ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta strojní – Ústav přístrojové a řídicí techniky

BAKALÁŘSKÁ PRÁCE

Modely reprezentace vnějšího prostředí v mobilní robotice

David Janouch **2017**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Janouch** Jméno: **David** Osobní číslo: **437286**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídicí techniky**

Studijní program: **Teoretický základ strojního inženýrství**

Studijní obor: **bez oboru**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Modely reprezentace vnějšího prostředí v mobilní robotice

Název bakalářské práce anglicky:

Representation of the external environment for robot navigation

Pokyny pro vypracování:

Autonomní roboty je možné rozdělit na dvě skupiny. Jedny se řídí pravidly typu: KDYŽ {rozpoznáš situaci}, TAK {proveď akci} ? nazývají se reaktivními. Druhé mají počítačovou reprezentaci vnějšího prostředí ? něco jako mapu. Ta může být robotu dodaná. Či si ji může interaktivně tvořit. Výhodou je možnost inteligentně plánovat trasu. Právě tento typ navigace je předmětem uvažované bakalářské práce. Cílem práce je:
1) Provést rešerši způsobů reprezentace vnějšího prostředí v robotu (robotické mapy)
2) Vyzkoušet jednoduchou neinteraktivní mapu na robotu ze stavebnice LEGO MINDSTORM.

Seznam doporučené literatury:

Petr Krček: Plánování cesty autonomního lokomočního robotu na základě strojového učení, disertační práce, VUT Brno, 2010.
Nahodil a kol.: Teorie a robotické aplikace umělého života. ČVUT 2003.
V. Mařík, O. Štěpánková, and J. Lažanský, Umělá inteligence. Praha: Academia, 1993.

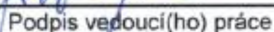
Jméno a pracoviště vedoucí(ho) bakalářské práce:

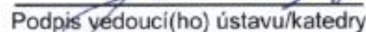
Ing. Mgr. Jakub Jura Ph.D., U12110.3

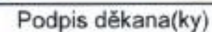
Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **19.04.2017** Termín odevzdání bakalářské práce: **16.06.2017**

Platnost zadání bakalářské práce: ____________


Podpis vedoucí(ho) práce

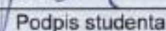

Podpis vedoucí(ho) ústavu/katedry


Podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

19. 4. 2017
Datum převzetí zadání


Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně s tím, že její výsledky mohou být dále použity podle uvážení vedoucího diplomové práce jako jejího spoluautora. Souhlasím také s případnou publikací výsledků diplomové práce nebo její podstatné části, pokud budu uveden jako její spoluautor.

Dne......................... *Podpis ....*

Poděkování

Tímto bych chtěl poděkovat všem, kteří mi pomohli, vyslechli mé názory a snažili se mě posunout dál ve vytváření této bakalářské práce. Osobně bych pak chtěl poděkovat mému vedoucímu Mgr. Ing. Jakubovi Jurovi Ph.D. za konstrukční rady a za konstruktivní připomínky a Šimonu Schierreichovi za pomoc při programovaní.

Abstrakt

Teoretická část této práce popisuje metody lokalizace robota, různé druhy map a samotné hledání cesty robota z místa A do místa B. V praktické části je využito stavebnice Lego Mindstorms EV3 a systému Linux v řídící kostce, naprogramování proběhlo v jazyce Python. Výsledkem je robot, který dokáže dojít z místa A do místa B a vyhýbat se překážkám.

Klíčová slova: Lego Mindstorms, EV3, robot, odometrie, lokalizace, potenciálové pole, mřížka, ev3dev

Abstract

The theoretical part of the bachelor thesis describes methods of localizing a robot, different types of maps and the search for a path. In the practical part a construction set of LEGO Mindstorms EV3 is used together with a Linux operating system uploaded into the brick. Python language is applied for the programming part. This makes the resulting robot capable of walking and avoiding obstacles.

Keywords: Lego Mindstorms, EV3, robot, odometry, localization, potential field, grid, ev3dev

Obsah

1)Úvod do problematiky 8
2)Lokalizace robota 9
6.1)Relativní 9
a)Odometrie 9
6.2)Absolutní 12
a)Aktivní orientační body 13
b)Pasivní orientační body 13
3)Mapy pro roboty 14
3.1)Senzorická mapa 14
3.2)Geometrická mapa 14
3.3)Topologická mapa 15
4)Plánování cesty 16
4.1)Plánování na mřížce 16
a)Potenciálová pole 16
b)Potenciálové pole se šířením tepla 17
c)Dijkstra 17
d) A* 18
e) B* 18
4.2)Exaktní plánování 18
a)Lichoběžníková dekompozice 19
b)Graf viditelnosti 20
c)Voroného diagramy 20
4.3)Pravděpodobnostní plánování 21
5)Závěr rešeršní části 22
6)Praktická část 23
6.1)Výběr lokalizace a mapy pro našeho robota 23
6.2)Problém se softwarem od Lego Mindstorms 23
6.3)Kompilace Pythonu do jazyka Lego Mindstorms 24
a)Návod na nahrání Debianu Linuxu do kostky Lego Mindstorms 24
b)Propojení kostky s počítačem 28
c)Nastavení statické IP adresy 29
d)Nastavení výchozího programu pro úpravu skriptů – PyCharm (JetBrains) 32
e)Zapínaní samotných skriptů z Lego kostky 33
6.4)Samotná konstrukce robota 33
6.5)Odometrie 35
6.6)Vytvoření mapy 38
6.7)Pohyb robota po mapě – Potenciálová pole 39

7.Závěr 41
8.Seznam použité literatury 42
9.Seznam obrázků 43

1) Úvod do problematiky

Co si představit pod tímto názvem? To je dobrá prvotní otázka. Vždy, když jsem někomu řekl název mého zadání, tak si neuměl představit o co se jedná a jeho první otázka zněla: „To znamená co?“.

