nebot' při letmém pohledu na nalezené cesty (viz obr. 2.3) mezi počátečním a koncovým bodem by byla navigace robota touto cestou velice nešt'astná.

2.2.1 Redukce bodů cesty

K redukci bodů na cestě, jež zanechají podobný popis cesty se využívá tzv. *Douglas-Peucker* [8][9] simplifikace. Redukce bodů našla uplatnění zejména v kartografii, kde se nalézají křivky čítající tisíce bodů, které je možné re-

dukovat a zjednodušit popis dané křivky se zachováním stejných vlastností. Na obrázku si povšimněte zjednodušení popisu cesty (viz obr. 2.3).

V základu nám jde o to zjednodušit počet bodů popisující danou křivku s minimálním vlivem na kvalitu popisu dané křivky. Počet bodů křivky nezjednodušujeme pouze z důvodů rychlejšího vykreslování cesty, méně „kostrbaté“ dráhy, nicméně hraje také podstatnou roli při následných úpravách jak se dozvíte v další sekci, neboť při letmém pohledu (viz obr. 2.3) každého jistě napadne, že pro navigaci agenta po dané upravené křivce stále není trasa vhodná.



(a) Výsledek hledání bez redukce bodů (celkem 592 bodů)

(b) Výsledná cesta s redukcí počtu bodů (celkem 62 bodů)

Obrázek 2.3: Výsledek hledání cesty mezi dvěma zadanými body. Výchozí bod je označen zeleně, koncový naopak černě. Rozdíl mezi cestami při redukci bodů – na levém obrázku bez redukce výslednou cestu vyobrazuje 592 bodů, zatímco vpravo je výsledek s 62 body.

2.2.2 Vyhlazení cesty

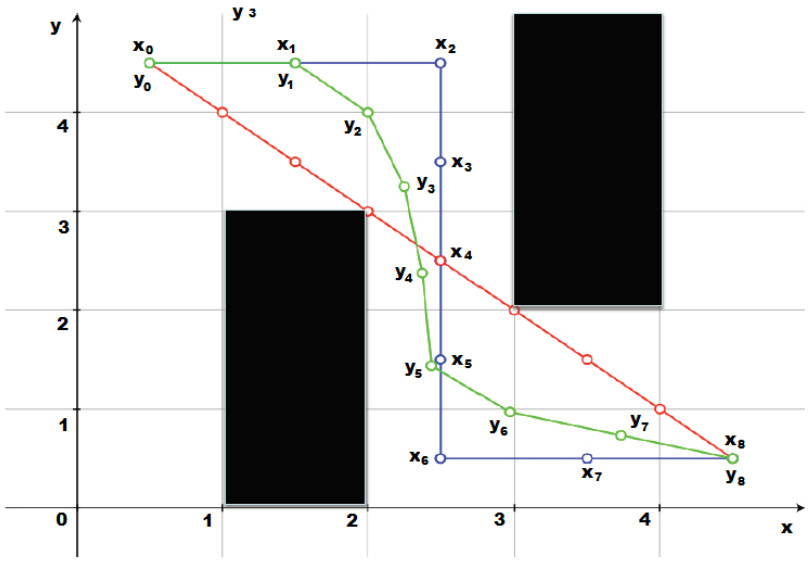
Vypočtenou cestu je třeba vyhladit, abychom mohli robota po trase lépe navigovat. Rádi bychom, aby vypočtená cesta byla plynulá a dávala co nejméně ostrých hran mezi přechody jednotlivých segmentů cesty. Naše ideální cesta je vyobrazena zeleně na následujícím obrázku (viz obr. 2.4). Pro více informací na toto téma nahlédněte do literatury [10].

K potřebnému vyhlazení je třeba si definovat množinu bodů X na cestě cestě, kde $x_i \in G$ a také stejnou množinu Y, kde $y_i \in G$.

Vyhlazení cesty spočívá v nalezení minima mezi následujícími rovnicemi.

$$(x_i - y_i)^2 \longrightarrow minimum \tag{2.3}$$
$$(y_i - y_{i+1})^2 \longrightarrow minimum \tag{2.4}$$

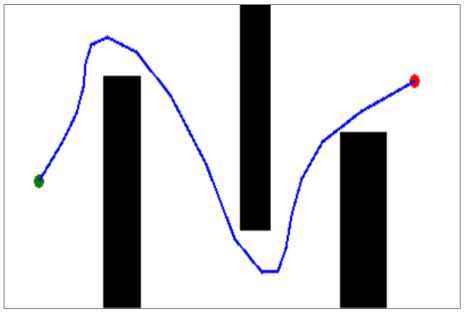
K minimalizaci těchto rovnic se využívá gradientní metoda největšího spádu [20].



Obrázek 2.4: Vyhlazení cesty. Zelená barva vyznačuje požadovanou cestu agenta, modře je potom vyznačena vypočítaná cesta.

2.3 Navigace agenta po cestě

Máme-li vytvořený plán cesty (viz obr. 2.5), resp. pozice, po nichž se musí robot vydat, je třeba robota na těchto pozicích s co nejmenší odchylkou od vypočtených pozic udržet. Jakékoliv velké odchýlení ze směru nepřipadá v úvahu, nebot' může vést ke kolizi robota. Zde oproti algoritmům plánování cest byla naopak volba jediná, a to sice využití PID navigace.



Obrázek 2.5: Výsledný vyhlazený plán cesty robota.

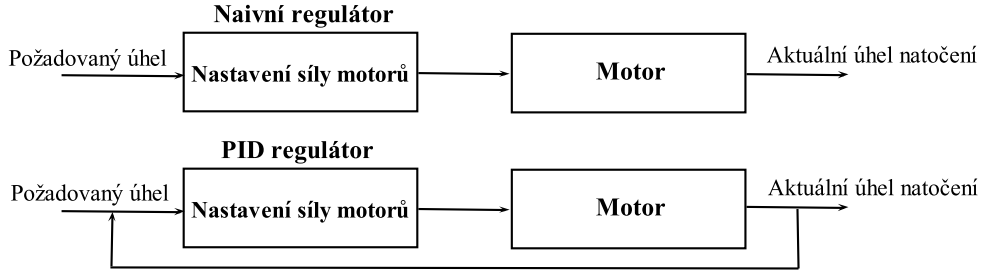
2.3.1 Regulace pohybu

Než psát zbytečnou spoustu řádků teorie, pokusím se zde vysvětlit princip použité PID regulace pohybu a její základní teorii pro navigaci agenta na trase na našem konkrétním řešení platformy Kinbo a způsobu, jakým je kontrolována síla motorů a tím výsledná rychlost či zatáčení, a také tím, jak je možno dosáhnout regulace této síly naším požadovaným směrem (setrvání na vytyčené trase).

K regulaci rychlosti platformy Kinbo je používána technika tzv. pulzní modulace – PWM (pro bližší informace nahlédněte do literatury [15]). Výsledná rychlost (resp. síla) motorů je udána rychlostí pulsů dané modulace. Prakticky v reálném programovém systému funguje kontrola motorů v rozmezí $< 0; 1 >$, kdy hodnota *1* znamená plný výkon motorů, *0* naopak vypnuté motory. Regulace směru zatáčení je pak dána množinou $< -1; 1 >$, kde pokud je nastavena hodnota *-1*, je zapnutý levý motor na plný předem definovaný výkon a pravý působí stejnou měrou proti směru otáčení, toto způsobí otáčení vozidla doprava po směru hodinových ručiček.

K regulaci po dané trase předpokládáme znalost lokalizace vozidla – pozici na mapě (x, y), úhel natočení θ a požadovaný úhel (resp. aktuální odchylka) natočení θ_{poz} vzhledem k požadavkům trasy.

Při krátkém zamyšlení vás jistě napadne nejjednodušší možný způsob řešení regulace, který je schématicky znázorněn níže (viz obr. 2.6). Při pohybu robota vypočteme jeho odchylku od trasy θ_{poz} v čase t a tuto odchylku nastavíme motorům. Při tomto naivním řešení se zásadně projeví změny požadavků na zatočení v čase a způsobí velkou chybu a pomalý nárůst v čase na



Obrázek 2.6: Schématické znázornění kontroly robota pomocí PID regulátoru.

požadovanou hodnotu (viz obr. 2.7(a)).

$$X = \theta_{poz} \tag{2.5}$$

Proporcionální regulátor:
Nabízí se tedy způsob tzv. *proporcionální* regulace, kdy rozdíl mezi požadovanou a aktuální hodnotou zatáčení vynásobíme váhou α a tuto změnu přičteme k požadované hodnotě zatočení. Označme si chybu v čase t od požadovaného zatočení (resp. odchylku od trati), jako $e(t) = (\theta_{poz} - \theta_{akt})$, potom :

$$X = \alpha \cdot e(t) \tag{2.6}$$

Tím docílíme velmi rychlého žádoucího úhlu natočení, neboť počáteční rozdíl mezi požadovaném a aktuálním natočením je poměrně veliký a dojde k přičtení velkých hodnot k původní požadované míře zatočení. Tento způsob má velkou nevýhodu, jakmile nedochází k dosažení nastavených hodnot, aktuální úhel zatáčení se stále zvyšuje a může se tedy stát že značně „přestřelíme“ požadované hodnoty (viz obr. 2.7(b)). Je-li hodnota α moc veliká, nedosáhneme jednoduše ustálené hodnoty.

Potřebujeme tedy nějaký způsob, který zachová vlastnosti Proporcionálního regulátoru a zároveň zpomaluje nárůst hodnot zatočení v čase, když se blížíme k ideálnímu ustálenému stavu (k naší vytyčené trase).

Proporcionálně-diferenční regulátor:
S tímto přichází Proporcionálně-diferenční regulátor, který přináší diferenční složku.

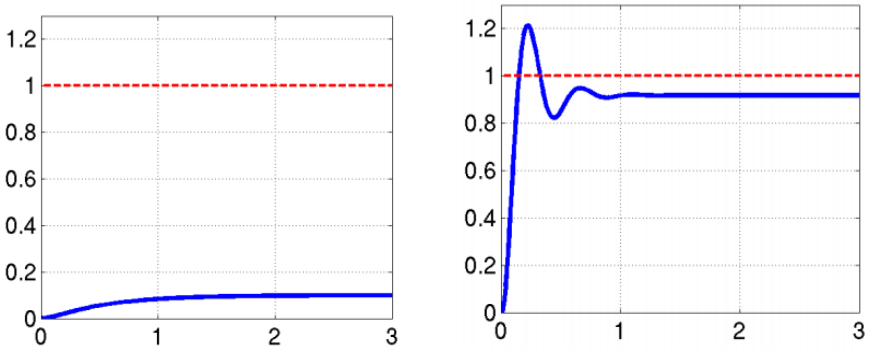
$$X = \alpha \cdot e(t) + \beta \cdot \frac{de(t)}{dt} \tag{2.7}$$

Výhody takto upraveného proporcionálního regulátoru jsou zřejmé, pokud se k trase přibližujeme velice rychle (rychle klesá odchylka od trasy), diferenční složka způsobí zpomalení rychlosti přibližování se k trase a zabrání nám tedy ve výrazném „přestřelení“ trasy. Má však drobný problém – nebere v potaz průměr celkové chyby na trase. Ve skutečnosti se totiž může stát, že budeme mít například na vozidle špatnou geometrii a bude se nám vlivem této nepřesnosti vozidlo stáčet stále doprava, s diferenčním regulátorem se tedy může stát, že nedocílíme, aby robot jel přesně na trase, nebot' se ustálí na hodnotě odpovídající dané odchylce kol (viz obr. 2.7(c)). Musíme tedy brát v potaz akumulaci chyby na trase, k tomuto přichází integrální složka regulátoru.

PID regulátor:

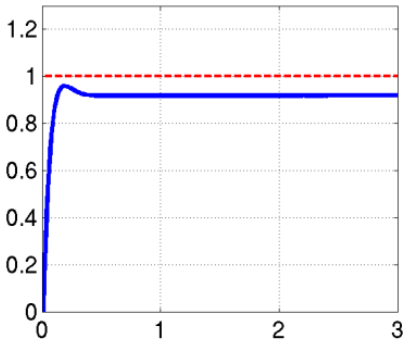
$$X = \alpha \cdot e(t) + \beta \cdot \frac{de(t)}{dt} + \gamma \cdot \int e(t)dt \tag{2.8}$$

Integrální složka regulátoru nám vyjadřuje sumu odchylek od počátku regulace na trase a tudíž dokáže vymazat naakumulovanou chybu, která mohla vzniknout např. právě díky výše uvedeným skutečnostem (viz obr. 2.7(d)).

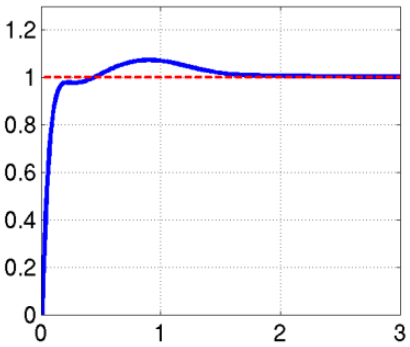


(a) Naivní regulátor

(b) Proporcionální regulátor



(c) PD regulátor



(d) PID regulátor

Obrázek 2.7: Zhodnocení chyb jednotlivých řešení kontroly vozidla. Osa x odpovídá průběhu úhlu natočení robota v čase t, zatímco osa y vynáší výsledný aktuální úhel natočení. Požadovaný úhel nastaven na obloukovou míru 1 radián $\approx$ 57 stupňů. Obrázek převzat z [16].

3 Lokalizace

Jelikož samotná lokalizace nebyla součást diplomové práce, algoritmy řešení lokalizace se vyvíjí již dlouhá léta a většinou v týmech více lidí na prestižních univerzitách, proto pro účely lokalizace doporučuji jedno z hotových navrhovaných řešení s využitím knihovny *MRPT* a propojením se softwarovým vybavením platformy Kinbo – více v sekci 4.4.2 na straně 48. Než se čtenář dostane k dané problematice, měl by také mít alespoň částečné teoretické povědomí o možnostech a principech fungování lokalizace. Samozřejmě toto bude velice stručný popis, nicméně vzhledem k autorova návrhu lokalizace a možnosti pokračování se snažíme čtenáři vštípit jen základní teorii dané problematiky. Pokud by se čtenář rád seznámil více s problematikou, doporučuji nahlédnout do literatur [5][23][24][25].

3.1 Filtry

Nejčastěji používanými filtry pro řešení lokalizace a mapování oblasti jsou *Částicové filtry* a *Kalmanovy filtry*. Filtry se používají ke kompenzaci chyb daných měřicími přístroji a samotným nedeterministickým pohybem robota.

3.1.1 Částicové filtry

Částicové filtry jsou třídou tzv. *Monte Carlo* simulačních metod. Jsou využívány v případech, kdy vidíme podmíněnou závislost mezi náhodnými veličinami spojených *Markovovým* řetězem [28]. Proces je tedy velmi efektivní na využití paměti počítače, další stav závisí pouze na aktuálním stavu a ne na stavech předchozích. Existují dostatečně robustní implementace aplikovatelné pro nelineární systémy šum nepodléhající Gaussovo rozdělení.

Hlavní úkol „Částicových filtrů“ je odhad stavu veličiny (X) založený na pozorovaných datech – odhad pozice vypočítaný z *odometrie* společně se senzorickými daty (s využitím externích senzorů). Pravděpodobnost distribuční funkce je využívána k reprezentaci stavu informace. Tyto filtry užívají velké množství částic x_i k reprezentaci případných stavů systému.

V čase nula je pravděpodobnost distribuční funkce dána jako $P(X_0)$. Jak

se systém v čase vyvíjí, upravuje se distribuční funkce, v čase $(t + 1)$ je potom dána $P(X_{t+1}|X_t)$. Za předpokladu senzoricky naměřené observace (Z) je $P(Z_{t+1}|X_{t+1})$. K řešení je využívána Bayesova formule. Více se dozvíte v již uvedené literatuře.

3.1.2 Kalmanovy filtry

V roce 1960 R. E. Kalman [27] navrhl efektivní řešení odhadu stavu statického, nebo dynamického procesu. Tento filtr dokáže odhadnout minulý, současný i budoucí stav, i když jsou části modelu neznámé.

Jedná se o rekurzivní filtr, který zaručuje optimální odhad požadovaného stavu lineárního systému ze série senzorického zašumělého měření. Problém těchto filtrů je, že jsou rekurzivní a jsou vyžadovány velké paměťové požadavky. Většina reálných systémů je však nelineárních.

3.1.3 Typy technik lokalizace a mapování

Existuje nepřeberné množství technik lokalizace a mapování v reálném čase – *SLAM*[1]. *SLAM* jsou iterativní metody a přináší odpověď na fundamentální otázky : „Jak vypadá okolní svět?“ a „Kde se nacházím?“.
Na to, abychom mohli položit odpověď na dané otázky jsou techniky většinou založeny na principech částicových nebo Kalmanových filtrů, které za pomoci senzorů robota (typicky *odometrie* a laserového snímání okolí) dokážou odhadnout aktuální polohu, také zahrnout nejistotu měřicích přístrojů a pohybu robota. Šum, který vzniká nepřesností měřicích přístrojů a pohybem robota, vede k akumulaci chyby v čase a také nepřesnému generování mapy prostředí. Toto naštěstí dokáže být kompenzováno řadou statistických metod, kde právě dvě zmíněné nejužívanější metody vedou k částečné eliminaci těchto chyb.

Zde nastíníme jen základní velmi stručné rozčlenění a principy jednotlivých technik:

- Pravděpodobnostní techniky – do nich spadají dvě nejvyužívanější techniky:

[1] Simultaneous Localisation and Mapping

  - *Monte Carlo* – SLAM založená na využití částicových filtrů.
  - Lokalizace a mapování s využitím Kalmanových filtrů.

- *Vizuální odometrie* – výše zmíněné techniky pracují na vstupu dvou nezávislých informací o okolí s využitím měření a poté odometrie vozidla. Techniky *vizuální odometrie* umožňují extrahovat informaci o odometrii pouze z jednoho zařízení, typicky stereokamery, či v poslední době právě populární kamery Kinect, která dokáže společně s klasickou VGA scénou dodávat právě mapu vzdáleností ke předmětům snímané scény. Z VGA kamery je poté s každým snímkem extrahován příznak v daném prostředí. Tyto příznaky jsou extrahovány typicky s využitím hranových detektorů. S dalším snímkem kamery je tento příznak znovu zaznamenán a jsou porovnány vzdálenosti mezi polohou kamery a daným příznakem po sobě jdoucích snímků, z těchto snímků je vypočítán výsledný pohyb kamery (resp. robota) a pozice robota společně s mapou prostředí. Při řešení se často využívají výše uvedené techniky filtrací.

Daný popis dává čtenářovi hrubý náhled do problematiky, a proč je v našem případě možno využít pouze výše zmíněnou metodu *Vizuální odometrie*, neboť naše platforma neumožňuje čtení *odometrie* robota. Bohužel algoritmy založené na tomto principu jsou vyvíjeny ve velké míře právě v posledních letech právě díky levné kameře Kinect a dostatečným výpočetním výkonům počítačových jednotek. Zatím tedy nejsou veřejně dostupné v takové míře jako ostatní metody založené na fyzické *odometrii* na vozidle, a proto i námi použitá knihovna *MRPT* sice nabízí spoustu funkcí ulehčujících vytvoření takovéto lokalizace, nicméně žádné robustní hotové řešení.

4 Realizace

Ke zhotovení práce byl vybrán programovací jazyk *C#* s *.NET Framework v. 4.0*. Program je rozdělen do několika souborů, ve kterých jsou části kódu rozčleněny do tříd podle vlastní datové hierarchie. V této kapitole si řekneme jen základní shrnutí používaných prostředků a velmi stručný popis zdrojových souborů. Na přiloženém CD (více v Příloze C.1 na straně 89) naleznete kompletní programátorskou dokumentaci aplikace s popisem jednotlivých tříd, které se nalézají v níže uváděných souborech.

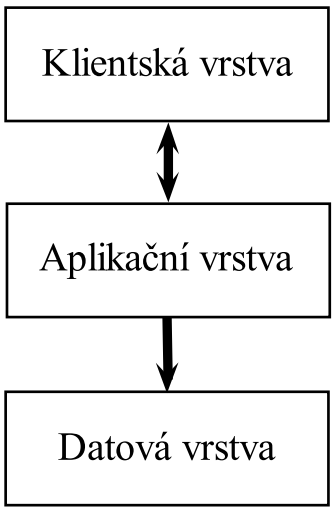
Protože úkolem práce bylo zhodnocení výsledků jednotlivých navrhovaných řešení navigace a plánování, byla první volba vytvořit přehledný software umožňující simulaci řízení, plánování a různých cest robota. Schéma průběhu simulátoru si můžete prohlédnout na obrázku níže (viz obr. 4.2). Abychom mohli nasimulovat chování robota včetně různých scénářů stochastického chování platformy, bylo také třeba vytvořit patřičný kinematický model robota, který bude matematicky podložen a podávat věrohodné výsledky vzhledem k nastavení modelu.

Samotná realizace byla rozdělena do více částí. Nejprve byl vytvořen samotný program simulace navigace a strategií plánu cesty, poté byla z dalšího zájmu v pokračování vývoje navržena podoba a základní funkcionalita klientské části. Tato část je založena na programovém vybavení platformy Kinbo umožňující manuální ovládání platformy, byl také navrhnut způsob lokalizace s využitím vizuální *odometrie* za pomoci knihovny *MRPT* [12] a jeho případné propojení s platformou *Kinbo*, kde byla také zhodnocena a navržena řada nutných úpravy platformy.

Při implementaci byla dodržována vlastnost třívrstvé architektury (viz obr. 4.1), a proto jsou jednotlivé části kódu rozčleněny do tříd podle vlastní datové hierarchie. Dále také byla zachována stálá možnost interakce uživatele s uživatelským rozhraním. Veškeré akce vykonávané aplikační vrstvou jsou proto zpracovávány v samotných příslušných vláknech.

Simulator:

- Adresář aplikace simulace obsahuje příslušné zdrojové soubory:
  - *„Aplication.cs“* – samotné grafické rozhraní aplikace s podpůrnými



Obrázek 4.1: Třívrstvá architektura aplikace.

metodami pro předzpracování obrazu,

- *„PID.cs“* – třída PID regulátor včetně algoritmu umožňujícího nastavení hodnot PID regulátoru,
- *„Planning.cs“* – aplikační třída obstarávající plánování trasy robota.
- *„Robot.cs“* – třída robota vč kinematiky pohybu,
- *„GaussianRandom.cs“* – náhodný generátor čísel Gaussova rozdělení,
- A další převážně podpůrné třídy datové vrstvy.

- Adresář *„bin“* s podadresářem *„Release“* obsahuje přeložený program *„RobotNavigation.exe“*.

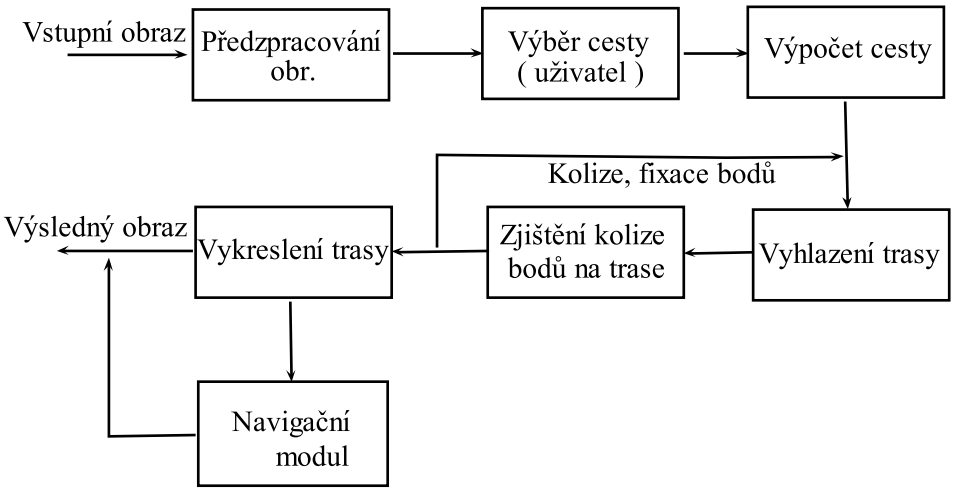
Klientský modul a modul lokalizace

- Adresář aplikace klientského modulu obsahuje složku *„bin“* s podadresářem *„Release“*, který obsahuje přeložený program ve formě *dll* knihovny. Tuto knihovnu spouští samotný klient platformy (více v sekci A.2 na straně 81). Dále zde nalezneme příslušné zdrojové soubory:
  - *„ClientClass.cs“* – klientský proces platformy Kinbo. Zde je zajištěn příjem signálu z kamery a zakomentované řešení meziprocesové komunikace pro další vývoj a integraci řešení (více v sekci 4.4.2 na straně 48),

- *„KinboClass.cs“* – proces běžící na straně robota,
- *„ServerClass.cs“* – serverová část platformy Kinbo,
- *„NamedPipeServer.cs“* – *meziprocesová* komunikace umožňující sdílení dat mezi *MRPT* [12] knihovnou a klientskou částí založenou na platformě Kinbo,
- A další převážně podpůrné třídy pro reprezentaci dat

- Adresář aplikace lokalizačního modulu obsahuje následující zdrojové soubory:
  - *„Program.cs“* – vstupní metoda aplikace,
  - *„Aplication.cs“* – inicializace uživatelského rozhraní a meziprocesové komunikace,
  - *„NamedPipeServer.cs“* – metody pro nastavení serveru zajišťující vytvoření požadovaného spojení mezi dvěma procesy.
  - *„SLAM.cpp“* – metoda lokalizace založená na vizuální odometrii,
  - A další převážně podpůrné třídy pro reprezentaci dat.

Popišme si tedy vše popořadě a začněme s hlavní programovou částí – simulačním softwarovým vybavením. Schéma postupu zpracování trasy a modulu navigace na této cestě je znázorněno na následujícím obrázku (viz obr. 4.2).



Obrázek 4.2: Schéma postupu při zpracování trasy a navádění robota na trase.

4.1 Předzpracování obrazu

Vstupní obraz, který pochází z knihovny pro lokalizaci *MRPT* [12], ale i z jiných obecných způsobů reprezentace mapy je vždy víceméně dvoubarevný, resp. oblasti překážek jsou vyznačeny tmavší barvou a naopak volný prostor barvou velice světlou (viz obr. 6.5). Z tohoto důvodu je prováděn převod vstupní obrazové bitmapy do šedé stupnice.

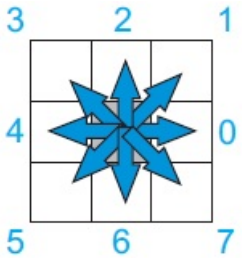
Testování regulace jsme chtěli umožnit na různých uzavřených křivkách, proto bylo třeba, aby si danou křivku mohl uživatel jednoduše sám vytvořit podle svého uvážení. Bylo tedy nutné nalézt trasu takového obrazu, která odpovídá hranici takto vytvořené křivky. K tomu poslouží Freemanův řetězový kód [1]. Čili výsledný simulátor nabízí samotný prostor pro testování výsledků hledání, PID regulace a následnou finální navigaci robota.

Převod do šedotónové oblasti Převod každé buňky obrázku do šedotónové oblasti se provede pomocí následujícího vzorce :

Vzorec pro převod do šedotónové stupnice:

$$gray_{R,G,B} = R \cdot 0.299 + G \cdot 0.587 + B \cdot 0.114 \tag{4.1}$$

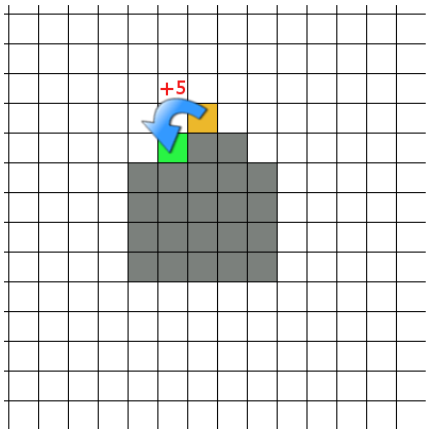
Freemanův řetězový kód:
Směry *Freemanovy* růžice jsou značeny čísly 0-7 (viz obr. 4.3).



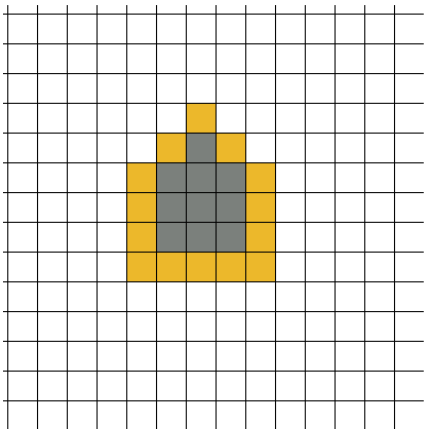
Obrázek 4.3: Směry Freemanova řetězového kódu

Algoritmus řešení nalezení hranice:

1. Procházíme body vstupního obrázku (resp. matici reprezentující obrázek) a hledáme první bod (viz obr. 4.4) popředí (resp. bod reprezentující danou křivku). Tento bod si zapamatujeme.
2. Nastavíme výchozí hodnotu proměnné reprezentující směr procházení na 5.
3. K poslednímu nalezenému směru přičteme (viz obr. 4.4) hodnotu 5 (výslednou hodnotu musíme dělit modulo 8).



Obrázek 4.4: Přičtení hodnoty +5 k původnímu nalezenému bodu popředí.



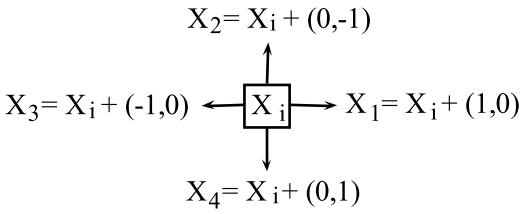
Obrázek 4.5: Výsledná nalezená hranice.

4. Postupně funkcí procházíme všechny možné směry, dokud nenalezneme další bod popředí a uložíme výsledný směr do spojového seznamu.
5. Po nalezení nového bodu popředí si tento bod zapamatujeme a opakujeme krok 3.
6. Kroky 3, 4 a 5 opakujeme tak dlouho, dokud nedosáhneme původního bodu (viz obr. 4.5) popředí (viz krok 1). Výsledný Freemanův kód uložený ve spojovém seznamu nám udává požadovanou cestu, po které budeme robota navádět.

4.2 Výpočet cesty

U všech plánovacích strategií při hledání vyhovujících sousedů prohledáváme ve smyslu čtyř-okolí (viz obr. 4.6), tzn. můžeme postupovat k sousedům ver-

tikálním a horizontálním směrem, neuvažuji možnost diagonálního přechodu. Ve skutečnosti je toto řešení také rychlejší a používanější, není proto důvod implementace procházení ve smyslu osmi-okolí, navíc ve výsledku dochází k vyhlazení cesty, kde se jednotlivé pozice bodů mohou posunout na lépe vyhovující, jež mohou vést přes pozice sousedství bodů ve smyslu osmi-okolí.



Obrázek 4.6: Procházení sousedních pozic ve smyslu čtyř-okolí.

Pohybová množina:
Z výše uvedeného obrázku (viz obr. 4.6) sestavíme tedy množinu pohybových vektorů robota D.

$$D = \{(0,-1),(1,0),(0,1),(-1,0)\}, \tag{4.2}$$

při prohledávání je poté tato množina využita a umožňuje expanzi sousedních bodů bodu X_i.

4.2.1 Dijkstrův a $A^\star$ algoritmus

Dijkstrův a stejně tak algoritmus $A^\star$ plánují cestu z výchozí pozice $s_{start} \in s$. Algoritmus řešení je poměrně známý, uvedu proto jen stručný popis našeho řešení:

- v každé iteraci si ukládáme aktuální cenu pozice s_i od počáteční pozice a prozkoumáme okolí akt. zkoumaného bodu.
- Poznamenáme v mapě pozice těchto prozkoumaných bodů $s_{i_x} \subseteq S_{free}$ a přidáme do seznamu pozici s_{i_x} s nejmenším ohodnocením funkce $f(s)$ (viz. 2.2).
- Seznam seřadíme od nejmenšího ohodnocení pomocí prioritní fronty[1] s vložením nového prvku o složitosti O(n) a vybereme právě prvek

[1]Používána velmi rychlá a kvalitní implementace fronty od Alexey Kurakina.

s nejnižším ohodnocení a celý proces opakujeme do nalezení cílové pozice. Při vybrání sousedního bodu s nejnižší hodnotou si také uložíme akci nutnou k vykonání expanze tohoto vrcholu. Tato akce bude později důležitá pro nalezení výsledného seznamu bodů vedoucích k cíli.

Heuristické funkce:
Pro algoritmus $A^\star$ a testování jeho vlastností byly implementovány čtyři heuristické funkce. Každé pozici $s_i = (x_i, y_i) \in s$ náleží hodnota funkce $h(s_i)$, která udává odhad ceny cesty k cílové destinaci.

1. Heuristická formule Manhattan :

$$h(s_i) = cost \cdot (|x_i - x_{goal}| + |y_i - y_{goal}|) \tag{4.3}$$

2. Diagonální zkratka :

$$\begin{aligned} h_{diag} &= Min\{|x_i - x_{goal}|, |y_i - y_{goal}|\} \\ h_{straight} &= (|x_i - x_{goal}| + |y_i - y_{goal}|) \\ h(s_i) &= 2 \cdot cost \cdot h_{diag} + cost \cdot (h_{straight} - 2 \cdot h_{diagonal}) \end{aligned} \tag{4.4}$$

3. Maximum rozdílu dx, dy:

$$h(s_i) = cost \cdot Max\{|x_i - x_{goal}|, |y_i - y_{goal}|\} \tag{4.5}$$

4. Formule Eukleidových vzdáleností (bez odmocniny):

$$h(s_i) = cost \cdot ((x_i - x_{goal})^2 + (y_i - y_{goal})^2) \tag{4.6}$$

5. Formule Eukleidových vzdáleností:

$$h(s_i) = cost \cdot \sqrt{(x_i - x_{goal})^2 + (y_i - y_{goal})^2} \tag{4.7}$$

Výsledek heuristik si můžeme prohlédnout níže, pro mapu o rozměrech 5×5 a s cílovou destinací uprostřed mapy. Pro různý typ map a překážek mohou vycházet lépe různé typy heuristik, čtenář si jistě udělá obrázek sám podle následujících příkladů (cena přechodu mezi buňkami nastavena na dva):

$$1) = \begin{bmatrix} 8 & 6 & 4 & 6 & 8 \\ 6 & 4 & 2 & 4 & 6 \\ 4 & 2 & 0 & 2 & 4 \\ 6 & 4 & 2 & 4 & 6 \\ 8 & 6 & 4 & 6 & 8 \end{bmatrix} \tag{4.8}$$

$$2) = \begin{bmatrix} 8 & 6 & 4 & 6 & 8 \\ 8 & 4 & 2 & 4 & 6 \\ 4 & 2 & 0 & 2 & 4 \\ 6 & 4 & 2 & 4 & 6 \\ 8 & 6 & 4 & 6 & 8 \end{bmatrix} \tag{4.9}$$

$$3) = \begin{bmatrix} 4 & 4 & 4 & 4 & 4 \\ 4 & 2 & 2 & 2 & 4 \\ 4 & 2 & 0 & 2 & 4 \\ 4 & 2 & 2 & 2 & 4 \\ 4 & 4 & 4 & 4 & 4 \end{bmatrix} \tag{4.10}$$

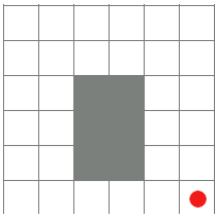
$$4) = \begin{bmatrix} 16 & 10 & 8 & 10 & 16 \\ 10 & 4 & 2 & 4 & 10 \\ 8 & 2 & 0 & 2 & 8 \\ 10 & 4 & 2 & 4 & 10 \\ 16 & 10 & 8 & 10 & 16 \end{bmatrix} \tag{4.11}$$

Nalezení seznamu pozic trasy:
Po dosažení cílové destinace algoritmem je nutné nalezenou cestu uložit, neboť algoritmus v průběhu expanze jednotlivých pozic mapy může vést do slepé uličky, načež vybral jiný předchozí lépe vyhovující bod cesty (dává tedy vzniku jistých možných odboček, které nevedou k cíl) a my na těchto odbočkách jednoduše nevíme, která cesta je správná. Výslednou cestu tedy nalezneme postupným reverzním zpracováním cesty z cílové destinace – na každé pozici si totiž ukládáme směr ze kterého jsme se na danou pozici dostali, jednoduše tedy zpětně dojdeme do poč. bodu trasy.

4.2.2 Dynamické programování

Tento algoritmus hledání cesty již bude u čtenáře jistě méně známý, proto se u něj trochu zastavíme.

Stručně řečeno, algoritmus prochází všechny buňky mapy, jež je možné navštívit a rekurzivně počítá nejmenší hodnotu pro danou buňku na základě sousedních buněk. Vychází z cílové pozice, jejíž hodnota je nastavena na nulu a všechny ostatní buňky jsou zpočátku nastavena na zápornou hodnotu jako nenavštívené, po konvergenci algoritmu bude každá buňka mapy ohodnocena váhou (krom překážek) a budeme tedy znát cestu z libovolné pozice bez dalších potřebných výpočtů.



Obrázek 4.7: Výchozí pozice algoritmu dynamického prohledávání.

Vysvětleme si průběh algoritmu podrobněji na příkladu:

- Jak bylo řečeno, algoritmus vychází z cílové destinace a jedná se ve svém principu o rekurzivní algoritmus semínkového plnění [14]. Hodnota cílové pozice je nastavena na *0*, je vytvořena pomocná matice M, která bude po konvergenci algoritmu obsahovat směry cesty, a všechny neprozkoumané body na mapě jsou zpočátku označeny hodnotou *-1*. Dále potřebujeme matici ohodnocující jednotlivé body mapy, evaluační funkce pro bod $X_i = (x_i, y_i)$ je poté dána předpisem :

$$f(x_i, y) = minf(x^{'}, y^{'}) + cena, \tag{4.12}$$

  kde $(x^{'}, y^{'})$ jsou souřadnice sousedních bodů bodu X_i.

- Rekurzivně procházíme celou matici obrázku do té doby, dokud docházelo ke změnám minim v okolí všech bodů $X_i \in S_{free}$. Byla-li cena přechodu mezi jednotlivými body nastavena na 1, bude výsledná ohodnocená matice pozic po konvergenci, která odpovídá výše uvedenému obrázku (viz obr. 4.7), vypadat následovně:

$$\begin{bmatrix} 10 & 9 & 8 & 7 & 6 & 5 \\ 9 & 8 & 7 & 6 & 5 & 4 \\ 8 & 7 & -1 & -1 & 4 & 3 \\ 7 & 6 & -1 & -1 & 3 & 2 \\ 6 & 5 & -1 & -1 & 2 & 1 \\ 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix} \tag{4.13}$$

- Rozhodujeme-li se na závěr k výběru cesty z námi požadované pozice do cílové destinace, vybíráme sousedy s $\leq$ hodnotou, než je aktuální hodnota dané pozice. Více se dozvíte dále.

Nalezení seznamu pozic cesty:
Protože při průběhu algoritmu jsme si již ukládali k jednotlivým bodům směr cesty k sousedovy s nejnižší hodnotou evaluační funkce $f(X_i)$, dané nejvýhodnější vypočtené směry (resp. pohybový vektor) směry již máme uložené v matici M.

Ve výsledku stačí projít matici M. Matici tedy procházíme od pozice s_{start} a na všechny pozice s_i aplikujeme pohybový vektor uložený v této matici. Tímto postupem se dostaneme z výchozí pozice s_{start} do naší cílové destinace s_{end}. Každý následující bod cesty s_{i+1} je vypočten jako: $s(i+1) = s(i) + M(d(i))$.

Stochastický pohyb

Jak již bylo napsáno v teoretickém úvodu v sekci 2.1.4 na straně 10, cílem následné úpravy algoritmu *dynamického programování* je vzít v potaz vlastnosti stochastického pohybu robota a plánovat cestu, jež nevede podél překážek, ale bude si udržovat odstup.

Zavedeme si tedy $P_{suc}(a_i|X_i, X_{i-1}) = 0.8$, jež nám udává pravděpodobnost, s jakou se robot objeví na požadované pozici X_i z předchozí pozice X_{i-1} vykonáním určité akce $a_i \in a$. Protože tato akce není deterministická, nemůžeme předpovědět, zda došlo ke správnému vykonání akce a_i, kde např. mohly robotu podklouznout kola a místo požadovaného vyústění akce a_i (řekněme jízdy vpřed) se robot vyskytl s pravděpodobností $P_{miss} = (1 - P_{suc})/2$ na místě vpravo, či vlevo od požadované pozice. Rozšíříme tedy originální algoritmus dynamického programování o danou skutečnost :

- Nejprve si přeznačíme pozice $X_j \in s$, s cenou $f(X_j) = 1000$ (typicky v programu uvedené jako šířka x výška matice).
- Dále postupujeme opět od cílové destinace X_{goal} a ohodnocujeme jednotlivé buňky matice (pouze ty na které lze vstoupit a nevyskytuje se překážka) podle následující funkce :

$$\begin{aligned} f(X_i(x,y) \times a_j) = P_{suc} \cdot f(X_i(x,y-1)) \\ +P_{miss} \cdot f(X_i(x+1,y)) \\ +P_{miss} \cdot f(X_i(x-1,y)) + cost, \end{aligned} \tag{4.14}$$

  kde $a_{j=1}$ je řekněme akce odpovídající posunu robota nahoru. Hodnota výsledné ceny buňky je tedy ovlivněna také hodnotou vah jeho sousedů po levé, resp. pravé, straně. Takto je vypočtena váha pro všechna $a_i \in a$. Stejným způsobem je vypočtena také váha pro ostatní směry pohybu a_2, a_3, a_4 a výsledné $f(X_i)$ je poté vypočteno jako :

$$f(X_i) = Min_{\forall j}\{f(X_i(x,y) \times a_j)\}, \tag{4.15}$$

  nachází-li se tedy v bezprostřední blízkosti překážka, hodnota funkce f pro daný bod vzroste právě díky možné nepřesnosti pohybu a v sousedství se poté bude nalézat výhodnější bod, který směřuje dále od překážky (viz obr. 2.2).
- Po konvergenci algoritmu je směr cesty vybírán opět obdobným způsobem jako u klasického řešení algoritmu dynamického programování. Nicméně je zde brán v potaz právě nedeterministický pohyb robota s jistou pravděpodobností. Více se dozvíte níže.