Pokusím se popsat, jak to vnímám já. Každý člověk potřebuje mapu, aby někam trefil. To je jasné. Další, co potřebuje je nějaký talent či rozum, jak mapu používat a podle čeho se orientovat. Poslední důležitou věcí je, aby člověk věděl, kde se na mapě nachází a kam chce dojít. Bez toho by nám talent hledání v mapě a mapa samotná byli k ničemu. Ideálně, aby člověk věděl v každém momentu, kde je. Nejen si to myslet, což se dost často stává. Po nějaké době se nám totiž začne mapa s realitou neshodovat a poté zjistíme, že jsme ztraceni.

To je v podstatě reprezentace vnějšího prostředí u člověka. Úplně stejná bude vnější reprezentace prostředí u robota. Budeme mu muset dát počáteční a koncovou pozici. Dát mu také mapu, která bude odpovídat skutečnosti a v ní mu vyznačit místa kam nemůže, jinak řečeno překážky. Poslední věcí je náš um. U robota to bude jeho algoritmus, podle kterého najde cestu. Zase stejné jako u člověka. Buď cestu krátkou nebo bezpečnou. Stejné jako v realitě. Také u robota je dobré, když v každém momentě bude vědět, kde se nachází a nebude si to jen myslet.

V této bakalářské práci se pokusím přiblížit metody reprezentace vnějšího prostředí v mobilní robotice. Pokusím se rozepsat různé druhy lokalizace, map a hledání cest.

Mým hlavním úkolem bude sestrojit robota, který dokáže jít podle zadané mapy. Dokáže dojet ze startu do cíle, a přitom se vyhýbat překážkám.

Budu pracovat se stavebnicí Lego Mindstorms. Tato stavebnice má mimo jiné různé senzory a motory. Hlavní součástí stavebnice je chytrá kostka, do které se zapojují senzory a motory. Tato kostka se dá programovat a ovládat pomocí softwaru od Lego Mindstorms. Nejnovější verze kostky nese označení EV3 a s touto kostkou zde budu pracovat a programovat ji.

V práci se pokusím nastínit možné problémy jednotlivých technik a samotné nedostatky stavebnice a budu se snažit tyto nedostatky překonat a vyřešit problémy, které nastanou.

Začnu tedy samotnou rešeršní částí, kde nastíním jednotlivé druhy a techniky a dále budu pokračovat praktickou částí, která by měla ukázat, jak se dá takový robot postavit a naprogramovat.

Přejdu tedy rovnou k rešeršní části této bakalářské práce.

2) Lokalizace robota

Nejdůležitější informací je pro mě poloha robota. Samozřejmě, když nepočítám cíl. Ještě důležitější je, ale tato informace pro samotného robota, aby se mohl rozhodnout, kudy se vydá na svou cestu. Jedná se v podstatě o to, aby robot věděl, kde se na mapě nachází a odkud vlastně hledá cestu do svého cíle. K zjišťování své polohy používá senzory, které jsou jeho součástí. Mnohdy také využívá právě okolí, kterým projíždí. Je nespočet variant, které umožní najít robotovi svou polohu. Snažil jsem se vybrat metody, které mě zaujaly, a hlavně se mi zdály použitelné pro náš případ. Samotné metody se dělí na absolutní a relativní.

6.1) Relativní

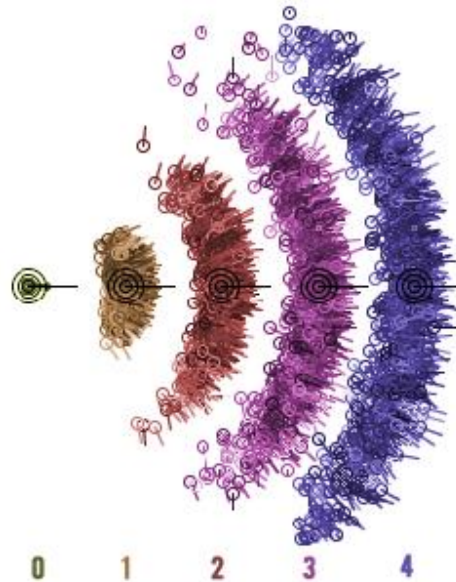
Už z názvu je jasné, že se nedá přesně určit absolutní poloha robota. Jedná se o metody, které jsou závislé na každém pohybu robota. Tyto metody se snaží měřit či odhadovat pozici robota. Tedy vnáší se do nich každý pohyb, každé potočení. Ovšem to sebou nese i následek vzniku nepřesností. Jelikož pohyby na sebe navazují, a tedy i z nich vyrůstající chyba se akumuluje a zvětšuje. Z toho mi vyplývá, že se tyto metody hodí spíše pro krátkodobé sledování robota a jeho pozice. Nejčastěji se jako zástupce relativní lokalizace robota udává odometrie.

a) Odometrie

Odometrie je relativní metoda pro lokalizaci robota v prostoru. Pojmenování odometrie je v podstatě utvořeno složením řeckých slov hodos (cesta) a metron (měřit). Základní myšlenkou odometrie je v podstatě číst ze senzorů, jak se otáčí kola a z toho si následně dopočítat hledanou polohu. Odometrie také předpokládá, že jakékoli pootočení kola je možné převést na posunutí robota v rovině. Zde vidím, že odečet ze senzorů nebude bez chyby a také čím déle se bude robot pohybovat tím více zanáším do hledání polohy chybu.

Senzorům používaných pro odečet otáčení kol se říká encodery. Je velké množství typů a druhů encoderů. Ovšem pro svou cenu a spolehlivost se v mobilní robotice používají optické. Stejně tak u mého motoru, který budu používat, je použit optický encoder.