Poupravený[2] příklad uvedený v předchozí kapitole s počátečním ohodnocením (pokutou) 1000 bude po konvergenci algoritmu obsahovat matici s následujícími hodnotami (zaokrouhleno na celá čísla) :

$$\begin{bmatrix} 632 & 490 & 394 & 302 & 222 & 242 & 424 \\ 539 & 463 & 387 & 292 & 168 & 157 & 223 \\ 592 & 579 & 1000 & 1000 & 128 & 115 & 144 \\ 663 & 660 & 1000 & 1000 & 107 & 90 & 116 \\ 733 & 726 & 1000 & 1000 & 113 & 64 & 134 \\ 811 & 769 & 805 & 607 & 210 & 0 & 285 \end{bmatrix} \tag{4.16}$$

[2]přidán navíc sloupec s volným prostorem, abychom měli lépe odlišen vnitřní prostor mezi hranicí objektu a překážkami.

Samozřejmě implementace je řešena vcelku jednoduchým způsobem a jistě existují komplikovanější a rychlejší implementace řešení, avšak dosáhnout závratných rychlostí nebylo cílem a vzhledem k principu, jak algoritmus funguje, bude počáteční výpočetní náročnost vždy velká. Smysl použití je jasný, neboť po počáteční, byť výpočetně náročné operaci máme pro dané prostředí k vytyčenému cíli stále připravenu z jakéhokoliv bodu cestu, kterou je možné ihned následovat.

Nalezení seznamu pozic cesty:
Nalezení cesty je poté jednoduché, v samotném algoritmu při vážení jednotlivých pozic obrázku také ukládám matici M, kde každé pozici S_{free} této matice náleží index z množiny pohybových směrů D. Tento index nám dává směr $d_i \in D$, kterým se z dané pozice vydat k cílové destinaci. Zde vás možná zaráží vzhledem k výše uvedeným skutečnostem a smyslu tohoto algoritmu, proč máme blízko překážek relativně nízké hodnoty, probíhal-li by výběr směru cesty stejným způsobem jako u klasické implementace algoritmu dynamického plánování, poté by výsledná cesta vedla podél překážek.
Vězte, že hodnoty jsou inteligentně spočteny a při následném výběru cesty se uvažují právě sousedé, kteří následný výběr směru cesty změní. Pro pořádek si uveďme opět příklad řešení směru cesty na konkrétní pozici z výše uvedené vypočtené matice cen (4.16) :

- Uvažujme bod na pozici $s(x = 5, y = 3)$, cena na této pozici je poté $f(s(5, 3)) = 128$.
- Výsledný směr cesty by měl za předpokladu klasického způsobu dynamického programování vést podél překážky na pozici $s(x = 5, y = 4)$ s cenou $f(s(5, 4)) = 107$. Nicméně jednotlivé hodnoty matice M jsou spočteny s využitím nedeterministického pohybu robota, proto také samotný výběr cesty musí tento pohyb uvažovat.
- Pohybujeme-li se tedy směrem dolů (a_2) podél překážky, bude mít zmíněný pohyb z pozice $s(5, 3)$ při 50% pravděpodobnosti správnosti pohybu[3] hodnotu $f_p(a_2) = 332, 25$, neboť při nedeterministickém pohybu je třeba brát v potaz, že namísto pohybu dolů – pozice $s_0(5, 4)$ robot mohl s 25% pravděpodobností vykonat pohyb vlevo/vpravo a objevit se tedy na pozici $s_1(4, 3)$ (resp. $s_2(6, 3)$). Je třeba uvažovat také tyto pozice ve výběru posunu robota :

[3]Respektive s jakou pravděpodobností lze uvažovat, že daný pohyb opravdu proběhne.

$$\begin{aligned} f_p(s_0 \times a_2) &= 0,5 \cdot f(s_0) + 0,25 \cdot f(s_1) + 0,25 \cdot f(s_2) \\ &= 0,5 \cdot 107 + 0,25 \cdot 1000 + 0,25 \cdot 115 \\ &= 332,25 \end{aligned} \tag{4.17}$$

- Můžeme jistě najít jiný pohyb s menší hodnotou, zde konkrétně nejmenší hodnotu bude mít pohyb vpravo, a to sice hodnotu $f_p(a_3) = 126,25$.

Ve výsledku tedy stačí opět projít matici M, ve které již máme uložené nejvýhodnější vypočtené směry (resp. pohybový vektor). Matici procházíme stejným způsobem jako je již uvedeno v popisu řešení klasického algoritmu dynamického programování.

4.2.3 Úprava cesty

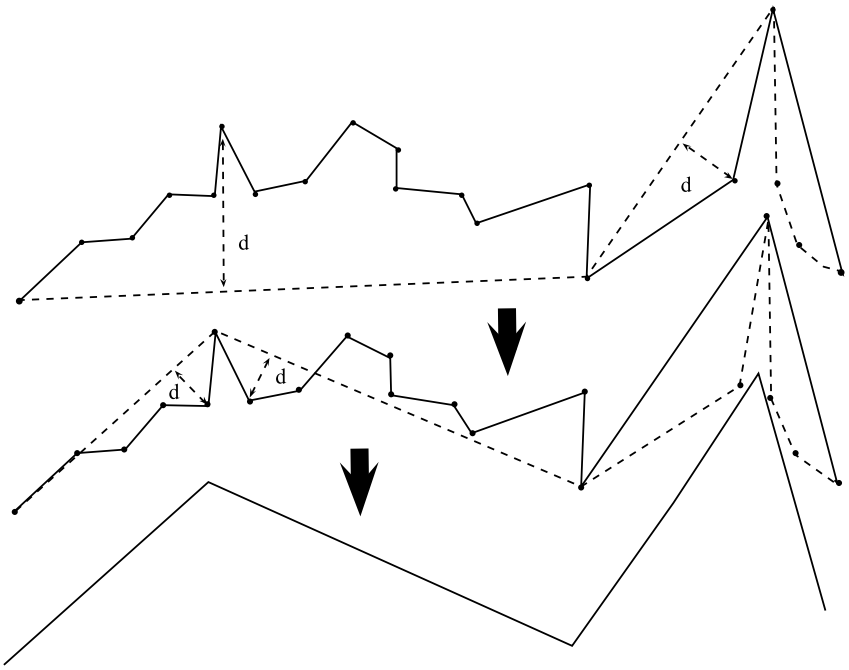
Následuje jednoduchá a výpočetně nenáročná úprava nalezené cesty.

Douglas-Peucker redukce

K redukci bodů na ceste je používán výše zmíněný tzv. *Douglas-Peucker* algoritmus. Pro podrobné informace doporučuji nahlédnout do literatur [8][9]. Základním principem algoritmu je, že rekurzivně dělí křivku na segmenty mezi prvním a posledním bodem, počáteční délka segmentu je vždy vybrána náhodně, proto, jak si můžete povšimnout, v simulátoru po opakování výběru cesty mezi body může být nová cesta poněkud jiná, neboť byly prořezány jiné body na cestě. Bod v segmentu s nejvyšší chybou d ponechá na cestě a zkoumá chybu vzdáleností d mezi daným bodem a zvoleným počátkem. V každém takovém segmentu prozkoumá, zda jsou body v menší vzdálenosti než d. Pokud ano, může takový bod vymazat, neboť po zjednodušení křivky nebude daná chyba horší jak vybrané d. V každém segmentu vybere bod s největší chybou, který ponechá v cestě a poté odstraní všechny body v daném segmentu, jež mají menší, než stanovenou chybu d (viz obr. 4.8).

Vyhlazení cesty

Samotná redukce bodů je dělána kvůli následnému vyhlazení bodů, pokud

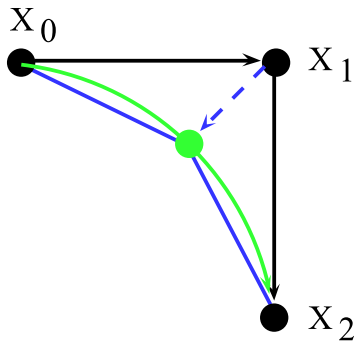


Obrázek 4.8: Postup prořezávání bodů algoritmu *Douglas-Peucker*.

bychom měli moc bodů na cestě, vyhlazení by spočívalo v ideálním nastavení sousedních bodů vůči okolí (jak se také dále dozvíte), nicméně ve výsledku by cesta byla sice plynulejší, ale stále by byla nepěkně kostrbatá díky vlastnostem algoritmů plánování a hustotě generované cesty. Pokud však pustíme algoritmus vyhlazení cesty na originální cestu, bude výsledek cesty stejný, neboť dojde k nepatrnému posunutí „mezibodů“. Toto posunutí díky hustotě bodů bude v rámci desetinného místa a po opětovném zaokrouhlení na celá čísla (z důvodu umístění bodu na mřížku mapy) bude výsledná cesta stejná.

Jak tedy vyhladíme vzniklou cestu? Iterativně upravujeme pozice bodů na cestě dvěma směry (viz obr. 4.9). Jak je tedy vidět na tomto obrázku, úkolem je provést redukci vzdáleností mezi daným bodem a jeho sousedy.

Redukovanou vzdálenost vypočteme tak, že odečteme bod X_1 od každého z jeho sousedů, tedy :



Obrázek 4.9: Itarace vyhlazení cesty. Jak pohybujeme bodem X_1, redukujeme tím také vzdálenost mezi tímto bodem a jeho sousedy (délka modré čáry se zmenšuje). Úkolem je tedy provést „rozumnou“ redukci této modré čáry.

$$\begin{aligned} X_1 &= X_1 + \alpha((X_0 - X_1) + (X_2 - X_1) \\ &= X_1 + \alpha((X_0 + X_2) - 2X_1), \end{aligned} \tag{4.18}$$

kde α udává váhu takovéto redukce (jak rychle se pohybujeme z místa daného bodu (X_1). Jak jistě tušíte, pokud bychom toto opakovaně aplikovali na naší cestě, skončíme přímkou mezi body X_0 a X_1, což je špatně a takový výsledek nechceme, nebot' může vést přes překážky na cestě. Potřebujeme tedy obdobným způsobem působit proti váženým bodům, abychom „vybalancovali“ takovéto nepřípustné vážení cesty. Označme bod, jež se pohybuje ve smyslu minimalizace vzdálenosti místo X_1 jako Y_1. Upravme proto rovnici 4.19 podle této skutečnosti :

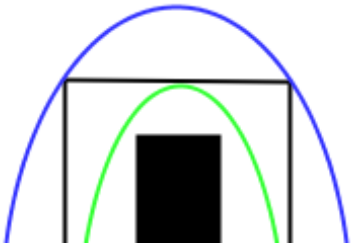
$$Y_1 = Y_1 + \alpha((X_0 + X_2) - 2Y_1) \tag{4.19}$$

Vyhlazené, plynulé cesty – vyznačeno zeleně (viz obr. 4.9) docílíme tedy dalšími body, že budeme naším bodem Y_1, pohybovat proti bodu s původní pozicí s vyšší váhou, jež nám zabrání v tom, aby výsledná cesta byla přímka:

$$Y_1 = Y_1 + \beta(X_1 - Y_1) \tag{4.20}$$

, kde váha β vyjadřuje, jak rychle se pohybujeme ve směru naší původní pozice. Výsledné 4.19 a 4.20 iterativní rovnice aplikujeme do doby, dokud dochází ke změně pozice do nějaké předem daném malém rozdílu hodnot mezi novou a původní pozicí.

Tento způsob řeší vyhlazení cesty, nicméně vyskytuje se zde další problém, a to sice ten, že objíždíme-li překážku, tak s výše zmiňovanou metodou sice docílíme vyhlazenější plynulé cesty, nicméně velice nebezpečně se tímto přiblížíme k dané překážce a původní rezerva mezi překážkou pro šíři robota bude smazána (viz obr. 4.10).



Obrázek 4.10: Vyhlazení cesty, zeleně vyznačena vyhlazená cesta, modře potom cesta, kterou bychom si jistě raději přáli.

Potřebujeme tedy zafixovat rohové pozice na původním originálním místě. Toto je řešeno jednoduše přidáním binárního pole o velikosti dané cesty. Při klasickém vyhlazení potom jsou zafixovány ty pozice, jež se nebezpečně přibližovaly k překážce, resp. výsledná pozice byla právě na překážce. Vnější vyhlazení těchto zafixovaných pozic provedeme následovně:

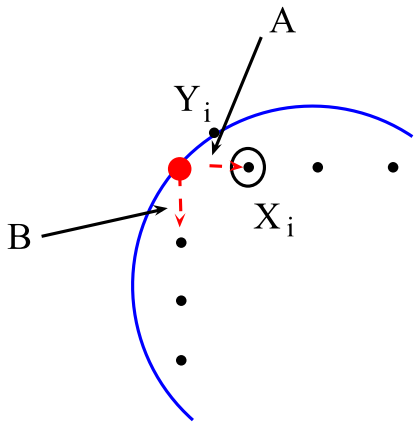
- Potřebujeme, aby vzdálenost vyznačeného bodu X_i (viz obr. 4.11) od hraniční neměnné pozice odpovídala velikosti vektorů $\mathbf{A} = \mathbf{B}$, kde :

$$A = X_{i+1} - X_i \tag{4.21}$$
$$B = X_{i+2} - X_{i+1} \tag{4.22}$$

- Pokud se podíváme na rozdíl mezi vektory $\mathbf{A} - \mathbf{B}$ (4.22 - 4.22) a tuto chybu použijeme k redukci našeho bodu X_i, výsledné rovnice upraveného bodu jsou tedy :

$$\begin{aligned} Y_i &= X_i + \delta \cdot (2 \cdot X_{i-1} - X_{i-2} - X_i) \\ Y_i &= X_i + \delta \cdot (2 \cdot X_{i+1} - X_{i+2} - X_i), \end{aligned} \tag{4.23}$$

kde jako váhu δ použijeme poloviční váhu β, jež byla použita k vyhlazení cesty.



Obrázek 4.11: Docílení vnější vyhlazené cesty vzhledem k zafixované pozici (vyznačena červeně).

Parametry nastavení úpravy cesty:
Shrňme si zde parametry nastavení úprav cest.

- *Douglas-Peucker tolerance* – parametr určující toleranci, resp. velikost vektoru **d**) při prořezávání bodů (viz obr. 4.8).
- *Data weight parameter* – váha β rovnice 4.20.
- *Data smooth parameter* – váha α rovnice 4.19.

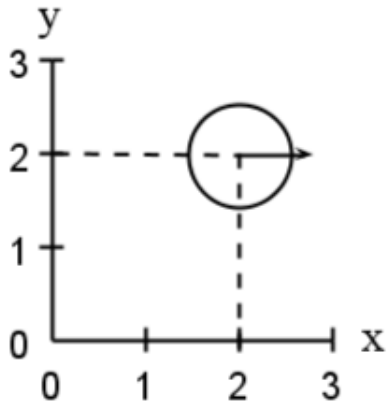
4.3 Modul navigace

Povězme si o způsobu navigace na cestě. Samotný modul navigace robota zahrnuje třídu „*Robot.cs*“ a „*PID.cs*“, ve kterých se nalézají veškeré podpůrné systémy pro správnou orientaci a navigaci po dané cestě.

4.3.1 Kinematika simulovaného robota

Zde si představíme kinematický model [5] agenta a jeho matematický popis. Kinematika je kalkulus popisující efekt kontrolních akcí (jeď dopředu, vpravo, vlevo) na výslednou pozici robota a jeho natočení. Pozice agenta je obvykle popsána šesti hodnotami, tří-dimenzionální kartézskou soustavou souřadnic a tři Eulerovy úhly (úhel vlastní rotace φ, precesní úhel ψ a nutační úhel ϱ) [11].

Prezentovaná práce nicméně počítá s robotickou platformou operující v rovinném prostředí, kinematiku takového robota lze popsat vektorem souřadnic $[x, y, \theta]$ (viz obr. 4.14).

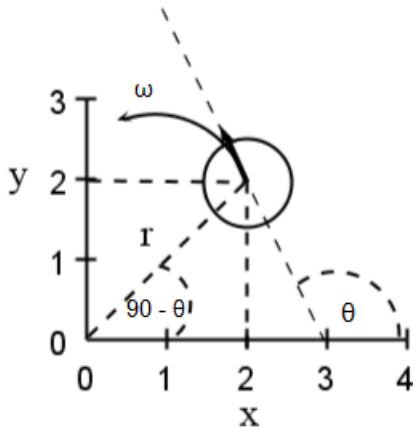


Obrázek 4.12: Znázornění pozice a směru robota. Na vyobrazeném obrázku se nachází robot na pozici (2,2) a jeho pohyb směřuje v ose x, tedy $\theta = 0$, pokud by směřoval ve směru osy y, jeho úhel natočení by byl poté $\theta = 0,5 \cdot \Pi$

Popišme si kinematiku pohybu ideálního robota bez zašuměných dat. Předpokládáme složky rychlosti pohybu v a úhlového zrychlení ω v čase t konstantní, $u_t = (v, \omega)^T$. Pokud jsou obě složky neměnné v celém intervalu měření pro všechna $t \in T$ (interval měření) a úhel natočení $\omega \neq 0$, potom se robot pohybuje v kruhu (viz obr. 4.13), jehož poloměr je :

$$\begin{aligned} r &= \left|\frac{v}{\omega}\right| \\ &= L/tan(\omega), \end{aligned} \tag{4.24}$$

kde L je vzdálenost náprav modelu vozidla a ω úhel natočení.



Obrázek 4.13: Pohyb robota po kružnici o poloměru r.

Rovnice 4.24 nám udává důležitý vztah mezi úhlovým zrychlením ω a rychlosti v robota, jež se pohybuje na kruhové trajektorii s poloměrem r. Nezahrnuje však tato rovnice vztah, kdy se robot pohybuje po přímce. Přímka nicméně koresponduje s pohybem po kružnici, kde předpokládáme $r = \text{inf}$.

Označme dále $x_{t-1} = (x, y, \theta)^T$ jako výchozí pozici robota, předpokládáme konstantní zrychlení $(v\omega)^T$ pro nějaký čas Δt. Střed kružnice potom bude na pozicích :

$$x_c = x - \frac{v}{\omega} sin\theta \tag{4.25}$$

$$y_c = y + \frac{v}{\omega} cos\theta \tag{4.26}$$

Po čase Δt bude náš ideální robot na pozici $x_t = (x', y', \theta')^T$:

$$\begin{aligned} \begin{pmatrix} x' \\ y' \\ \theta' \end{pmatrix} &= \begin{pmatrix} x_c + \frac{v}{\omega}\sin(\theta + \omega\Delta t) \\ y_c - \frac{v}{\omega}\cos(\theta + \omega\Delta t) \\ \theta + \omega\Delta t \end{pmatrix} \\ &= \begin{pmatrix} x \\ y \\ \theta \end{pmatrix} + \begin{pmatrix} -\frac{v}{\omega}\sin\theta + \frac{v}{\omega}\sin(\theta + \omega\Delta t) \\ \frac{v}{\omega}\cos\theta - \frac{v}{\omega}\cos(\theta + \omega\Delta t) \\ \omega\Delta t \end{pmatrix} \end{aligned} \tag{4.27}$$

Tyto rovnice vznikly z jednoduché trigonometrie (viz obr. 4.13). Po čase Δt robot urazí $v \cdot \Delta t$ po kružnici, což způsobí, že se natočí o $\omega \cdot \Delta t$. Ve stejné době jsou potom souřadnice x a y robota popsány kružnicí kolem $(x_c y_c)^T$ a polopřímkou vycházející z bodu $(x_c y_c)^T$ s úhlem $\omega \cdot \Delta t$. Druhá část výrazu potom vychází z 4.26.