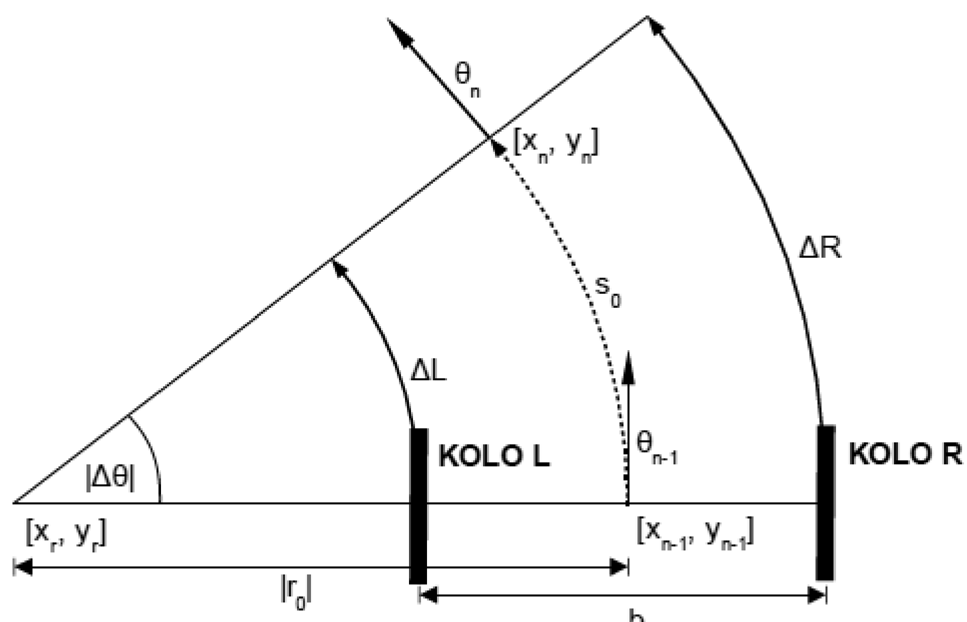
Za jisté je však daleko více příčin vytvoření chyby. Jsou to drobné nepřesnosti, které ovšem v konečném důsledku velmi ovlivňují odhad konečné pozice robota. Uvedu jen pár příčin chyb, které jsou dle mého úsudku nejzásadnější. Rozdíl mezi skutečným a naměřeným obvodem kola, rozdíl mezi skutečnou a naměřenou hodnotou rozchodu kola. Samotné měření encoderu u každého kola a nedokonalý povrch (nerovnost, druh povrchu). Tyto chyby můžu rozdělovat na systematické (chybu můžu předvídat a trochu ji korigovat) a nesystematické (jsou náhodné a nemůžu je předvídat). [4]



Obrázek 2.1: Ilustrace chyby, která roste s ujetou vzdáleností a která je způsobena drobnými rozdíly mezi nominálním a skutečným obvodem levého a pravého kola. Vzorky označují potenciální polohu robota při jízdě po (dle měření odometrie) přímé dráze, barevně jsou odlišeny jednotlivé kroky 0 až 4

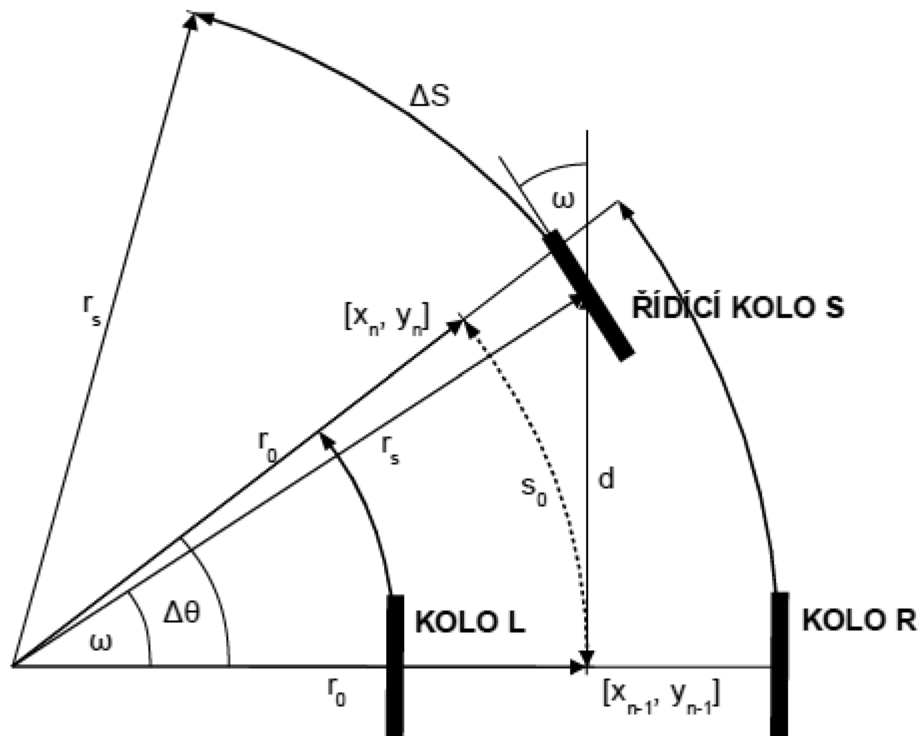
Další velmi důležitým předpokladem je konstrukce robota. Musím zde uvážit do jaké „škatulky“ bude náš robot spadat. Jelikož pro každý typ konstrukce je důležité mít správné rovnice pro výpočet polohy.