Samozřejmě, že ve skutečnosti robot nemůže „skákat“ z jednoho úhlového zrychlení do jiného, proto k plynulému pohybu modelu robota doporučuji v simulátoru rozumné nastavení parametrů a užívat malé změny za stanovený čas Δt. Bystrý čtenář si jistě také všiml, že daný matematický model simulátoru spíše odpovídá modelu *auta*, který dokáže zatáčet jen přední nápravou. Proč tedy nebyl vybrán model tanku, jehož rovnice jsou uvedeny níže? Důvod je prostý, řešení ovládání pohybu robota na platformě Kinbo je vskutku nelogicky implementováno a dochází k již jistému přepočtu na úhlové natočení. Je umožněno pouze ovládání samotné rychlosti a nastavení parametru úhlu natočení. Není možno jednotlivě přímo regulovat rychlosti jednotlivých motorů. Ostatně tyto rychlosti regulujeme právě nastavením daného úhlu, nicméně úhel v podstatě není úhel natočení, neboť platforma nemá *odometrii* a nemůže tedy docházet v žádném případě k požadovanému natočení. Jedná se tedy o velice podivné nastavení, které nedává smysl a ani požadovaný úhel natočení, jen jeho velmi hrubou hodnotu. Nicméně k redukci robota na vytyčené cestě, jak bylo testováno na výše uvedeném modelu s využitím šumů kinematického modelu (více v následující sekci 4.3.2 na straně 41) toto bude stačit. I bez žádoucí *odometrie* s velkým šumem na požadovaný úhel natočení robota si regulátor poradí obstojně – více v sekci 6.2 na straně 62. Způsob ovládání vozidla s využitím platformy Kinect se tedy spíše podobá výše uvedenému modelu, ve skutečnosti dostaneme po každé akci robota výsledné pozice a úhel natočení θ. Výsledné pozice a úhel natočení po provedení požadované akce robota budou vyčteny z modulu lokalizace, načež bude vykonaná příslušná akce pro zachování požadovaného směru – o tom, ale až v následující sekci 4.3.3 na straně 44. Nicméně pro pozdější zpřesnění navigace na reálné platformě doporučuji kompletně předělat způsob řešení ovládání robota za využití senzorů pro čtení odometrie kol a pro kontrolu aktuátorů využít níže uvedených vzorců. Toto bohužel není jediný problém platformy, více se dočtete v sekci 5.1 na straně 53.

Model tanku: Pro úplnost tedy dodejme rovnice tankového modelu. Pokud je rozdíl rychlostí mezi nápravami nenulový, ale obě se točí na stejnou stranu, pohybuje se i tento robot po kružnici. Vzdálenost středu této kružnice je dána poměrem obou rychlostí – vL a vR:

$$r = \frac{\frac{1}{2}b \cdot (vL + vR)}{vL - vR}, \tag{4.28}$$

kde b je vzdálenost kol od sebe.

Pokud levé „kolečko“ pásu ujede vzdálenost dL a pravé dR, změní se orientace o úhel θ:

$$\theta = \frac{(dL - dR)}{b} \tag{4.29}$$

Celková ujetá vzdálenost $d = (dL + dR)/2$. Z daného modelu plyne ponaučení – natočení robota s tankovým podvozkem závisí pouze na rozdílu celkové ujeté vzdálenosti pravého a levého pásu (kolečka) a nikoliv na průběhu dílčích změn. K výpočtu tedy stačí i jednoduchý čítač pro každé kolo (senzor) a není třeba využití goniometrických funkcí.

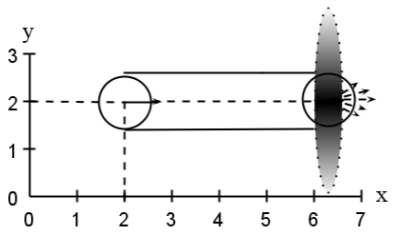
4.3.2 Reálný pohyb robota

Reálný pohyb robota je však vždy zatížen šumem vzniklý z nepřesnosti dat měření pozice, ať už vlivem naakumulované chyby *odometrie*, šumem měřicích senzorů, či např. na naší platformě naprosté absence *odometrie* a již výše zmíněných problémů.

Takto vzniklý šum, který může ovlivňovat nepřesnost ujeté vzdálenosti (nepřesnost akt. pozice) a úhel natočení robota, je modelován s využitím Gaussova (normálního) rozdělení. Reálné v a úhlová rychlost robota ω se tedy mohou (a v praxi jistě budou) lišit od námi požadovaných zadaných rychlostí. Budeme tedy tuto chybu modelovat jako:

$$\begin{pmatrix} v' \\ \omega' \end{pmatrix} = \begin{pmatrix} v \\ \omega \end{pmatrix} + \begin{pmatrix} \epsilon_1\sigma_1 + \sigma_2 v \\ \epsilon_2\sigma_3 + \sigma_4\omega\omega' \end{pmatrix} \tag{4.30}$$

Zde potom ϵ_i je chyba se střední hodnotou v bodě 0 s rozptylem σ_i. V našem modelu je rozptyl chyby závislý na požadovaném úhlu zrychlení robota, $\sigma_1 - \sigma_4$ modelují přesnost robota. Čím jsou tyto hodnoty větší, tím méně přesný je výsledný pohyb robota. Tato chyba se dá modelovat Gaussovým (normálním) rozdělením.

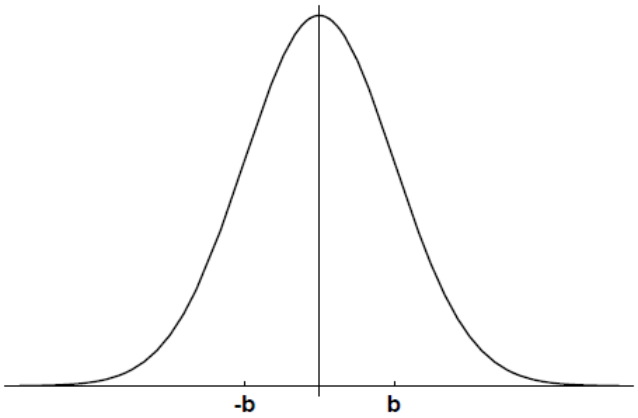


Obrázek 4.14: Nepřesnost pohybu robota.

Normální rozdělení:

- Normální rozdělení (viz obr. 4.15) s rozptylem b je poté dáno předpisem :

$$\epsilon_i = \frac{1}{\sqrt{2\Pi \cdot b}} \exp^{\frac{-1}{2}\frac{a^2}{b}} \tag{4.31}$$



Obrázek 4.15: Normální rozdělení.

Možnosti nastavení modelu:
Představme si veškeré implementované parametry modelu, jež ovlivňují vlastnosti chování robota. Všechny parametry jsou zakomponovány do matematického modelu robota.

- **Maximum steering angle.** Nastavení, do jakého úhlu můžeme natočit kola robota. Tento parametr má vliv na celkový možný poloměr zatočení a ovlivňuje tím přesnost setrvání robota na trase. Zde musí být uživatel velmi opatrný, neboť velký úhel natočení má poté negativní vliv na regulaci pohybu a může docházet k přílišným výkyvům ve snaze dostat se na trasu při prudkých změnách směru jízdy.
- **Robot speed.** Udává, jak rychle se robot může pohybovat. Rychlost pohybu je v simulaci dána v rozmezí $r =< 0,05; 1 >$.
- **Steering noise**. Udává chybu zatočení robota modelovanou normálním rozdělením. Je-li chyba nastavena např. na 10 stupňů, pokud chce robot zatočit plným maximálním úhlem natočení kol 45°, poté bude vzhledem k nastavenému šumu výsledná hodnota natočení v rozmezí $< 35; 55 >$ stupňů.
- **Distance noise.** Odchylka ujeté vzdálenosti. Je-li rychlost vozidla nastavena na 1, ujede robot za čas t právě danou vzdálenost (v simulaci při daném nastavení právě o jednu pozici buňky mapy). Při nezáporné odchylce je tento šum započítán do ujeté vzdálenosti a může se opět lišit o danou odchylku.
- **Wheel drift.** Přidání chyby ke geometrii kol. Nastaví neustálého stáčení kol v požadovaném úhlu.
- **Robot length.** Celková délka robota (kolik buněk matice zabere robot svou délkou). Délka robota má také vliv na poloměr zatočení robota.
- **Robot width.** Šířka robota.
- **Robot start angle.** Vychýlení úhlu robota od směru trasy na počáteční pozici.

4.3.3 PID

I hlavně díky simulátoru a možnosti nasimulovat chování regulátoru s různým nastavením chyb a odchylek vozidla při pohybu se nám díky skvělé literatuře [5] podařilo vytvořit velmi rozumně chovající PID regulátor, který byl následně poupraven a vylepšen. Nyní dává mnohem lepší výsledky než standardní regulátor definovaný 2.8, kde $e(t)$ je odchylka robota od trasy. Hlavní změna tkví v *Proporcionální* složce regulátoru. Kde mimo odchylky aktuální polohy robota od cesty bereme také předchozí úhel natočení kol vozidla, touto změnou zabráníme nebezpečnému přetočení robota při prudkých změnách směru jízdy a tím dojde k daleko čistější a rozumnější regulaci. Jedná se ve svém principu „proporcionálně-diferenční“ regulaci, nicméně nebere v potaz změnu odchylky od trati, jak tomu klasicky bývá u PD regulátoru 2.7, nýbrž srovnává původní zatočení kol vzhledem k aktuálním požadavkům trasy. Průměrné chyby a srovnání „klasického“ způsobu PID regulace založeného pouze na regulování trasy na základě odchylky od trati a mé navrhované změny naleznete v následující kapitole v sekci 6.2 na straně 62. Samozřejmě u reálného robota je třeba výslednou chybu porovnat s klasickou implementací *PID* regulátoru, který je také implementován a dává uživateli na výběr.

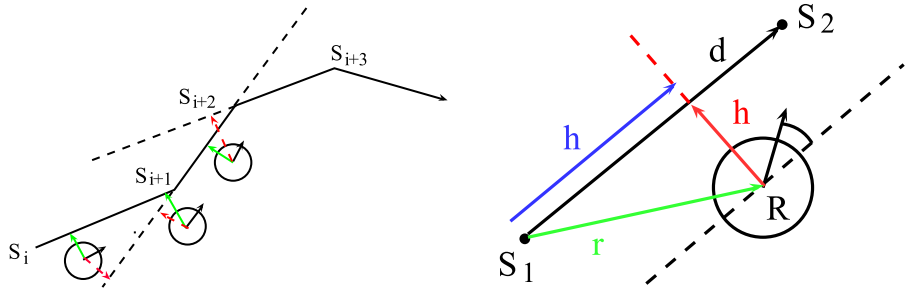
Výsledná rovnice PID regulace vypadá následovně:

$$X(t) = -e(t) - \alpha \cdot (X_{t-1} - e(t)) - \beta \cdot \frac{de(t)}{dt} - \gamma \cdot \int e(t)dt, \tag{4.32}$$

kde X je poté výsledné nutné zatočení robota vzhledem k chybě trasy $e(t)$ v čase t.

Abychom však mohli správně robota navigovat po cestě, je třeba znát odchylku robota od vypočítané cesty (viz obr. 6.1(c)) a také v jakém segmentu mapy (viz obr. 6.1(a)) se robot zrovna nachází, neboť jeho rychlost může být různá a výsledná cesta udává seznam bodů G, jež je nutné následovat. Cesta může být rozdělena do segmentů, kde každý segment představuje vektor spojující dva po sobě jdoucí body ze seznamu. K ohlídání v jakém konkrétním segmentu se robot nachází a výpočtu vychýlení (chyby) od plánované trasy si vypomůžeme lineární algebrou.

Označme si tedy poč. bod segmentu $S_1 = (x_1, y_1)$, a koncový bod daného segmentu trasy jako $S_2 = (x_2, y_2)$ a dále pozici robota $R = (x_r, y_r)$. Poté



(a) Pohyb robota v segmentu, nutná znalost správného přiřazení chyby trasy k odpovídajícímu segmentu (červeně vyznačeno špatné přiřazení odchylky trasy)

(b) Vektory vzdáleností robota v segmentu, jež je nutné vypočítat.

Obrázek 4.16: Pohyb robota po trase.

vektory vzdáleností mezi body $\mathbf{d} = S_1S_2$, $\mathbf{h} = S_1R$ a odchylku $\mathbf{h}$ vypočteme jako :

- Základní výpočty :

$$\begin{aligned} \mathbf{\Delta d_x} &= x_2 - x_1 \\ \mathbf{\Delta d_y} &= y_2 - y_1 \\ \mathbf{\Delta r_x} &= x_r - x_1 \\ \mathbf{\Delta r_y} &= y_r - y_1 \end{aligned} \tag{4.33}$$

- Poté poměr, jakou vzdálenost jsme již vykonali s robotem v daném segmentu, vypočteme jako součin vektorů a vydělíme poměrem celkové vzdálenosti segmentu (normalizovaná velikost = 1) :

$$ratio = \frac{\Delta r_x \cdot \Delta d_x + \Delta r_y \cdot \Delta d_y}{\Delta d_x \cdot \Delta d_x + \Delta d_y \cdot \Delta d_y}, \tag{4.34}$$

  protože takto zapsané velikosti vektorů jsou normalizované; jmenovatel zlomku je tudíž jedna a pokud bude velikost vektoru v čitateli > 1 víme, že se již nalézáme v dalším sektoru bodů cesty.

- Samotnou odchylku od cesty $\mathbf{h}$ vypočítáme následovně:

$$\mathbf{h} = \frac{\Delta r_y \cdot \Delta d_x - \Delta r_x \cdot \Delta d_y}{\Delta d_x \cdot \Delta d_x + \Delta d_y \cdot \Delta d_y} \tag{4.35}$$

Víme-li tedy, v jakém segmentu cesty se nalézáme a jakou máme odchylku od cesty, můžeme aplikovat kinematický model robota společně s PID regulátorem. Pozorný čtenář se mohl v tomto bodě zastavit a říct si : „Jak autor provede navedení robota na cestu, jež je naplánována opačným směrem, než je natočení robota ω “. V praxi se totiž u robotů řeší plánování trasy robota právě vzhledem k jeho natočení, zde navržené řešení vytvoří plán z konkrétní pozice robota. Jak již bylo řečeno v úvodu, navržený systém je zejména mířen pro použití na platformě Kinbo na Západočeské univerzitě a tato platforma je postavena na pásovém podvozku, umožňuje tedy libovolné natočení robota kolem své osy a tím možnost natočit ho právě ve směru jízdy a dále již navigovat výše uvedeným způsobem. Praktické testy však ukázaly, že si však regulátor velice obstojně poradí s počátečním poměrně značným odchýlením natočení robota od směru jízdy – více v sekci sekci 6 na straně 56. V simulátoru je toto natočení spočítáno pomocí funkce *atan2* [19] jakožto úhel natočení vektoru prvního segmentu (mezi body P_0 a P_1) cesty :

$$\begin{aligned} dx &= |x_1 - x_0| \\ dy &= |y_1 - y_0| \\ \theta &= atan2(dy, dx) \end{aligned} \tag{4.36}$$

Poté je již tato odchylka (resp. úhel natočení prvního segmentu cesty) nastavena jako počáteční úhel natočení robota. Simulátor následně umožňuje až 45° odchýlení od vypočteného směru cesty. Při navigaci byla také navrhnuta nízkoúrovňová metoda, která hlídá vzdálenost robota od překážek. Jednoduchými úpravami může být tato vzdálenost upravena a při přiblížení reálného robota příliš blízko překážky ho zastavit a cestu přepočítat, či vytvořit další nízkoúrovňový systém, který se již bude starat o dodržování dostatečné vzdálenosti o překážky a poté navedení na naplánovanou trasu. Nicméně při vhodně zvolených parametrech cesty (více v sekci 6 na straně 56) a samozřejmě nastavení reálné velikosti robota vzhledem k velikostem mapy, dochází k plánování trasy s dostatečnou vzdáleností od překážek a při dobře nastaveném PID regulátoru tedy ve statickém prostředí nemůže dojít ke kolizi.

Dále nastává otázka, jak nastavit PID regulátor, respektive jak provést nastavení vah α, β a γ (viz 2.8). Ukazuje se, že tato skutečnost může být velkým oříškem, avšak je rozhodující na správnou funkcionalitu regulátoru. V praxi se na nastavení těchto parametrů používá buď způsob *brute-force*, nebo námi implementovaný níže uvedený algoritmus *Twiddle*.

Brute-force spočívá v prozkoumání všech kombinací vah v nějakém rozumném definovaném rozpětí. Toto řešení je vcelku nešťastné a výpočetně neefektivní, proto byl implementován velice chytrý trénovací algoritmus.

Postup algoritmu:

- Zpočátku jsou všechny parametry regulátoru nastaveny na 1 : $PID_{\alpha,\beta,\delta} = [1, 1, 1]$.
- Dále se spočítá chyba pro referenční předem nastavené hodnoty parametrů α, β, δ.
- Následně algoritmus zkouší vychylovat v kladném a záporném směru jednotlivé parametry. Pokud dojde ke zlepšení (poklesu) střední kvadratické odchylce (viz 4.3.3) od dané cesty robota, jsou uloženy nové parametry a dochází k opětovném cyklu parametru s menším vychýlení kolem nově nalezených parametrů. Ty jsou následně opět uloženy. Je-li již pokles chyby pod definovanou mezní hodnotou změny ϵ_{err}, algoritmus končí a vrací vypočtené hodnoty.

$$MSE = \frac{1}{N}\sum_{i=1}^{n} \mathbf{h_i}^2, \tag{4.37}$$

kde n je počet kroků robota na cestě.

Algoritmus byl následně upraven, aby lépe vyhovoval našim podmínkám simulátoru a podával obecně kvalitnější výsledky. Každé cestě, ve které robot naboural či nedoběhne do cílové destinace, je definována nadstandardní chyba. Jinak se totiž může stát, že algoritmus „přetrénuje“ data, přičemž bude docházet, že robot se po rozjetí začne točit na jednom místě v bodě trasy, tím vznikne velmi malá odchylka a také nenarazí do žádné překážky.

4.4 Platforma Kinbo

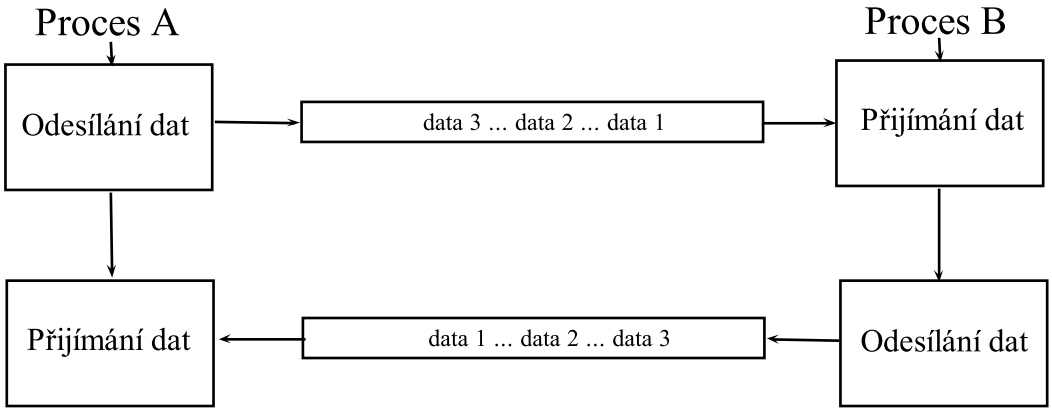
Tato kapitola a celé programové vybavení slouží zejména k dalšímu rozšíření platformy a bylo vytvořeno mimo zadání práce, ale i jako pomoc při daném

rozšíření a velmi podstatný základ řešení se snadnou budoucí integrací modulu navigace.

Nejdříve čtenáře seznámíme s navrženou funkční klientskou částí postavenou na platformě Kinbo, umožňující přenos scény přes WiFi a manuální navigaci vozidla. Dále také s navrženou formou lokalizace s využitím knihovny *MRPT* a její možnosti integrace do stávajícího řešení platformy a v následující kapitole si také uvedeme nutné změny platformy (více v sekci 5.1 na straně 53), jež jsou třeba před plnou integrací modulu navigace společně s řešením lokalizace vykonat.

4.4.1 Klientská část

Klientská část zahrnuje modul založený na softwarovém API platformy Kinbo a návrh uživatelského rozhraní. Je umožněn manuální pohyb robota pomocí šipek klávesnice a příjem dat z kamery klientovi přes WiFi. Projekt dále zahrnuje metody a funkční propojení s knihovnou *MRPT* a přenos dat mezi dvěma nezávislými procesy pomocí pojmenovaných rour [21] (viz obr. 4.17).



Obrázek 4.17: Sdílení dat mezi procesy.

4.4.2 Návrh modulu lokalizace

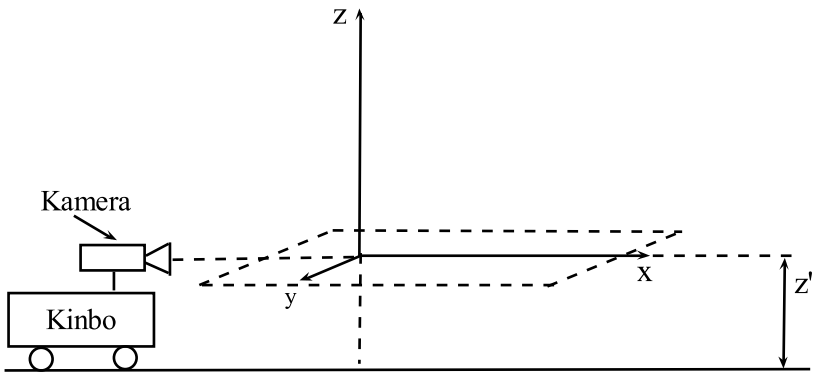
Modul lokalizace a jeho následná integrace formou 2D mapy prostředí byla navrhnuta s využitím knihovny *MRPT* a demo aplikace, která je dostupná společně s instalací knihovny a jmenuje se „Kinect 3D Slam“. Toto demo je založeno na způsobu SLAM lokalizace s využitím vizuální odometrie

a Kalmanových filtrů. Bohužel v základní implementaci je toto demo nepoužitelné. Dochází zde ke ztrátě informace namapované scény už po malém pohybu kamery po místnosti. Demo je primárně sestrojeno pro základní ukázku principu lokalizace a možností *MRPT* knihovny. Využívá detektory hran pro detekci příznaků objektů, tyto příznaky poté ukládá a s každým snímkem kamery se snaží tento příznak opět najít a spočítat okolní 3D mapu prostředí společně s pozicí. Pro více informací, jak fungují algoritmy lokalizace a mapování nahlédněte v sekci 4.4.2 na straně 48. Část s detektorem příznaků byla výrazně poupravena a změněny vlastnosti nastavení integrovaného řešení vizuálního detektoru. Jedna ze změn byla mazání nerozpoznaných příznaků ze seznamu, zvětšení velikosti oblasti rozpoznávaného příznaku a jiná drobná vylepšení. Díky tomu je možné při zachování rozumné[4] rychlosti robota detekovat i středně velké místnosti a lokalizovat pohyb. Na větších prostorech netestováno, ale zde jistě bude problém s hledáním příznaků v okolí vzhledem k omezeným HW vlastnostem kamery (více v sekci 1.4.2 na straně 3) a zde např. budeme muset onu místnost poupravit a dát v dostatečných vzdálenostech nějaké předměty ze kterých se budou generovat potřebné příznaky.

Protože mapa je generována ve třech dimenzích (v ose x, y, z), zatímco my pro potřeby plánování potřebujeme dvou dimenzionální mřížku mapy. Abychom tohoto dosáhli, byl vymyšlen jednoduchý způsob řešení; a to sice *MRPT* knihovna; ta nám umožňuje vrátit body mapy v určité výšce, k tomu slouží funkce *clipOutOfRangeinZ(fromZ, toZ)*. Protože však po každém spuštění aplikace se robot nalézá na relativní pozici $s(x' = 0, y' = 0, z' = 0)$ a my víme, že je kamera umístěna na vozidle nad úrovní vozovky (viz obr. 4.18), stačí potom odříznout body v této úrovni o malém rozpětí). Body mapy jsou tedy odříznuty v úrovni $0 \geq z < 0,2$. Výsledné body potom odpovídají bodům souřadnic překážek, či zdi daného prostředí. Je tedy nutné najít rozsah (minimum a maximum) těchto bodů v ose x, y a poté již stanovit velikost mřížky (*count*) výsledné mapy a přepočítat body do mřížky:

$$\begin{aligned} new\_X &= \frac{(x - x^{min}) \cdot count}{x_{max} - x_{min}} \\ new\_Y &= \frac{(y - y^{min}) \cdot count}{y_{max} - y_{min}} \end{aligned} \tag{4.38}$$

[4]Záleží na prostředí a počtu hran objektů v daném prostředí ve kterém se robot vyskytuje.

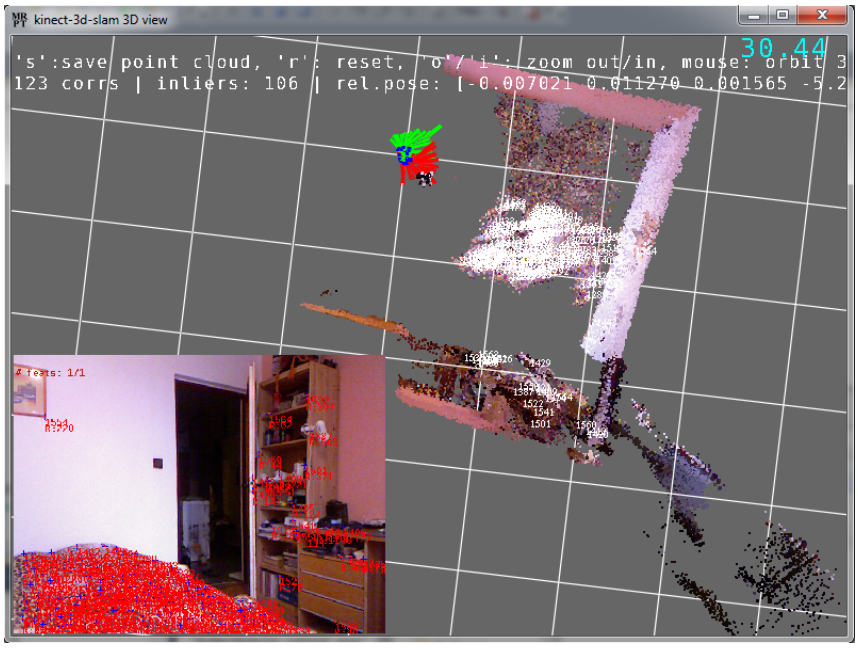


Obrázek 4.18: Způsob vytvoření 2D mapy prořízнutím dané vygenerované 3D mapy prostředí ve výšce z'

.

Máme-li mřížku mapy s pozicemi překážek, snadno již vytvoříme grafickou reprezentaci dané mapy a můžeme na takto vytvořenou mapu využít navrhovaný způsob navigace. Samotnou velikost mřížky volíme podle uvážení, rozměrů místnosti a také jakou požadujeme rozlišovací schopnost pohybu robota s přihlédnutím na hardwarové vlastnosti kamery Kinect (více opět v sekci 1.4.2 na straně 3).
Nejtěžší na tomto řešení byla kompilace knihovny *MRPT*, všech důležitých balíků a nastavení vývojového prostředí s odkazy na zdrojové soubory této knihovny – více se dozvíte v sekci A.3 na straně 84. Toto vše autorovi zabralo přes měsíc zjišťování nabízených možností a nastavování. V práci nabízíme již kompletní a nastavené funkční řešení i s propojením s jazykem C# a tedy jednoduchý způsob možnosti pokračování projektu, navázání na danou práci a případné propojení s platformou *Kinbo*. Na dalším obrázku (viz obr. 4.19) se podívejte na výsledek generování 2D mapy přenesené do aplikace v jazyce C# z knihovny *MRPT* a její 3D originální reprezentaci.

(a) Výsledek prořezání mapy s hranicemi místnosti ve 2D reprezentaci.



(b) Originální 3D reprezentace mapy.

Obrázek 4.19: Ukázka lokalizace a vytvoření 2D reprezentace mapy poslané pomocí *pojmenovaných rour* z 3D mapy vytvořené pomocí knihovny MRPT.

5 Možnosti rozšíření a použití

I přesto, že byly použity základní programovací znalosti rychlosti práce výpočtů algoritmů a např. přístupy do bitmap jsou řešeny přes ukazatele, jistě existují mnohem kvalitnější a rychlejší reprezentace řešení algoritmů $A^\star$ (resp. Dijkstrova), a to nejen při využití rychlejších programovacích jazyků, nicméně i daleko chytřejších řešení výpočtů či implementace – v literatuře [29] je např. možné narazit na vícevláknovou reprezentaci algoritmu $A^\star$. Zejména poté existují daleko lepší a rychlejší implementace algoritmu dynamického programování, který je v uvedeném řešení implementován velmi zjednodušeně, jak udává již zmíněná literatura. Nicméně rychlost plánování nebyla cílem a poměr rychlostí mezi danými řešeními vzhledem k principu, jak fungují, bude vždy velice podobný. Avšak i naše základní implementace algoritmu $A^\star$ ukázala právě jeho sílu a i vzhledem k tomu, že přístup je řešen sekvenčně a to i při naplnění počátečních pomocných matic, je již schopen relativně v „reálném“ čase naplánovat cestu (více v sekci 6 na straně 56), a proto se ve skutečnosti nejčastěji využívá právě algoritmus $A^\star$ v kombinaci s pravděpodobnostními [22] algoritmy hledání cest pro navedení robota v blízkém prostředí, poté i toto drobné plánovací zdržení vůbec nevadí. Při hodně rozsáhlých statických mapách či pro speciální účely (např. pokutování zatočení vpravo) se také používá navržený algoritmus dynamického programování, jenž je schopný po prvotním naplánovaní cest (máme-li již kompletní mapu prostředí) plánovat z každé pozice takřka okamžitě[1]. Například Google ve svých mapách používá právě tento algoritmus, kde má již data předem vypočítána pro různé cílové destinace.

I když jsou výsledné body cesty testovány, zda se nenachází v prostředí překážky, bylo by vhodné zkusit jiné způsoby redukce počtu bodů, v námi implementovaném případě se stává, že za jistých podmínek, pokud je originální výsledná trasa již dosti přímá, dochází k velkému ořezání bodů a u některého nastavení agresivity ořezání dochází dokonce k ořezání takovým způsobem, že výsledná cesta vede přes překážku. Při nastavení ořezávání je tedy třeba dbát zvýšené opatrnosti, vede-li daný segment cesty mezi dvěma body přes překážku, poté již není možné tuto skutečnost efektivním způsobem ověřit. V našem případě řešení je také vidět, proč se např. v algoritmu $A^\star$ pro plánování v robotice nejvíce používá heuristická funkce Manhattan (více v sekci 6.1.2 na straně 57), byť funkce Eukleidových vzdáleností podává

[1] V rámci provedení vyhlazení a vykreslení cesty.

nepatrně rychlejší výsledky řešení (srovnání v sekci 6.1.1 na straně 56), ale dává „moc rovné“ úseky cesty, na které se náš způsob redukce nehodí a chtělo by to ponechat více bodů v daných rovných úsecích pro plynulejší navigaci – v sekci 6 na straně 56.

Nutná rozšíření jsou naopak třeba na použitém softwarovém vybavení platformy Kinbo. Problémy dosavadní platformy jsou zejména v nedostačující snímkové frekvenci (pouze 5 *FPS* přes WiFi) pro použití jediného možného způsobu lokalizace – mapováním oblasti s využitím vizuální *odometrie*, ale také v použité složitosti řešení. Samotná platforma čítá přes 20 000 řádků a to je při řešení účelu, pro jaký byla stvořena, opravdu hodně. Navíc vzhledem k její modulárně stavěné architektuře softwaru je prakticky nemožné program za chodu rozumně ladit[2]. Stává se tedy nevhodnou pro vývoj. Dále se chová nepředvídatelně, příkaz k rozjezdu robota dorazí často se značným zpožděním, které není dáno jen odezvou posíláním dat přes WiFi, neboť odezva motorů se v jistých případech pohybuje i kolem 1 sekundy. Občas platforma také zcela padá bez zjevné příčiny. Ke zjednodušení a dosažení mnohem kvalitnějších výsledků lokalizace a navigace je třeba tak, jak je tomu u 99% speciálních robotických vozidel, sáhnout také po HW úpravách platformy a zahrnout senzory pro čtení *odometrie* vozidla a podstatně si tím rozšířit portfolio řešení lokalizace (více v sekci 4.4.2 na straně 48), ale také výrazně zkvalitnit možnosti výstupu navigace po výsledné trase.

Použití navrženého řešení je zřejmé, že bylo psáno na dostatečné abstraktní úrovni a samotné řešení je vhodné k použití při plánování a navigaci v prostoru po zemi pohybujících se robotických zařízení s koly (pásy), zejména poté bylo cíleno pro budoucí použití s platformou Kinbo.

5.1 Návrh řešení se stávající platformou

Samotná idea platformy byla taková, že v budoucnu bude obsluhovat více strojů zároveň a bude sloužit i jako podpůrný prostředek pro předměty typu *Inteligentní počítačové systémy*, *Interakce člověk a počítač* atp. O výpočetně náročná řešení jako je samotná lokalizace a plánování se bude poté starat server, který již klientům komunikujících s robotem zašle výsledná zpracovaná data. V dnešní době je zcela prakticky nemožné, aby samotný student

[2]Jediná možnost s využitím výpisů, které jsou však také posílány přes wifi, jedná-li se o výpisy modulu běžícího na samotném HW Kinbo, nemožné využití *debugeru*.

vytvořil kompletní robustní řešení lokalizace (zejména tedy řešení založená na vizuální *odometrii*) robota, neboť řešení lokalizace jsou již vyvíjena dlouhá léta i za pomoci soukromého sektoru a také těchto řešení existuje celá řada a jsou volně dostupná. Nabízí se tedy způsob využít buď jednotně volně dostupných řešení, či například vlastností *MRPT* a jím podobných knihoven (*ROS* ...), jež integrují všechny tyto metody do velmi pěkně řešené softwarové knihovny umožňující snadné použití a integrace jednotlivých řešení. Bohužel (resp. bohudík) tyto knihovny, stejně jako veškeré algoritmy, které vyžadují rychlé zpracování dat, jsou implementovány v jazyce *C++*. Nabízí se tedy možnost pro snadnější integraci knihoven využít pro budoucí návrh platformy jazyka *C++*.

Pokud bychom ale zůstali u daného hotového SW řešení platformy *Kinbo* a chtěli integrovat problém řešení lokalizace a námi navržené řešení navigace, zásadní věc, kterou je třeba brát na zřetel, je fakt, že samotné vzdálenosti a přesnost hloubkové mapy jsou nelineární. Knihovny pracují ve skutečnosti typicky v metrech a mají předem definované koeficienty na přepočet vzdáleností. Poté záleží na volbě driveru pro kameru Kinect a způsob reprezentace dat; zde se dostáváme k dalšímu jádru problému, všechny volně dostupné knihovny řešící lokalizaci jsou založeny na *opensource* ovladačích kamery Kinect a jedná konkrétně o ovladače *Freenect* a *OpenNI*. Platforma je založena na ovladačích *Freenect*, zdá se, že máme alespoň z části vyhráno, bohužel to není pravda. Reprezentace dat neodpovídá skutečnosti v jakém formátu bychom měli data dostávat z ovladače. Zejména potom hloubková mapa má být o rozlišení $640 \times 480 \times 3$, kde právě jednotlivé vzdálenosti jsou reprezentovány třemi byty (celkově dokáže kamera rozlišit 2048 hloubkových úrovní). Platforma však tuto hloubkovou informaci degraduje do 255 hloubkových úrovní a posílá v 1 bytu. Dále po zkoumání a záměrném snížení snímkové frekvence v příkladech lokalizace knihovny *MRPT* na úroveň frekvence scény proudící ke klientovi při užití platformy Kinbo bylo zjištěno, že daná snímková frekvence je nedostačující pro použití vizuální *odometrie*.

Jak jistě tušíte problémů skýtá platforma více; pokud chcete i přesto dosáhnout úspěchu, doporučuji jediné implementačně nepříliš náročné řešení, a to spustit problém lokalizace přímo na výpočetní jednotce Kinba. Jak bylo zjištěno, CPU je dostatečně výkonné a zvládá v průměru lokalizaci s 20 *FPS*, samotná úprava platformy bude spočívat v tom, že již nebude přímo komunikovat s ovladači a veškerá data z kamery, včetně výsledné mapy lokalizace, budou posílána navrženou (více v sekci 4.4.2 na straně 48) *meziprocesovou* komunikací mezi knihovnou *MRPT* a platformou Kinbo; zde je však

potenciální problém vzniku zpoždění komunikace mezi procesy, nicméně toto zpoždění je zanedbatelné vzhledem k vlastnostem platformy Kinbo. K uživateli se již bude posílat hotová mapa – zde je možno využít navrhovanou realizaci lokalizace a hlavně poté výsledek této práce – modul navigace. Na straně uživatele (příp. serveru) bude zpracovávána samotná cesta a navigace robota – nutno opět brát v potaz, že při posílání dvou bitmap je rychlost přes WiFi dané platformy v průměru 5 *FPS* a je tedy nutné buď rychlost zvýšit lepší kompresí mapy, nebo použít velmi pomalou rychlost robota k zajištění spolehlivé navigace.

Nicméně silně doporučuji buď platformu přepracovat, nebo zkusit platformu *Kinbo2*, o které jsem se dozvěděl před necelým měsícem, že byla vytvořena. Zde je však problém, že je založena na Microsoft SDK driverech a principiálně všechny výše uvedené problémy stávající platformy zůstávají... Také je žádoucí namontovat senzory pro čtení otáček kol (řešení odometrie) a zkvalitnit tím nejen lokalizaci, nýbrž následnou navigaci vozidla.

6 Dosažené výsledky

V této kapitole čtenáře seznámíme s dosaženými výsledky a komplexním otestováním dané práce.
Byla navržena podoba navigačního systému a kladen důraz na abstrakci řešení s přihlížením faktů a postřehů při testování platformy platformě Kinbo ze Západočeské univerzity. Podoba nalezené cesty byla testována na umělých, záměrně složitých mapách s ostrými hranami překážek. Zde, jak se za chvíli přesvědčíte, podává za předpokladu správných nastavení parametrů pro daný typ mapy poměrně přesvědčivé (málo ostrých hran, vzdálenosti od překážek a plynulost cesty) výsledky nalezené cesty. K ověření předpokladů výsledků bylo dále také provedeno testování na reálných mapách prostředí, které generuje knihovna *MRPT*. Na reálných mapách podává řešení ještě přesvědčivější výsledky a podoba cesty vypadá pro účel navigace velmi slibně.

Dále byly také otestovány vlastnosti PID regulátoru a porovnány chyby při různým nastavení kinematického modelu s provedenými úpravami regulátoru 4.32 vzhledem k jeho originální podobě 2.8.

6.1 Plánování trasy

Vlastnosti plánování tras byly testovány na pěti mapách o záměrně složitém postavení překážek na mapě s ostrými hranami a různým rozmístěním po mapě, poté dvou reálných mapách, kde obě představují chodbu univerzit – z USA a Španělska ve města Malaga. Obrázky všech testovaných map naleznete v příloze.

6.1.1 Srovnání rychlostí algoritmů

Ukažme si nejdříve dosažené rychlosti generování cesty na první testované mapě – *map1*. Cesta byla generována z levého horního rohu do pravého dolního. Měření bylo provedeno 3× po sobě se zprůměrováním dosažených výsledků. Testováno bylo na stroji s procesorem Core 2 Duo 3.0GHz s 8GB DDR2 800MHz RAM. Zde je třeba zdůraznit, že u dynamického programování trochu také záleží na konkrétním postavení cíle. Pokud je zvolena cílová des-

tinace uprostřed mapy, tak algoritmus dosahuje zhruba o polovinu lepších časů, neboť se rekurzivně vydává všemi směry od dané cílové pozice. Řekněme, že ale vliv cílové destinace má na výsledek minimální vliv (ostatně zbylé algoritmy také neupřednostňujeme a záměrně volíme destinace co nejdál
od sebe), poté nejvyšší vliv má na výsledek právě velikost (rozlišení) mapy, neboť algoritmus rekurzivně několikrát prochází celou mapu. Pro pořádek tedy uvedu, že rozlišení mapy je 400×200 pixelů.

Rychlost plánování [s]	
$A^\star$ – heur. formule Manhattan	0,093
$A^\star$ – heur. formule Diag. zkratka	0,113
$A^\star$ – heur. formule Max. dx, dy	0,100
$A^\star$ – heur. formule Eukl. bez odmocniny	0,030
$A^\star$ – heur. formule Eukl. vzdáleností	0,126
Dijkstrův algoritmus	0,140
Dynamické programování	8,723
Dyn. programování – nedet. pohyb	56,130

Tabulka 6.1: Srovnání rychlostí použitých algoritmů pro plánování cest.

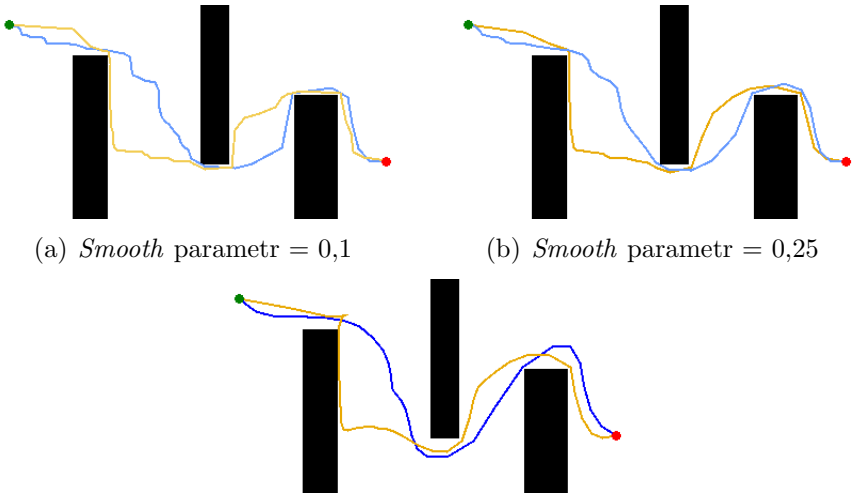
Z tabulky vyplývají teoretické předpoklady i dominance algoritmu $A^\star$. Zajímavá je rychlost algoritmu společně s použitím heuristické formule *Eukleidových vzdáleností* bez odmocniny. Patrná je také velmi dlouhá konvergence algoritmů *dynamického programování*, zvláště potom pro nedeterministický pohyb.