Prvním typem, o kterém se zmíním budou diferenčně řízená vozidla. Co se týká konstrukce robota, tak jsou zde dvě na sobě nezávislé nápravy. Pro upřesnění, pokud se kola na nápravách budou otáčet stejnou rychlostí a stejně dlouho, pojede robot rovně. Pokud bude zatáčet, tak se bude otáčet kolem své osy a v pohybu bude pouze jedno z kol. [4]



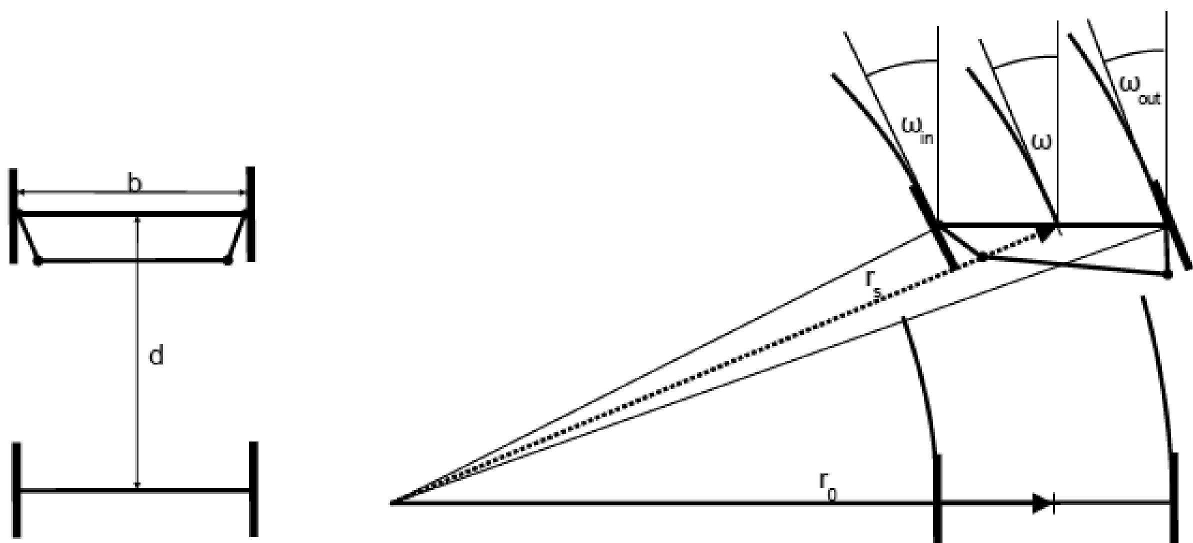
Obrázek 2.2: Diferenčně řízené vozidlo

Dalším typem konstrukce robota může být tříkolka, která se typově velmi podobá i autu. Jak auto, tak tříkolka má v podstatě dvě nápravy pevné a jednu respektive dvě nápravy řízené(otočné). To, jak bude tříkolka zatáčet určuje pouze jedno řiditelné kolečko. Tedy dokáže se buď pohybovat po kružnici anebo po přímce. [4]



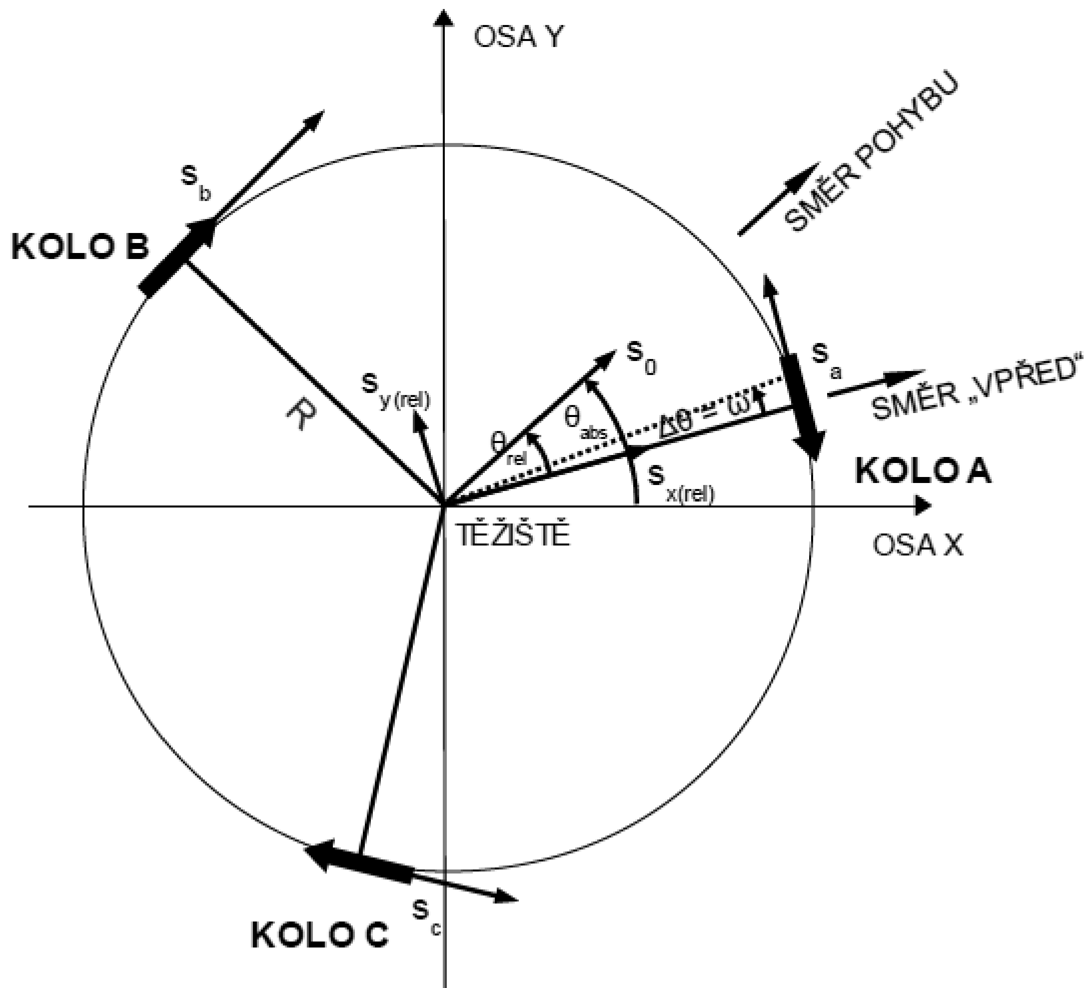
Obrázek 2.3: Vozidlo s pojezdem typu tříkolka