6.1.2 Porovnání vlastností algoritmů

Zde si ukážeme vliv různého nastavení na kvalitu výsledného obrazu a rozebereme si proč daná cesta vzhledem k nastavení tak vypadá, a také rozebereme rozdíl podoby cest u jednotlivých algoritmů.

Zpočátku nechme nastavení d pro *Douglas-Peuckerův* algoritmus na 0,1 a srovnejme zejména vlastnosti Dijkstrovo a algoritmu $A^\star$ vzhledem ke třem hodnotám nastavení váhy vyhlazení – $0,1; 0,25; 0,4$. Nevyhlazená originální cesta nás totiž moc nezajímá, po ní bychom naváděli robota velmi těžko.

Jak je patrné z uvedeného obrázku (viz obr. 6.1) parametr vyhlazení má



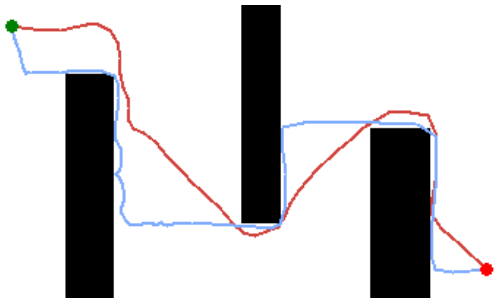
(a) *Smooth* parametr = 0,1 (b) *Smooth* parametr = 0,25

(c) *Smooth* parametr = 0,1

Obrázek 6.1: Porovnání vlivu nastavení parametru vyhlazení na výsledek cesty. Modře vyznačena cesta algoritmu $A^\star$ a oranžově potom Dijkstrův algoritmus.

podstatný vliv na výslednou podobu cesty. Na obrázku je také vidět, že nejen díky rychlosti – v sekci 6.1.1 na straně 56, ale také díky výsledné podobě cesty se hodí algoritmus $A^\star$ více. Vlivem diagonální zkratky, kdy algoritmus bere v potaz umístění cíle a mezi jednotlivými překážkami vede originální cesta napříč, zatímco u Dijsktrova algoritmu je brána v potaz jen vzdálenost od počátku, čili podoba cesty vede podél prví překážky a pod úrovní středové překážky jde teprv doprava směrem k cíli. Vzhledem k principům vyhlazení cesty (více v sekci 4.2.3 na straně 33) má potom jakýkoliv přímočarý pohyb negativní výsledek podoby cesty, která má poté logicky také přímočarou podobu s relativně stále ostrými hranami.

Porovnejme si také mezi sebou vlastnosti algoritmů dynamického programování. Nastavení váhy vyhlazení je $0,25$ a parametr **d** Douglas-Peucker algoritmu je nastaven stále na hodnotu 0,1. Na uvedeném obrázku (viz obr. 6.2) je poté jasně vidět teoretická výhoda upraveného algoritmu s nedeterministickým pohybem. Důvody jsou podobné s předchozím případem, neboť algoritmus dynamického plánování s nedeterministickým pohybem vytváří cestu s určitou vzdáleností od překážek, navíc není cesta tolik přímočará jako při klasickém řešení algoritmu dynamického plánování.



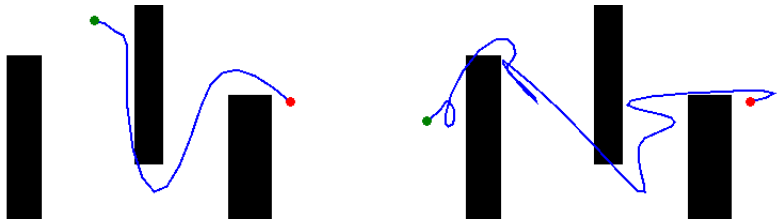
Obrázek 6.2: Výsledek cesty pro algoritmus Dynamického programování. Červeně vyznačen upravený algoritmus dynamického programování pro stochastický pohyb s pravděpodobností přechodu 50%. Modře je poté vyznačena klasická podoba algoritmu – bez nedeterminismu.

6.1.3 Nevhodná nastavení parametrů

Nyní se podíváme na některé případy se špatně zvolenými parametry a proč k dané chybě dochází.

Jak již bylo řečeno, volby parametrů vyhlazení cesty a převážně potom parametru redukce bodů na cestě je třeba volit s rozumem. Zejména pak druhý zmíněný parametr může vytvořit nežádoucí cesty. Na následujícím obrázku si povšimněme, že cesta uprostřed vede přes roh překážky. Ačkoliv při vyhlazování cesty ošetřujeme body, aby nedošlo k „převyhlazení“ cesty a tím fixujeme body, jejichž pozice se změnila a odpovídá pozici překážce, k jejím původním hodnotám. Nicméně vede-li již segment cesty přes překážku, jak je tomu na daném obrázku (viz obr. 6.3(a)), oba krajní body jsou mimo překážku a ověření cesty selže, resp. projde bez chyb a navracení pozic. Dále se společně také podívejme na špatně nastavenou váhu parametru vyhlazení cesty, kde následně došlo k přehlazení cesty a zároveň k chybné redukci bodů na cestě (viz obr. 6.3(b)).

Jak sami vidíte jednotlivá nastavení mohou mít drastický vliv na výslednou kvalitu cesty a je třeba při volbě dbát velké obezřetnosti. V budoucnu je právě počítáno s nastavením parametrů na hodnoty podávající při konkrétním řešení algoritmu nejrozumnější hodnoty. Každý algoritmus má totiž jiné vlastnosti a vyžaduje odlišné nastavení parametrů.



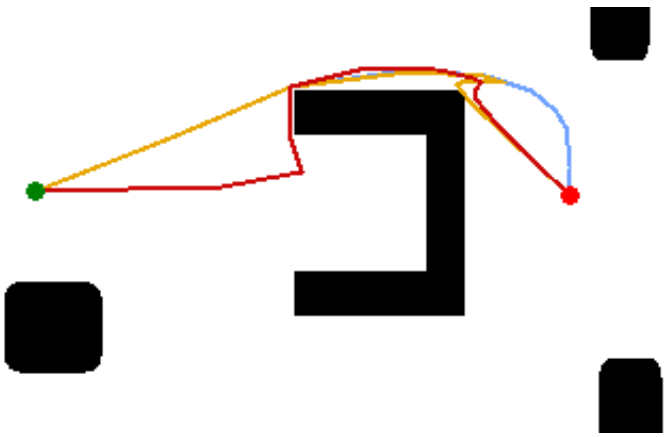
(a) Douglas-Peucker parameter = 0,3

(b) *Smooth* parametr = 0,45

Obrázek 6.3: Ukázka 6.3(a) s nastavením parametru d pro Douglas-Peucker redukci bodů na 0,3. Vysoká tolerance algoritmu simplifikace bodů vede k výsledné cestě přes překážku. Dále 6.3(b) také ukazuje „přehlazení“ cesty
.

6.1.4 Doporučení a reálné mapy

Díky výše uváděným skutečnostem vychází jako vítězný algoritmus pro plánování cest vzhledem k podobě cesty algoritmus $A^\star$ společně s dynamickým plánováním při nedeterministickém pohybu. Díky časovým náročnostem poté jako nejlepší vychází algoritmus $A^\star$. Srovnejme si tedy ještě výsledky různých formulí algoritmu $A^\star$, protože všechny dosud uvedené obrázky byly s použitím formule *Manhattan*.

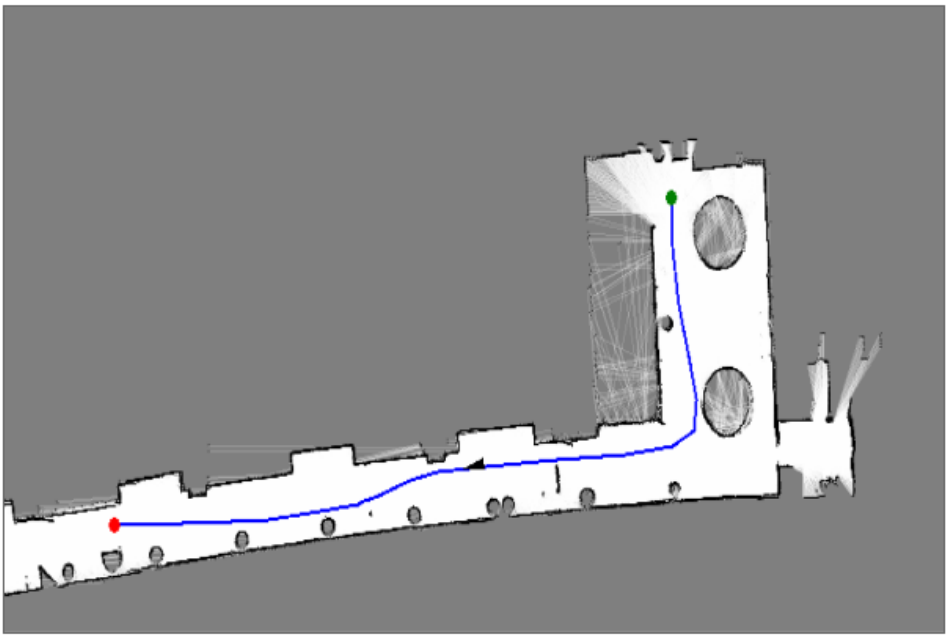


Obrázek 6.4: Srovnání heuristických formulí. Modrá odpovídá formuli *Manhattan*, oranžová formuli *Maximum dx, dy* a červená potom formuli *Euklidových vzdáleností*.

Jak je vidět na obrázku výše (viz obr. 6.4), nejlepší trasu podává heuristika *Manhattan*, která je také nejvíce používána. Nutno podotknout, že na

konkrétním případě heuristika *Diagonální zkratka* podala zcela totožnou křivku a i obecně podává podobné výsledky cest, nicméně vzhledem ke skutečnosti, že formule *Manhattan* nalezne cestu obecně rychleji, tak vychází lépe. Zcela potom propadla formule *Eukleidových vzdáleností*, i když ve všech případech najde cestu nejrychleji ze všech řešení, výsledná podoba křivky je bez úpravy opět „moc přímá“ a nedochází proto k žádoucímu vyhlazení cesty. Pro použití dané formule by bylo vhodné řešení simplifikace bodů zcela přepracovat.

Algoritmy byly následně testovány také na reálných mapách, které generuje knihovna *MRPT*. Byly proto staženy nasnímané scény chodeb z různých univerzit, načež byla vytvořena pomocí knihovny 2D mapa prostředí. Na těchto mapách byla poté testována námi navrhovaná řešení. Podle předpokladů podávají mnohem lepší výsledky než na uměle vytvořených mapách. Povšimněte si zejména velice plynulé cesty s dostatečnou vzdáleností od překážek a plynulého objetí překážek (viz obr. 6.5).



Obrázek 6.5: Výsledek cesty u reálné mapy s využitím algoritmu $A^\star$. Váha pro vyhlazení cesty nastavena na 0,4 a Douglas-Peucker tolerance na 0,15.

6.2 PID navigace

Byly testovány vlastnosti PID regulátoru společně s námi navrženou úpravou. Jelikož při klasickém matematickém modelu bez šumu jsou rozdíly velice nepatrné, či v rámci chyby měření, zaměříme se na velmi zašumělý případ robota, kde nastavíme chybnou geometrii robota, který se bude stáčet pořád doprava o 10 stupňů. Dále nastavíme počáteční odchylku od směru jízdy na 45 stupňů, délku robota na 20 a šířku na 10 (v bodech mřížky mapy) s maximální rychlostí 1, a také šum při zatáčení s rozptylem 10 stupňů. Takto upravený model otestujeme na mapě *PID.png* s oválnou cestou. Cesta byla prováděna v rámci téměř celého oválu (krom počátečního nutného rozestupu postavení). Celkově byly provedeny tři měření. Před každým měřením bylo nalezeno optimální nastavení regulátoru pomocí dodané funkce.

Rozdíly chyb PID regulátorů		
Použitý regulátor:	Naše PID	Klasické PID
	Rozptyl od trati	
Měření č. 1:	2,42	5,91
Měření č. 2:	10,45	16,95
Měření č. 3:	3,12	21,08

Tabulka 6.2: Rozdíly střední kvadratické odchylky od vypočítané cesty mezi navrhovanými implementacemi PID regulátoru.

Jak lze vidět na výše uvedené tabulce 6.2, střední kvadratická odchylka (rozptyl) od trati se u klasického řešení PID regulátoru pohybuje podstatně výše. I opticky za celou cestu nedošlo ke konvergenci a robot stále „přestřeloval“ danou trasu, zatímco u našeho řešení byla jasně patrná konvergence, kdy se po polovině trati robot ustálil a již ze své trasy podstatně nevychyloval, čemuž také odpovídá průměrná odchylka od trati ze všech měření 5,33 bodů. Pokud by čtenáře zajímaly konkrétní konfigurace PID regulátoru pro jednotlivá měření, uvádím je v následující tabulce 6.3.

6.3 Simulovaná navigace na reálné mapě

Na závěr byl zkoušen model robota především na reálných mapách generovaných knihovnou *MRPT*. Vzhledem k velikosti chodby a rozměrů generované mapy byl zvolen robot o délce 10 bodů a šířce 5 bodů, rychlost

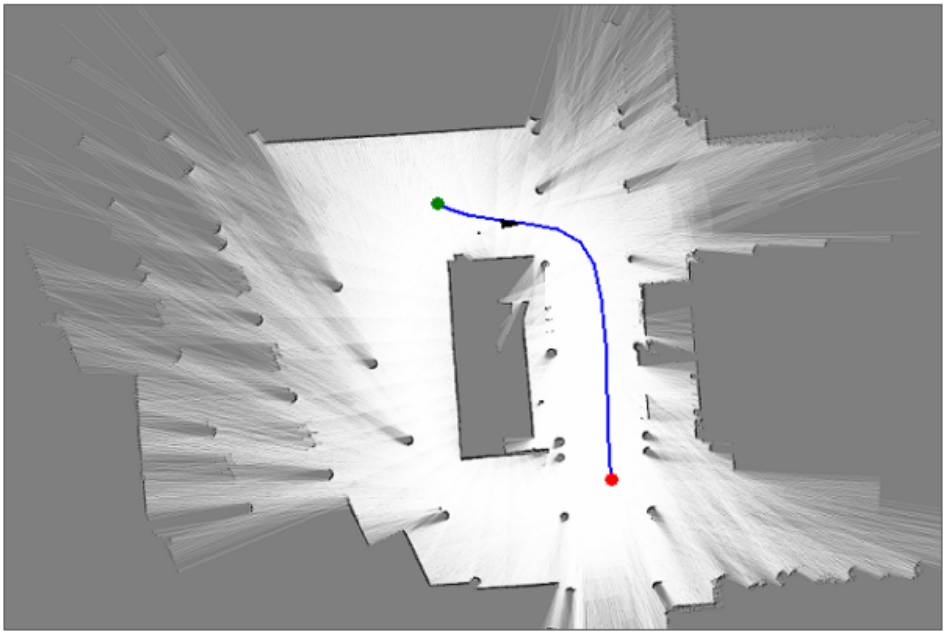
Nastavení vah PID regulátoru						
Použitý regulátor:	Naše PID			Klasické PID		
	Nastavení vah regulátoru α, β, γ					
Měření č. 1:	-0,18	1,14	0	2,20	1,42	0
Měření č. 2:	-0,53	3,32	0	1,20	0,89	0
Měření č. 3:	0	0,804	0	1,47	0,94	0

Tabulka 6.3: Konfigurace PID regulátorů pro měření z tabulky 6.2

robota poté nastavena na hodnotu 0,5. Pro algoritmus hledání cesty byl zvolen jeden z nejvhodnějších kandidátů (více v sekci 6.1.4 na straně 60), a to sice algoritmus A^{star} s nastavením *Douglas-Peucker* tolerance na 0,15 a váhy vyhlazení 0,4 s *Manhattan* heuristikou. Výsledky nalezených cest je třeba předem zkontrolovat a zejména potom se ujistit, zda cesta nevede přes překážku a také s dostatečnou vzdáleností od překážek, aby se robot v rámci jeho šířky na danou trasu vešel. Při výše zmíněném nastavení je průběh nalezené cesty velice rozumný (viz obr. 6.5) a výsledná navigace robota s nalezenými parametry vah PID regulátoru $\alpha = -0,72, \beta = 6,03, \gamma = 0$ vedla k plynulé jízdě robota bez kolize a s nulovou střední kvadratickou odchylkou od cesty. Vzhledem ke skvělým možnostem PID regulátoru jsme se chtěli přesvědčit, do jaké míry šumu nám umožní regulátor pohyb dostatečně zregulovat.

Dále jsme testovali druhou mapu reálného prostředí se stejným výsledkem. Přejděme tedy k druhé fázi testování. Uvažujme postřehy z reálné platformy Kinbo; zde nedochází k téměř žádné odchylce při jízdě rovně, nastavme tedy hodnotu geometrie na tři stupně stáčení vpravo, předpokládejme že ale vzhledem k absenci *odometrie* bude složité regulovat pouze pomocí *PWM* modulace samotné zatáčení. Namodelujme tuto situaci 20 stupňovou odchylkou zatáčení. Zde musíme podotknout, že implementovaná kinematika modelu robota nastaví požadovanou úroveň natočení v rámci času t okamžitě, ve skutečnosti však bude docházet k postupnému natáčení platformy v čase a výsledný pohyb bude mnohem plynulejší. Avšak nám šlo o to zjistit, jak si poradí regulátor s takovýmto zašumělým nastavením modelu robota. Ostatní parametry ponechme jako v předchozím případě.
I zde poté docházelo na jednoduché křivce (viz obr. 6.6) při nalezení nejvhodnějších parametrů regulátoru $\alpha = 0,09, \beta = 0,31, \gamma = 0$ k výborným výsledkům regulace a trasy s takřka nulovou odchylkou a bez kolize s okolím. Dále byl k předchozímu nehostinnému nastavení přidán šum k rychlosti 0,1, kde se vzhledem k přednastavenému parametru rychlosti na 0,5 jedná o 25% šum (odchylku) rychlosti. Nezapomeňme však, že i samotná ujetá vzdálenost

(resp. rychlost) má vliv na poloměr zatáčení a nastavíme-li takto velkou odchylku v čase t, dochází ke značnému odchýlení od naplánované změny směru jízdy při zatočení a výslednému ujetému poloměru zatočení. Na reálné platformě však nebude docházet k šumu v rychlosti a nastavíme-li rychlost na nějakou definovanou hodnotu, u které spočteme ujetou vzdálenost za námi stanovený čas Δt, tak s tímto nebude žádný problém. Nicméně i s tímto si byl schopen regulátor poměrně obstojně poradit a následoval trasu i po zatočení s velmi malou odchylkou a rychle se vracel ke středu osy cesty.



Obrázek 6.6: Druhá testovaná mapa reálného prostředí.

Z výše uvedených skutečností tedy plyne, že regulátor PID s námi navrhovaným řešením je schopen za předpokladu správně nastavených parametrů regulátoru jistě bude schopen regulovat platformu Kinbo na Západočeské univerzitě.

7 Závěr

V dané práci jsem se seznámil s problematikou navigace a plánování cest pro použití speciálních vozidel v robotice. Všechny cíle zadání byly splněny. Mimo zadání byla také navrhnuta podoba lokalizace s využitím knihovny *MRPT*[1] a následné propojení této knihovny s platformou Kinbo (viz sekce 4.4).

Byl implementován a otestován způsob plánování tras a navigace robotických vozidel. Vlastnosti vytvořeného subsystému byly ověřeny simulačním softwarovým vybavením. Simulovaný pohyb robota je podložen matematickým modelem, který umožňuje nasimulovat různé způsoby projevu stochasticity při pohybu robota s využitím Gaussova rozdělení.

Jsou implementovány čtyři druhy algoritmů plánování. Vlastnosti těchto algoritmů byly komplexně otestovány v simulačním softwarovém vybavení. Při daném testování se jasně projevily jednotlivé výhody/nevýhody použitých implementovaných řešení (více v sekci 6 na straně 56). Za předpokladu správně nastavených parametrů na vyhlazení cesty a redukce počtu bodů na cestě nabízí implementovaná řešení velmi rozumné podoby výsledných cest 6.1.2. Navigace po generovaných trasách byla ověřena za pomoci PID regulace. Dále byly navrženy drobné úpravy PID regulátoru, které dávají s daným matematickým modelem lepší výsledky – více v sekci 6.2 na straně 62. Mimo zadání práce byl navržen způsob lokalizace a meziprocesové komunikace s platformou Kinbo a dále nutné změny platformy pro plnou integraci navrhovaného řešení navigace.

Tato práce může sloužit jako navigační softwarové řešení pro různé druhy robotických vozidel. Zejména je cílena k budoucí integraci do platformy Kinbo (viz sekce 5.1).

[1] Mobile robot programming toolkit

Seznam obrázků

Seznam tabulek

Literatura

[1] *Edge Detection Boundary Tracing.* EE 528 Digital Image Processing, 2005.

[2] Federico Greco *Travelling Salesman Problem.* ISBN 978-953-7619-10-7, 2008.

[3] Dave Ferguson, Maxim Likhachev, Anthony Stenz *A Guide to Heuristic-based Path Planning.* School of Computer Science, Carnegie Mellon University Pittsburgh, 2005.

[4] Hart, P. E., Nilsson N. J. , Raphael B. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* 100–107. doi:10.1109/TSSC.1968.300136, 1968.

[5] Sebastian Thrun, Wolfram Burgard *Probabilistic robotics.* , 1999-2000.

[6] *Graph Theory: Shortest Path.* CPSC 490.

[7] Luca De Filippis and Giorgio Guglieri *Advanced Graph Search Algorithms for Path Planning of Flight Vehicles*, str. 163-173, Politecnico di Torino Italy.

[8] D. H. Douglas and T. K. Peucker. *Algorithms for the reduction of the number of points required to represent a line or its caricature.* The Canadian Cartographer, 10(2):112-122, 1973.

[9] T. K. Peucker. *A theory of the Cartographic line.* Simon Fraser University, N00014-73-C-0109, 1973.

[10] Andrew Richardson, Edwin Olson. *Iterative Path Optimization for Practical Robot Planning.* Department of Computer Science and Engineering, University of Michigan, `http://aprol.eecs.umich.edu`

[11] Jaroslav Reichl, Martin Všetička. *Encyklopedie Fyziky.* `http://fyzika.jreichl.com`

[12] Jose Luis Blanco Claraco, *Development of Scientific Applications with the Mobile Robot Programming Toolkit.* University of Malaga, 2010.

[13] W. Durfee, *Arduino Microcontroller Guide.* University of Minnesota, 2011.

[14] Wikipedia, *Flood fill*. `http://en.wikipedia.org/wiki/Flood_fill`, 2007 (cit. 10.12.2012).

[15] Robert McDowall, *Fundamentals of HVAC control systems*. ISBN 978-0-08-055233-0, ©2008.

[16] Christopher Batten, *Control for Mobile Robots*. Maslab IAP Robotics Course, ©2005.

[17] Matt Krass, *PID Control Theory*. ©2006.

[18] Dijkstra, E. W. *A note on two problems in connexion with graphs*. Mumerische Mathematik 1:269-271.doi:10.1007/BF01386390 ©1959.

[19] Wikipedia, *Arcus tangens 2 in Computer Languages*. `http://en.wikipedia.org/wiki/Atan2`, 2013 (cit. 26.08.2012).

[20] Wikipedia, *Gradient descent*. `http://en.wikipedia.org/wiki/Gradient_descent`, 2013 (cit. 26.08.2012).

[21] MSDN Library, *Named Pipes, Interprocess Communications*, 2001.

[22] Steven M. Lavalle, James J. Kuffner, Jr. *Randomized Kinodynamic Planning*. The International Journal of Robotics Research, Vol. 20, No.5, pp. 378-400, May ©2001, Sage Publications.

[23] Elmar A. Rückert, *Simultaneous Localisation And Mapping For Mobile Robots With Recent Sensor Technologies*, Master Thesis, Graz, Austria 2009.

[24] Mgr. Miroslav Kulich, *Lokalizace a tvorba modelu prostředí v inteligentní robotice*, disertační práce, 2009.

[25] Soren Riisgaard and Morten Rufus Blas, *SLAM for Dummies*, disertační práce, 2009.

[26] Akash Patki, *Particle Filter Based SLAM to Map Random Environments Using IROBOT ROOMBA*, Thesis, Vanderbilt University, 2011.

[27] Kalman R. E., *A new approach to linear filtering and prediction problems*, Transactions of the ASME Journal of Basic Engineering, (82(Series D)):35-45, 1960.

[28] Persi Diaconis, *The Markov Chain Monte Carlo Revolution*, Departments of Mathematics and Statistics, Stanford University, 2011.

[29] Rafia Inam, *A star Algorithm for Multicore Graphics Processors*, Master's Thesis, Chalmers University of Technology, Göteborg, 2009.

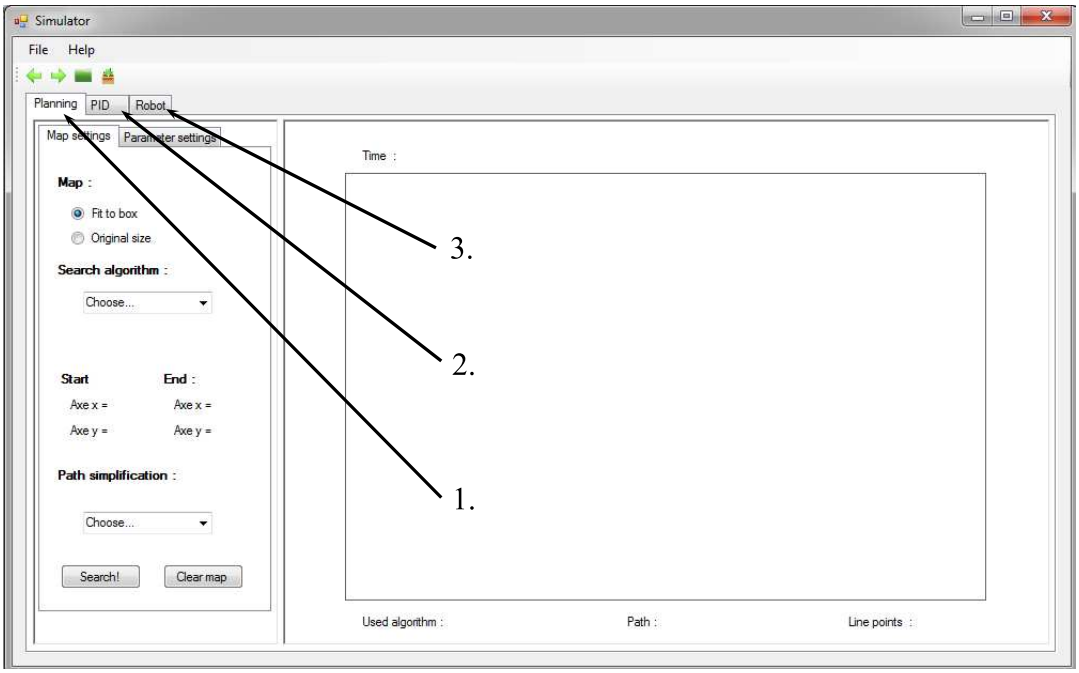
Přílohy

Příloha A

A.1 Uživatelská příručka – Simulátor

Upozorňuji, že pro běh aplikace je nutný počítač s Windows XP a výše. Dále mít nainstalováno v osobním počítači *.NET Framework* ve verzi alespoň 4.0. Popišme si nadále ovládání hlavního programového vybavení této práce – simulovanou navigaci robota.

Spust'te z CD ze složky (více v Příloze C.1 na straně 89) „*RobotNavigation*“ a následné podložky „*bin/release*“ spustitelný soubor *RobotNavigation.exe*. Po správném spuštění uvidíte základní obrazovku aplikace (viz obr. A.1).



Obrázek A.1: Obrázek ukazující úspěšně spuštěnou aplikaci.

Popišme si tedy jednotlivé části aplikace (viz obr. A.1):

1. Prvním bodem bude základní okno plánování – *Planning*, zde by si měl

uživatel vyzkoušet různé metody plánování cest na různých mapách a nastavit nejlépe vyhovující nastavení pro konkrétní mapu a algoritmus plánování.

2. v okně *PID* si můžeme vyzkoušet na předdefinovaných testovacích mapách vlastnosti regulátoru a automatického nastavení PID regulace.

3. v posledním třetím okně poté nalezneme již samotnou simulaci robota s nastavením, které jsme vybrali v předchozích oknech.

A.1.1 Okno plánování

První věcí, kterou musíme udělat je vybrat požadovaný snímek mapy. Rozlišení snímku doporučuji v rozlišení 300×200 až 600×400. Nicméně pokaždé dojde k umístění snímku do definovaného rozlišení panelu s mapou – 600×400. Obrázek s mapou prostředí je možné vybrat jeden z testovaných obrázků ve složce „*images*“ (více v Příloze C.1 na straně 89), nebo si obrázek sám vytvořit. Aplikace umožňuje načtení obrázků v následujících formátech – *jpg, png, bmp, tiff a tif*. Podmínka na vzhled mapy je jediná, aby zdi, či překážky byly vyznačeny tmavou barvou a volné prostory naopak co nejsvětlejší barvou.

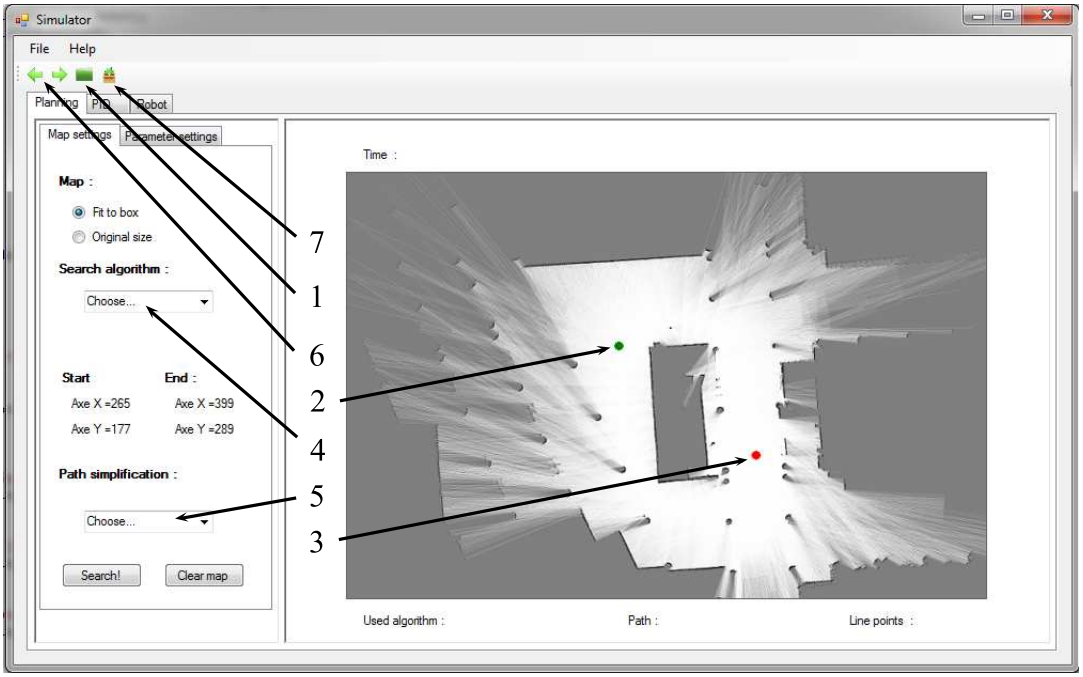
Pro naplánování cesty je potřeba vykonat několik kroků (viz obr. A.2):

1. Nejprve je třeba vybrat mapu prostředí na které následně naplánujeme cestu.

2. Druhým neméně důležitým krokem je vybrat počáteční umístění startovní pozice bodu na mapě. Toto učiníme poklepáním levým tlačítkem myši na požadované místo počátku.

3. Koncový bod vybereme obdobně jako v bodě 2) jen za použití pravého tlačítka myši. Máte-li dokončený výběr cesty a vidíte na mapě dva kruhovité body – zelený jako počátek a červený koncový bod, můžeme vybrat typ algoritmu plánování.

4. Toto učiníme rozklepnutím výběru pod nadpisem *„Search algorithm“*. Zde máme na výběr mezi použitím Dijkstrova algoritmu – *Standard search*, algoritmu $A^\star$ – *Astar* či výběr *dynamického*



Obrázek A.2: Výběr mapy a plánování cesty.

programování – Dynamic prg. (discrete motion) (resp. *Dynamic programming (stochastic motion).*

5. Dále potom vybereme způsob úpravy cesty, zde je na výběr :

   - *Point reduce + smooth* – doporučený výběr pro generování cesty robota s redukcí počtu bodů na cestě a následném vyhlazení cesty.
   - *Reduce point only* – pouze redukce počtu bodů na trase
   - *No simplification* – surová cesta bez úprav

   Jsme-li v pátém bodě výběru, můžeme již tlačítkem *Search!* vygenerovat výslednou cestu.

6. Máme-li již vygenerovánu cestu a zkoušíme různé nastavení a vliv na kvalitu cesty, můžeme se pomocí šipek kdykoliv vrátit a podívat se na předchozí vygenerované cesty.

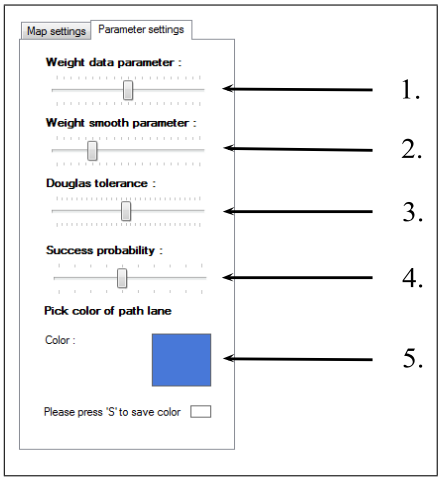
7. Můžeme obrázek s vygenerovanou cestou uložit do souboru.

8. Nezapomeňme také vyzkoušet příjemnou vlastnost dynamického programování, ponecháme-li nezměněn výběr algoritmu plánování na je-

den z nabízených řešení dynamického programování, tak po prvním vypočítání cesty změníme li pouze počáteční pozici, naplánování z jakékoliv jiné startovní pozice je již okamžité.

Zaměříme se nadále na jednotlivé parametry a možnosti nastavení generování v okně *Planning*. Máme na výběr „podokno“ *Map setting*, kde jsme si již většinu nastavení popsali výše, dále se zde akorát nachází možnost zobrazení mapy v originální velikosti zaškrtnutím výběru *Original size* pod nadpisem *Map*. Důležité pro nás bude také podrobné nastavení parametrů v následujícím okně *Parameter settings* (viz obr. A.3):

Parametry nastavení plánování:



Obrázek A.3: Výběr nastavení parametrů plánování cesty.

1. První parametr nastavení *Weight data parameter* udává s jakou váhou jsou vážena jednotlivé pozice bodů cesty zpět na původní místo – parametr tedy zabraňuje „přehlazení cesty“ – více v sekci 2.2 na straně 11. Hodnota tohoto parametru musí být vždy větší než následující druhého parametru.

2. Parametr *Weight smooth parameter* udává s jakou rychlostí (vahou) budou body na cestě odtahována od originálních pozic a do jaké míry bude zmenšována vzdálenost mezi krajními body, tzv. do jaké míry bude docházet k uhlazení cesty mezi třemi po sobě jdoucími body – více se dozvíte v sekci 2.2 na straně 11.

3. Třetí parametr *Douglas tolerance* udává toleranci d s jakou dochází k prořezání bodů na cestě – více

se dozvíte opět v sekci 2.2 na straně 11.

4. Nastavení pravděpodobnosti úspěchu přechodu mezi dvěma body.

5. Barva výsledné cesty, výběr učiníte najetím myši na libovolnou barvu, která se nachází na Vaší pracovní ploše a stisknutím klávesy „S“ tuto volbu potvrdíte.

Dále také nesmíme zapomenout, že veškerá nastavení jsou společné pro všechny záložky v nabízeném řešení. Protože šířka robota má vliv na plánování cesty u Dijkstrova algoritmu a algoritmu $A^\star$, nezapomeňme také vyzkoušet změnu šířky robota (v okně *PID* a vliv na výsledný plán cesty.

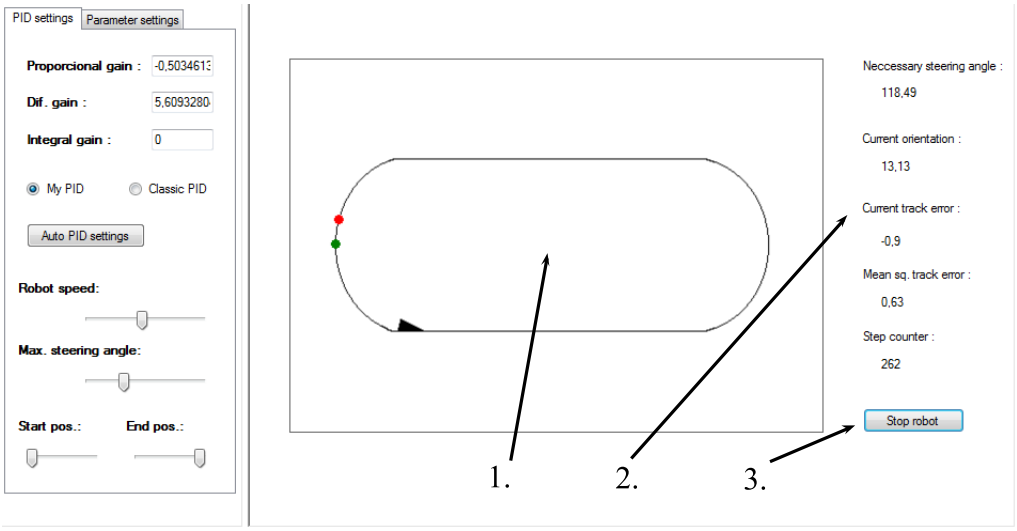
A.1.2 PID

Přejděme do další sekce simulačního programového vybavení. Okno PID (viz obr. A.4) umožňuje výběr požadované testovací uzavřené křivky cesty a následné navigace robota po testovací křivce. Mapa může být opět vlastnoručně vytvořena s jediným požadavkem na uzavřenost dané křivky a doporučeném rozlišením 500×350 bodů, nebo můžete vybrat jednu z testovaných map – *PID, PID2, PID3*. Projděme si nejdříve základní body (viz obr. A.4):

1. Uprostřed mezi panely se nalézá načtená mapa. Načtení mapy provedete stejným způsobem jako u panelu plánování.

2. Vpravo potom po spuštění robota uvidíte jeho aktuální parametry trasy – průměrný rozptyl od trasy, aktuální odchylku od trasy, natočení robota apod.

3. Jediné tlačítko nacházející se v testovacím panelu PID regulátoru spustí nastaveného robota na trase a průběhu času provádí jeho navigaci po dané křivce.



Obrázek A.4: Okno PID regulace.

Dále si povězme o možnostech nastavení v záložkách *PID setting* a *Parameter settings* (viz obr. A.5). :

1. U bodu č. 1 na uvedeném obrázku (viz obr. A.5) můžete u panelu *PID settings* nastavit jednotlivé váhy α, β, γ regulátoru. Na panelu *Parameter settings* poté nastavíte šum povelu zatočení robota.

2. Panel *PID settings* v bodě 2 umožňuje změnu nastavení mezi dvěma PID regulátory (klasický PID a náš upravený PID). Panel *Parameter settings* poté skrývá nastavení šumu vzdálenosti, resp. nastavení šumu pro rychlost.

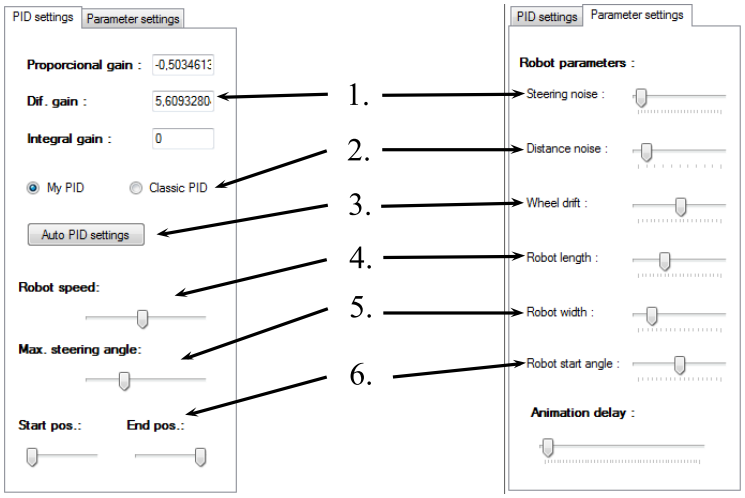
3. Tlačítko *Auto PID settings* automaticky nastaví nejvhodnější parametry PID regulátoru. Parametr *Wheel drift* poté nastavuje stáčení kol robota (chyba geometrie kol).

4. v bodě 4 poté naleznete nastavení rychlosti robota v rozmezí $< 0,05; 1 >$ a také délku robota.

5. Další bod umožňuje nastavit maximální úhel natočení kol a šířku robota.

6. Poslední bod poté umožňuje nastavit výchozí/konečnou pozici robota a počáteční odchylku robota od trati.

Více ke všem výše uvedeným parametrům se dočtete v sekci 4.3.3 na straně 44.



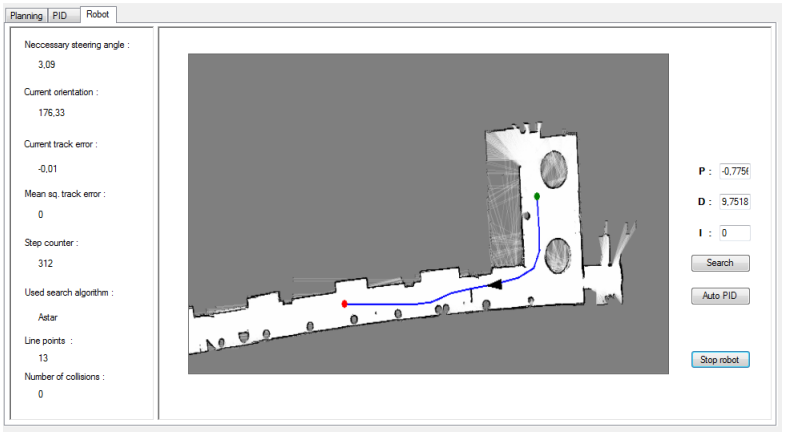
Obrázek A.5: Parametry nastavení PID regulace.