Co se týká samotného auta, tak se toto řízení nazývá Ackermanovo řízení. Aby byl hladký průjezd bez smýkání kol, tak je potřeba, aby zde platila podmínka: Kolo, které se pohybuje po menší kružnici, tedy kolo vnitřní bylo od podélné osy vozidla vytočené více než kolo, které se pohybuje po větší kružnici, tedy kolo vnější. [4]



Obrázek 2.4: Ackermanovo řízení

Poslední typ, který zmíním je pro všesměrové roboty. Jednoduchá definice pro všesměrové roboty je, že počet stupňů volnosti rychlostí odpovídá počtu stupňů volnosti pozice. Čili tento robot může měnit svou rychlost bez ohledu na směr. Tento typ nalezneme velmi často pod anglickým názvem omnidirectional drive. Asi nejjednodušeji lze ukázat tento příklad na tříkolovém všesměrovém robotu, který má tedy všechny tři kola všesměrová a pohaněná umístěná na kružnici po 120°. [4]



Obrázek 2.5: Nejjednoduší konfigurace pojezdu všesměrového robota se třemi koly

Odometrie je velmi často užívaná. Má svoje klady, ale i zápory. Kladem je její jednoduchost, záporem je samozřejmě její nepřesnost. Je důležité s ohledem na odometrii myslet i na samotnou konstrukci robota a jeho podvozku.

Existují také i jiné metody relativní lokalizace pro příklad jich pár uvedu, ale nebudu je dále popisovat, jelikož jsou méně používané. Senzory využívající Dopplerova jevu, Navigace výpočtem (angl. Dead Reckoning), Inerciální navigační systémy a další.

6.2) Absolutní

Absolutní lokalizace robota mi poskytuje absolutní pozici, která není ovlivněna žádným předchozím stavem či dějem. Samozřejmě, jak relativní lokalizace má své mouchy, tak i absolutní lokalizace není bez chyby. Velká výhoda je v tom, že se zde neakumuluje chyba.

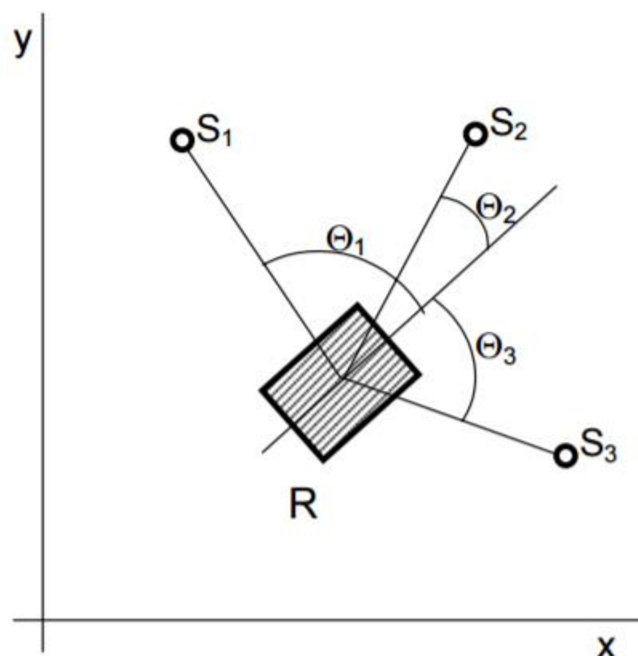
Avšak na rozdíl od relativní je velmi náročná na výpočet. Dále je pro mě také omezující prostředí, ve kterém se robot může pohybovat. Jako reprezentující metody zde uvedu aktivní a pasivní orientační body.

Asi nejčastěji používanou metodou absolutní, je určovaní polohy pomocí orientačních bodů. Za orientační bod se dá považovat cokoli v prostředí, co robot dokáže svými senzory zachytit a pozice toho bodu je robotovi známá. Pokud je těchto orientačních bodů více, tak si robot dokáže díky metodám pro dopočet své polohy, svou polohu určit. Dále nastíním rozdíl a základní charakteristiku aktivních a pasivních orientačních bodů.

a) Aktivní orientační body

Název napovídá, že se bude jednat o body, které budou nějak aktivní. Tyto body, velmi často označované jako majáky, vysílají nějaký signál směrem k robotovi, který má senzor pro zachycení signálu. Tento signál (informaci) umí dále zpracovat. Spadají sem i majáčky, které naopak signál, který vysílá samotný robot, přijímají a jiným signálem odpovídají.

Velká přednost této metody je velmi lehká lokalizace, jelikož příjem signálu není pro robota těžký úkol. Ovšem jsou zde i nevýhody např. velmi složité prostředí, kde se signál nešíří dobře, příjem špatného či odraženého signálu a v neposlední řadě i cena, jak pořizovací, tak i náklady na údržbu a provoz majáků.



Obrázek 2.6: Aktivní majáky

b) Pasivní orientační body

Pasivní orientační body nevysílají ani nepřijímají žádný signál. Jedná se o prvky, které je možné detekovat nějakým senzorem. Jejich poloha musí být ale stálá a známá. Je jasné, že rozpoznávání takových prvků nebude jednoduché. Je to velice náchylné k chybě a výpočetně

velmi náročné. Což nejsou malé zápory, ovšem ty jsou vyvážené nižšími náklady na samotné prvky a jejich údržbu.

Tyto body rozděluji do dvou skupin na umělé a přirozené. Umělé jsou uměle vytvořené a rozmístěné tak, aby byly dobře identifikovatelné pro robota a dopočet polohy tak jednoduchý. Velmi často se používají retroreflexní plochy, čárové kódy anebo kontrastní geometrické obrazce. Zatímco přirozené jsou body samotného prostředí, které jsou dobře rozpoznatelné a dostatečně odlišné od okolí. Může to být v podstatě cokoli od stromů až po rohy místností. Velkou předností je, že se prostředí nemusí měnit a přizpůsobovat robotovi.