A.1.3 Panel Robot

Samotný poslední panel *Robot* umožňuje načtení mapy robota, zadání výchozí a konečné pozice cesty. Veškerá nastavení, parametry regulátoru a plánování jsou brány společné z příslušných panelů o kterých jsme si již pověděli. Silně je proto doporučeno nejdříve navštívit předchozí záložky, zde nastavit vhodné parametry a poté provádět navigaci v záložce *Robot*. Neboť nezadáte-li např. způsob algoritmu plánování v záložce *Planning*, nedojde k nalezení cesty robota... Uživatel zde musí opět stejným způsobem jako u záložky *Planning* zadat počáteční a koncový bod cesty, dále má možnost nastavit vhodné parametry regulátoru pomocí tlačítka *Auto PID* (silně doporučeno) a na závěr spustit navigaci robota na cestě (viz obr. A.6).

A.2 Uživatelská příručka – Klient

Pro běh aplikace je nutný počítač s Windows XP a výše. Dále mít nainstalováno v osobním počítači *.NET Framework* o verzi alespoň 2.0. Také



Obrázek A.6: Záložka robota s kompletním plánováním a regulace na trase mezi danými body.

je nutné vlastnit k tomu určenou kompletní platformu Kinbo včetně robota s kamerou.

Ke spuštění klientské aplikace určené na platformu Kinbo je nutné ze složky „*Software platformy Kinbo*“ (více v (více v Příloze C.1 na straně 89)) a z podsložky „*Release*“ postupně otevřít spustitelné soubory *KinboServer.exe* a *KinboClient.exe*. Než však toto uděláme, je třeba nastavit konfigurační soubory a správné cesty. Zkopírujte si tedy tuto složku na pevný disk vašeho počítače.

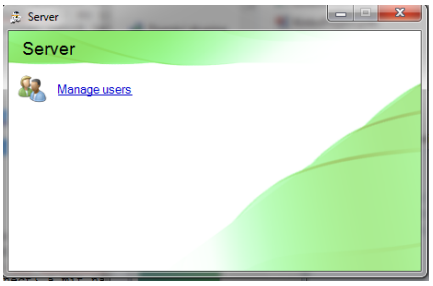
1. Spust'me počítač robota, po spuštění se automaticky zapne modul *Kinbo*. Musíme však nastavit správnou IP adresu serverového počítače v souboru *config.xml*, který se nachází na pracovní ploše. Vypněme tedy po prvním spuštění tento modul a nastavme danou adresu.

2. Máme-li nastavenu správnou adresu, můžeme již spustit samotnou aplikaci *Kinbo.exe* přístupnou také na ploše robota.

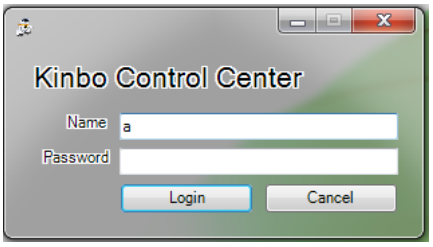
- Nastavme stejnou adresu ve stejně jmenovaném souboru *config.xml* také ve složce „*Release*“, kterou jsme si již předtím zkopírovali na pevný disk.

3. Dále bude potřeba po spuštění *KinboServer.exe* vytvořit nový název projektu a uživ. jméno a heslo k projektu využitím akce programu (viz obr. A.7(a)). Případně je možné zkopírovat z CD složku „*KinboProjects*“

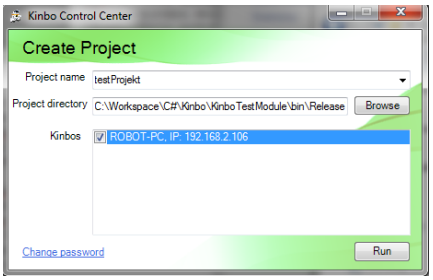
do umístění : „$C : \backslash Temp$“. V dané složce se již nachází vytvořený projekt jehož jméno je *testprojekt* a uživ. jméno pro přihlášení je: „a“, projekt je bez hesla.



(a) Spuštění serveru a možnost správy uživatelských účtů.



(b) Start klienta, nutné zadání uživ. jména a hesla vytvořeného projektu.



(c) Zadání cesty klienta a vybrání požadovaného robota

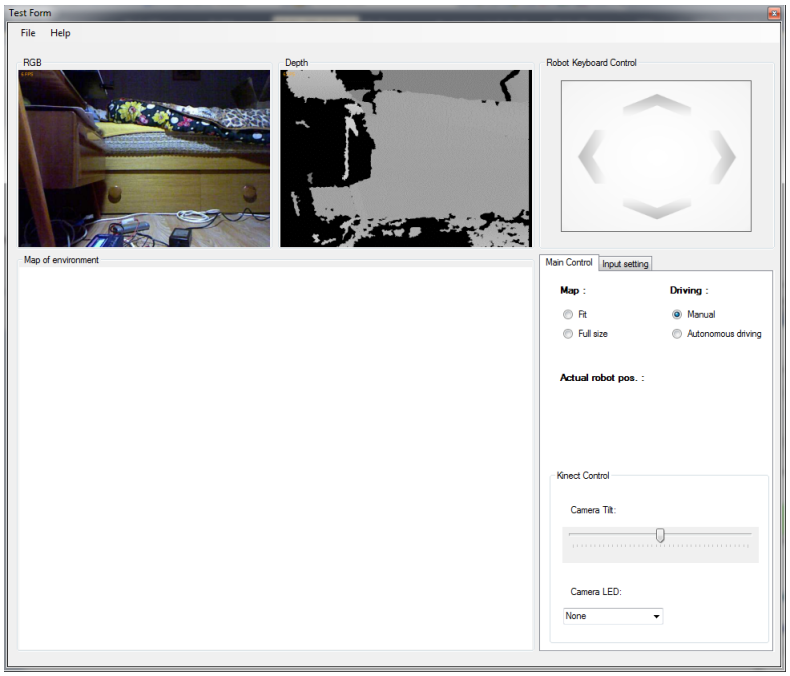
Obrázek A.7: Průběh spuštění modulu klienta platformy Kinbo.

4. Pokud máme spuštěn Server a vytvořený projekt, spusťme klienta *KinboClient* (viz obr. A.7(b)). Zadejme uživatelské jméno a heslo vytvořeného projektu a potvrďme zadání.

5. Nyní stačí zadat jméno vytvořeného projektu, správnou cestu (viz obr. A.7(c)) k přeložené *dll* knihovně našeho vytvořeného klienta. Pokud jme postupovali správně, uvidíme v seznamu aktivního robota. Konkrétní potřebná a již přeložená *dll* knihovna klienta se nachází ve složce:
„ $KinboTestModule \backslash bin \backslash Release \backslash UserModule$“ (více v Příloze C.1 na straně 89).

6. Pokud jsme dosud nenarazili na problém, stiskněme tlačítko *Run*.

7. Na závěr by se již mělo objevit okno s navrženým modulem klienta (viz obr. A.8)



Obrázek A.8: Výsledné uživatelské rozhraní modulu klienta umožňující snadnou obsluhu hardwaru robota.

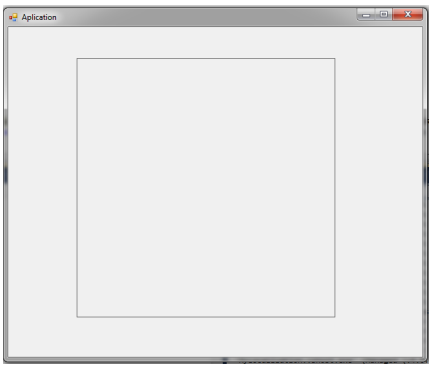
Pokud jsme došli až do posledního kroku a vidíme samotný funkční modul klienta, gratulujeme! Můžete vyzkoušet pohyb robota za pomoci šipek a výstup z kamery. Návrh podoby klienta počítá s budoucí plnou integrací navrženého modulu navigace společně s modulem lokalizace.

A.3 Uživatelská příručka – Návrh lokalizace

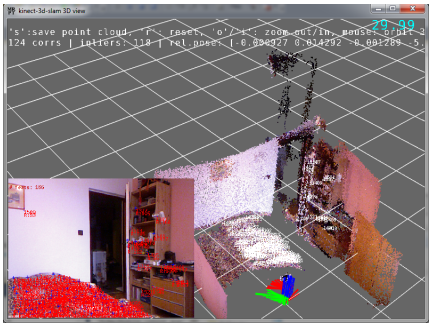
Upozorňuji, že pro běh aplikace je nutný počítač s Windows XP a výše a dále mít nainstalováno v osobním počítači *.NET Framework* o verzi alespoň 4.0. Také je nutné vlastnit kameru *Microsoft Xbox Kinect* a mít nainstalovánu knihovnu *MRPTv1.0.1* a ovladače pro kameru *Freenect*. Obojí možné stáhnout zde: `http://www.mrpt.org/Downloads`. Popišme si nadále ovládání generování mapy této práce.

Důležité je nejdříve spustit server, který vytváří spojení pomocí *Pojmenovaných rour*, tím je samotná C# aplikace s oknem pro vykreslení 2D mapy. Spust'te z CD ze složky (více v Příloze C.1 na straně 89)

„*LocalizationTestModule*“ a následné podložky „*UserTestModule/*“ spustitelný soubor *MyLocalization.exe*. Po správném spuštění uvidíte základní obrazovku s oknem pro vykreslení 2D mapy (viz obr. A.9(a)).
Dále spust'me ze složky „*LocalizationTestModule/Release*“ spustitelný soubor *SLAM.exe* s lokalizací. Pokud jste postupovali dobře, uvidíte okno s Vámi snímanou 3D scénou (viz obr. A.9(b)). Pamatujte, že kamera Kinect musí být umístěna v horizontální poloze nad úrovní země. Poté pro lepší přehled dané scény namapujte část místnosti a dále stiskem klávesy „M“ vytvoříte 2D mapu nasnímané scény. Dojde také k odeslání mapy do C# aplikace (viz obr. A.9(c)) a zobrazení mapy.



(a) Spuštění aplikace s oknem pro vykreslení 2D mapy



(b) 3D mapa scény generovaná knihovnou MRPT

(c) Výsledná 2D mapa scény

Obrázek A.9: Průběh spuštění modulu navržené lokalizace a přenos mapy.

A.3.1 Programátorská příručka

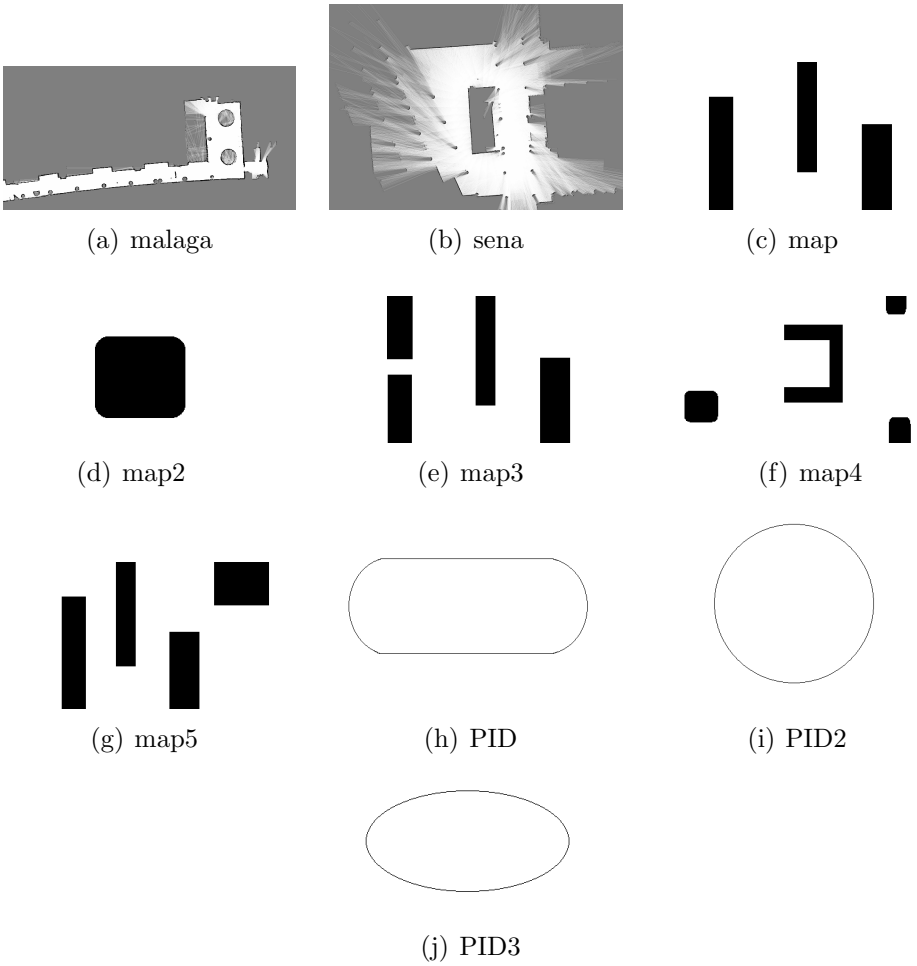
Programátora bude jistě zajímat nastavení cest a všech závislostí pro další vývoj, s použitím kompletně nastaveného projektu pro další vývoj je toto po nainstalování základních požadavků uvedených výše nebude programátor již nic potřebovat. Pokud by ale přeci jen chtěl pokračovat na vývoji v jiných projektech, jsou důležitá tato nastavení:

1. Po pravém kliknutí na *C++* projekt vytvořený v Microsoft Visual Studio je třeba vybrat položku vlastnosti (properties). Protože mám anglickou verzi, budu uvádět dané položky v originálním anglickém znění.
2. Musíme nastavit v záložce „*VC++ Directories*“ položku „*Include Directories*“. V této položce nastavit cesty pro všechny hlavičkové soubory knihovny *MRPT*. Dále ve stejné záložce nastavíme položku „*Library Directories*“, kde nastavíme cesty souborům ve složce *Lib* knihovny *MRPT*.
3. Dále rozbalíme záložku *Input* a zde nastavíme cesty v položce *Additional Dependencies*. Cesty odpovídají všem **lib* knihovnám knihovny *MRPT*

Při nastavování cest na svém projektu je doporučeno podívat se do vlastností poskytnutého, již nastaveného projektu.

Příloha B

B.1 Seznam testovaných map



(a) malaga (b) sena (c) map

(d) map2 (e) map3 (f) map4

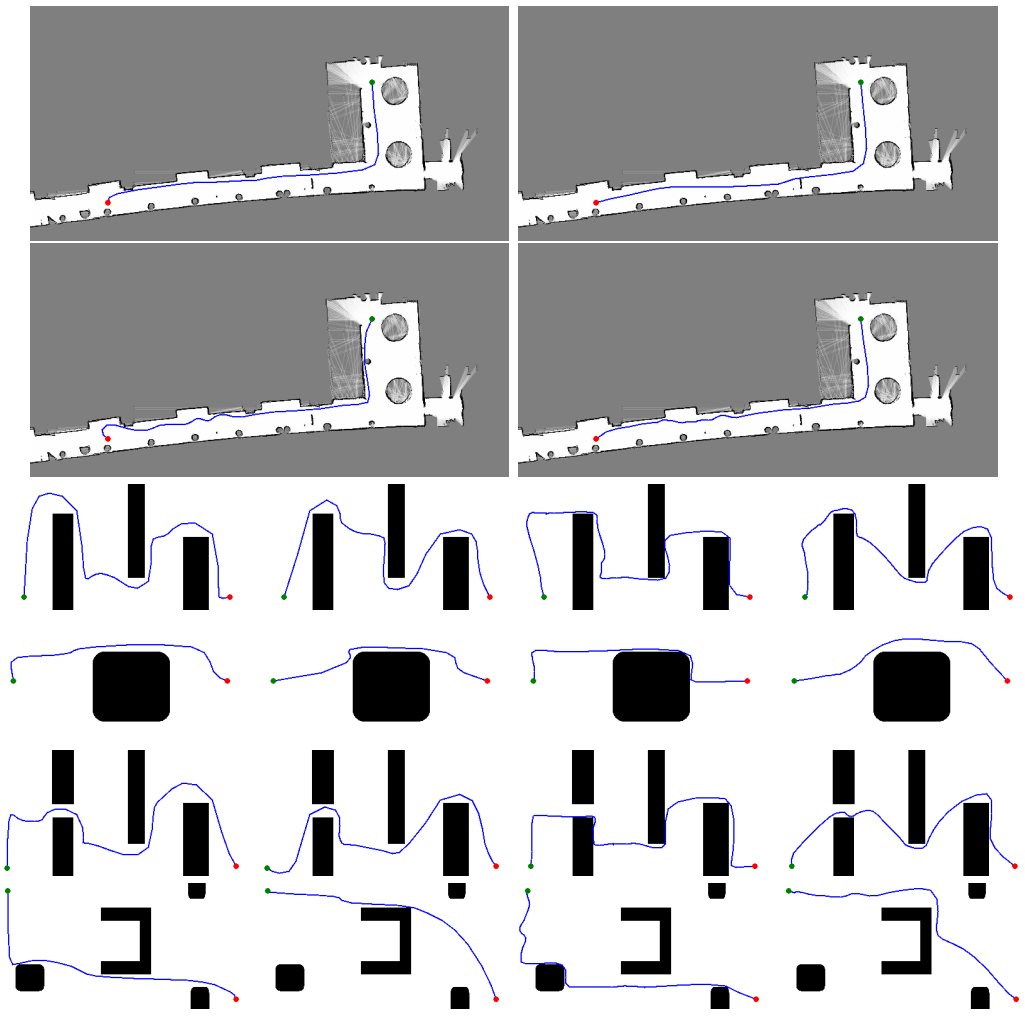
(g) map5 (h) PID (i) PID2

(j) PID3

Obrázek B.1: Seznam testovaných obrázků.

B.2 Výsledky plánů cest

Uvádím zde některé výsledky generování cesty u map, které nebyly uvedeny v textu. Mapy naleznete na CD (více v Příloze C.1 na straně 89) a můžete vlastnoručně zkusit generování cesty (více v Příloze A.1 na straně 75). U každého algoritmu byla snaha najít co nejoptimálnější parametry pro generování cesty na dané mapě.



Obrázek B.2: Zleva doprava jsou vyobrazeny plány cest s použitím: Dijkstrům algoritmus, $A^\star$, dynamické programování, dynam. programování – nedeterministický pohyb.

Jak si lze z uvedených obrázků cest povšimnout, nejlepší cesty podává stabilně algoritmus $A^\star$.

Příloha C

C.1 Obsah CD

Přiložený disk CD obsahuje složky:

- „*Dokumentace*“ – samotná dokumentace k této práci. Složka obsahuje také programátorskou dokumentaci k hlavní aplikaci – Simulace navigace.
- „*Literatura*“ – použitá literatura v elektronické podobě, společně s další užitečnou literaturou.
- „*Software platformy Kinbo*“ – softwarové *API* platformy Kinbo. Složka obsahuje podpůrné programy nutné pro spuštění klientské aplikace navržené pro platformu Kinbo.
  Adresářová struktura složky:
  - „*Kinbo\C-Temp*“ – tato složka obsahuje vytvořený projekt s názvem *testprojekt*,
  - „*Kinbo\inf*“ – ovladače *Freenect* pro kameru Microsoft Xbox Kinect,
  - „*Kinbo\Kinbo*“ – zdrojové soubory platformy Kinbo,
  - „*Kinbo\KinboTestModule*“ – samotný návrh klienta pro platformu Kinbo. Stejná složka se také nalézá ve složce „*src*“, kde jsou pohromadě všechny aplikace, které vzešly z dané práce,
  - „*Kinbo\Release*“ – spustitelné programy platformy Kinbo, které zastřešují *API* platformy a umožňují běh jednotlivých modulů vytvořených pro tuto platformu.
- „*src*“ – tato složka skrývá vlastní projekty aplikací vytvořených v programovacím prostředí *Visual Studio 2010* od Microsoftu. Je zde vždy přeložený program i zdrojové soubory. Dále složka obsahuje zdrojové soubory dokumentace sázené prostřednictvím typografického rozhraní LaTeX.
  Struktura adresáře:

  - „*Dokumentace*“ – dokumentace aplikace,
  - „*Klient pro platformu Kinbo*“ – zdrojové soubory klienta,
  - „*Lokalizace a mapování*“ – zdrojové soubory lokalizace a mapování oblasti s vytvářením 2D mapy a přenosem dat mezi dvěma procesy,
  - „*Simulator*“ – hlavní aplikace Diplomové práce obsahuje zdrojové a přeložené soubory Simulačního softwaru,

- „*Mapy*“ – složka s testovanými obrázky (mapovými podklady).