Jen pro dokreslení je zde dalších mnoho metod relativní lokalizace polohy např. kontinuální lokalizace, měření orientace a další.

Úplně na závěr je dobré podotknout, že v praxi se nejčastěji pracuje nejen s jednou metodou, ale i s více metodami dohromady. Velmi častým jevem je propojení aktivní lokalizace s pasivní lokalizací. Důležité je minimalizovat možné chyby, a proto se tyto metody propojují. Např. je časté a možné propojit odometrii s aktivními majáky, kde použijeme i nějaké pasivní body k určení polohy robota.

3) Mapy pro roboty

Další velmi důležitou kapitolou jsou druhy map pro roboty. Mapu můžu brát naprosto stejně jako běžnou mapu, kterou používám pro naší orientaci. Robot už ví, kde se nachází a teď potřebuje informace o tom kudy může jít a kam má vlastně dojít. Toto mu umožní mapa. Druhů map je, jak ve skutečném životě, tak i v tom robotickém mnoho. Zmíním se tu o dvou metrických a topologických mapách.

3.1) Senzorická mapa

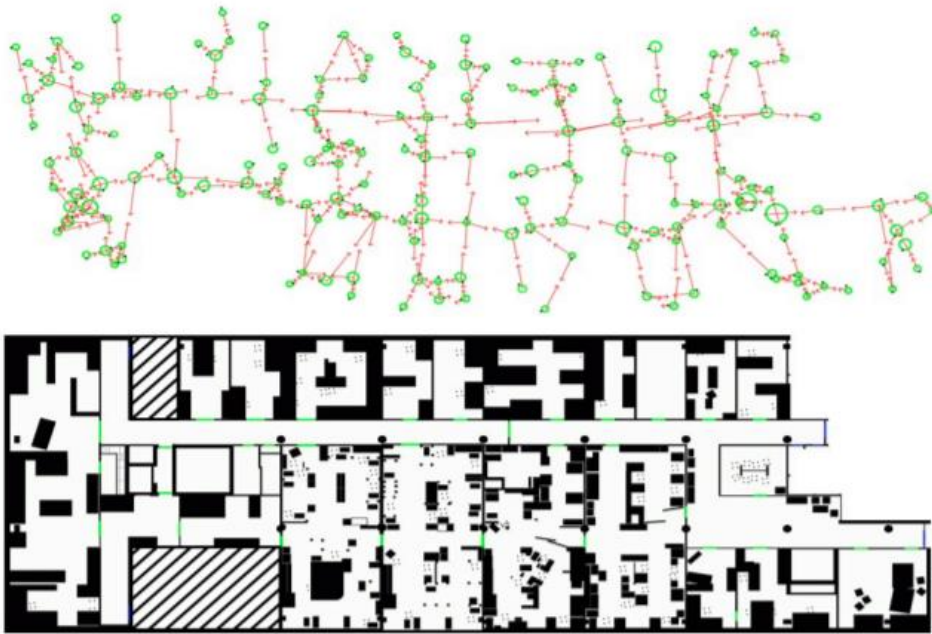
Je v podstatě nejnižší možná reprezentace prostředí. Jsou zde pouze senzorická data. Velmi často je mapa tvořena do mřížky, kde se rozlišuje, která mřížka je obsazená (je v ní překážka) anebo prázdná (je zde možný průjezd). Jasnou předností je snadná tvorba takové mapy. Ovšem nevýhodou je vysoká náročnost na paměť a také mnohdy nebývá nejlepší na další použití jako je plánování trasy. [3]

3.2) Geometrická mapa

U těchto druhů map se prostředí reprezentuje pomocí geometrických útvarů. Např. přímky, polygony, kružnice a další. Tyto mapy jsou abstrakcí prostředí. Jsou náročnější na tvorbu a na výpočetní techniku, avšak plánování a další práce je zde jednoduší. Zajímavé je, že není tak náročná na paměť. Je to zřejmě díky abstrakci. [3]

3.3) Topologická mapa

Poslední je mapa topologická. Jedná se ještě o větší abstrakci než u předchozího typu. Sestavuje se na základě nějakých významných vlastnostech. Tyto mapy mají formu grafu. Jednotlivé uzly na grafu představují právě ty významné vlastnosti. Hrany mezi těmito uzly jsou vztahy mezi těmito vlastnostmi, což se dá považovat, jak se robot přemístí z jednoho uzlu do druhého. Velkou výhodou je, že mapy jsou naprosto nenáročné na paměť. Nemusí obsahovat, žádné geometrické informace. Je však těžké určit významné body, tak aby je robot jednoduše rozpoznal. Ovšem je fakt, že plánování na takovéto mapě nejjednodušší. [3]



Obrázek 3.1: Převod mapy do topologické místnosti

Je velice důležité mít rozmyšlené, jak s mapou budu nakládat dále, jelikož pro různé plánování jsou vhodné různé mapy. Tedy je potřeba vědět, jaké plánování zvolím. Těchto variant je neskutečně mnoho.

4) Plánování cesty

Plánování cesty záleží hlavně na zvolné mapě. Je mnoho variant a metod, jak naleznout tu správnou cestu z bodu A do bodu B. Pro mě by byl samozřejmě nejvýhodnější nějaký způsob, který by dokázal být rychlý, výpočetně nenáročný, nenáročný na paměť a v neposlední řadě přesný. Některý, který by hledal, pokud možno nejkratší a nejbezpečnější cestu. I zde však musím dělat kompromisy a vždy vybrat nejvíce uspokojivé řešení, které na druhé straně nebude ideální, ale pro můj případ dostačující.

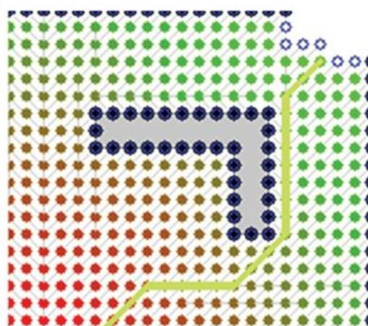
Jak jsem již výše zmínil, metod plánování je obrovská řada. Můžu jej rozdělit do třech hlavních typů: plánování na mřížce, exaktní plánování a pravděpodobnostní plánování cesty. Mě bude nejvíce zajímat plánování na mřížce a potažmo druhy tohoto plánování. Zmíním se však o všech typech plánování a o jejich nejzajímavějších podtypech.

4.1) Plánování na mřížce

Jak název napovídá, bude hlavní mapou nějaká mřížka. Rastrové mapy nebo pravděpodobnostní mřížky pro mě budou mapou. V mnoha případech se můžu s těmito typy setkat i v počítačové grafice, jelikož na pravděpodobnostní mřížku je možné nahlížet jako na obraz. V podstatě můžu říci, že rastrová mapa spadá do škatulky senzorických map. Pravděpodobnostní mřížky mně pomáhají v prostoru, kde si nejsem jistý, zda překážka je či není. Lze to řešit jednoduše, když si nejsem jistý beru, že tam překážka je. Ale výhodou těchto pravděpodobnostních mřížek je, že pokud bych chtěl prostor dodefinovat na místě a mapu poupravit. Např. kdyby mapa nebyla dopředu známá. [6]

a) Potenciálová pole

Prvním zástupcem plánování na mřížce, jsou potenciálová pole. V podstatě se jedná o naprosto jednoduchou myšlenku, kde se každé buňce přidělí hodnota. A robot pak už jen v každé buňce porovnává svou hodnotu s okolními hodnotami. Velmi často se nejvyšší hodnota přiřazuje ke startu a nejnižší hodnota k cíli. Překážky mají hodnoty velmi vysoko. Vyšší, než je samotný cíl, aby nehrozilo, že robot nabourá. Robota v podstatě nechám jezdit do té doby než „uvízne“, což můžeme brát jako cíl. [6]



Obrázek 4.1: Ukázka potenciálového pole

Tento princip je velmi jednoduchý, jak na výpočetní techniku, tak na samotné programování. Má však i svá úskalí. Tím největším je asi to, že se robot může zaseknout (uvíznout), ne přímo v cíli, což je náš požadavek, ale v nějakém lokálním minimu. Tato vada, se může ale jednoduše odstranit, mou úvahou. Pokud by se opravdu nacházelo na mapě jiné minimum než minimum v cíli. Musím tedy očíslovat buňky tak, aby se nám nemohlo stát, že hodnota v předvídaném minimu, je opravdu minimální. Zadáme tam tedy hodnotu větší. Dobře toto ilustruje obrázek číslo 4.2, kde je vyznačeno fialové pole, které by jinak představovalo lokální minimum.

9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	1
9	8	7	6	5	4	3	2	2	2
9	8	7					3	3	3
9	8	8	8	9	10		4	4	4
9	9	9	9	9	9		5	5	5
10	10	10	10	9	8		6	6	6
11	11	11	10	9	8	7	7	7	7
	12	11	10	9	8	8	8	8	8
		11	10	9	9	9	9	9	9

Obrázek 4.2: Tzv. metoda záplavového vyplňování (wavefront-expansion)

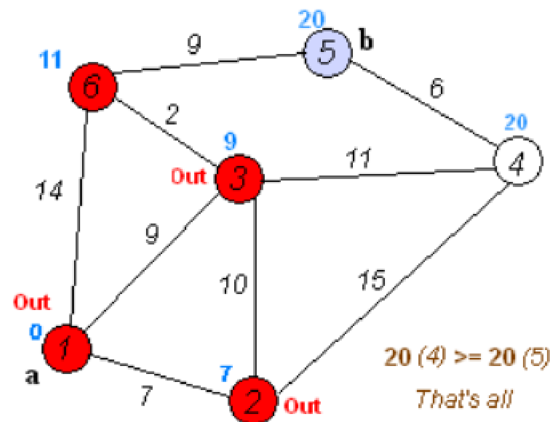
b) Potenciálové pole se šířením tepla

Dalším velmi zajímavým rozšířením metody potenciálového pole, je potenciálové pole se šířením tepla. Tato velmi zajímavá myšlenka vylepšuje předchozí ideu. Cesta pro tuto metodu by měla být nejen kratší, ale i výhodnější a lépe vyhledatelná. V podstatě celá myšlenka je založená na šíření tepla. Jediný zdroj tepla pro mě bude představovat právě cíl, kam mám dorazit, zatímco všechny překážky pro mě budou velmi chladné, takové, které nepřijímají teplo. Ostatní pole se budou ohřívat díky šíření tepla. Čím teplejší bude pole, tím budeme blíže k cíli. Tato metoda bude trochu více náročnější na programování a výpočetní techniku než moje klasické potenciálové pole, avšak mnohdy přesnější. A většinou mi poskytne lepší cestu. [6]

c) Dijkstra

Tato metoda se opírá o takzvaný Dijkstrův algoritmus. Základní myšlenkou tohoto algoritmu je, že hledá v grafu nejkratší možnou cestu z jednoho vrcholu do druhého. Přiřadím prvnímu uzlu nulovou hodnotu a poté další uzly (vrcholy) mají hodnotu nekonečno. Nekonečno, jelikož ještě nebyly navštívené, tudíž nemají přiřazenou hodnotu. Všechny vzdálenosti mezi jednotlivými uzly znám. Začneme tedy v uzlu s hodnotou nula a poté se dívám na vedlejší uzly. Spočítám všechny cesty do dalších uzlů a vyberu nejkratší cestu a vydám se do tohoto uzlu. Všem ostatním přiřadím hodnotu, kterou jsem vypočítal. A znovu hledám cestu nejkratší, tedy k uzlu, ve kterém jsem, přičtu hodnoty cest do dalších uzlů. Pokud jsem našel cestu z uzlu do jiného uzlu kratší, tak v tom uzlu přepíši hodnotu na tu kratší, pokud ne, hodnota mi zůstane. Navštívené uzly označuji jako navštívené a dále do nich nevstupuji. Takto můžu najít nejkratší

cestu z počátečního uzlu do jakéhokoli uzlu. Tento algoritmus končí, pokud už nemám další uzel, který lze navštívit. [8]



Obrázek 4.3: Konec Dijkstrova algoritmu. Všechny cesty mezi uzly jsou spočítané a nejkratší

I zde je ovšem hned několik nevýhod. První je, že nemůžu mít hrany se zápornou hodnotou, což pro mě asi není úplně zásadní nevýhoda. Ovšem druhá je pro mě dosti důležitá. Tento algoritmus totiž počítá všechny cesty mezi uzly. Tudíž je zbytečně náročný na výpočetní techniku a dává mi i informace, které pro mě nejsou významné a zabírají mi zbytečně paměť. Toto lze zmírnit různými modifikacemi.

d) A*

A* lze do jisté míry nazývat modifikací či vylepšením Dijkstrova algoritmu. Abych zjednodušil předchozí algoritmus, musím vyřazovat cesty, kterými se nevydám. Zanesením heuristické proměnné. V podstatě k vypočítané cestě z počátečního uzlu do nějakého dalšího přičtu právě tuto heuristickou hodnotu. Pro příklad můžu brát jako tuto hodnotu vzdušnou čáru. Tímto pak můžu udávat prioritu určitým cestám. Nejmenší výsledná hodnota bude mít největší prioritu. [7]

e) B*

Pokud to s určováním priority podle heuristické hodnoty přeženu, tedy pokud budu brát v potaz hlavně tuto hodnotu, pak tento algoritmus bude mnohem více agresivnější, ale daleko více náchylnější k omylu. Hlavně pokud se bude jednat o složitý a velký prostor.

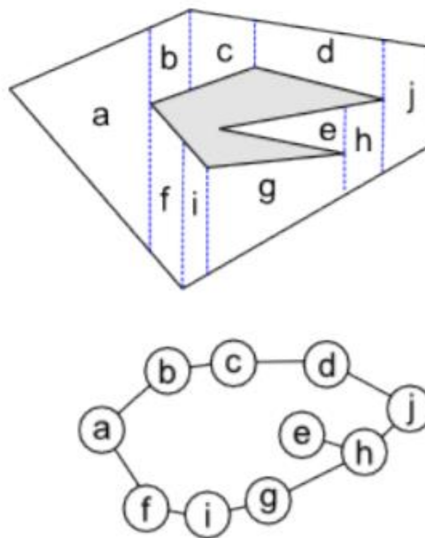
4.2) Exaktní plánování

Při pročítání může čtenáře pravděpodobně napadnout, proč jsem nezačal s „přesnými“ neboli s exaktními metodami rovnou. Nebo možná i to, proč zde zmiňuji nepřesné metody. Nazývat metody nepřesnými není úplně správná myšlenka. Lépe řečeno, možná méně spolehlivé. I tak to musím brát v uvozovkách. Řadím je podle mého úsudku použitelnosti, jednotlivých typů plánování. Takže proto jsem začal plánováním na mřížce.

Tímto ukončím obhajování svého postupu a pustíme se do toho, co je vlastně myšleno exaktním plánováním. „Jedná se o kategorii plánovacích algoritmů, které najdou cestu vždy, pokud existuje, a v opačném případě ověří, že neexistuje." [1] Samozřejmě opět tento typ dělím do dalších skupin. Do skupin, kde se rozděluje prostor do jednodušších částí a druhá skupina, vytváří mapu cest. Nyní se blíže podíváme na jednotlivé skupiny

a) Lichoběžníková dekompozice

Lichoběžníková dekompozice je zástupcem první zmiňované skupiny, tedy rozděluje plochu do jednodušších částí. Z názvu lze usoudit, že se nebude dělit na jakékoli časti, ale na lichoběžníky. Proč zrovna na lichoběžníky? Jelikož rozdělit prostor s překážkami na lichoběžníky je velmi jednoduché. [5] Vytváření je vidět na obrázku 4.4.



Obrázek 4.4: Prostor rozdělený lichoběžníkovou dekompozicí a výsledný graf souslednosti

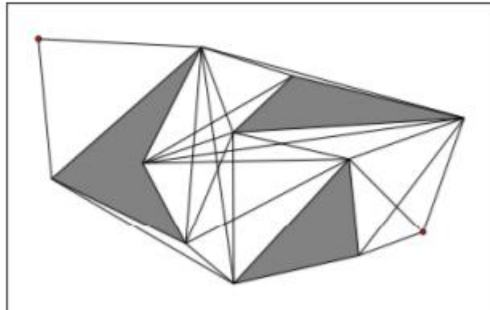
V podstatě rozdělím prostor rovnoběžkami. Jsou zde dvě pravidla. První pravidlo, které musím splnit je, aby některé z nich procházely každým vrcholem překážek, za které považuji i stěny. Druhé, aby byly kolmé na spojnici bodu startu a cíle robota. Někdy námi může vzniknout i trojúhelník, tedy „zmutovaný" lichoběžník.

Prostor mám tedy rozdělený. Teď v podstatě musím vymyslet, jak toho využít. Není to těžké, pokud se cíl bude nacházet v jednom z lichoběžníků, tak robotovy řekneme, aby šel přes lichoběžníky a, b, c a d, protože cíl leží v j. (viz. obrázek 4.4) Nejkratší cestu lze nalézt tak, že se vezmou středy jednotlivých úseček (hran), které mi vznikli z přímek.

Uvažoval jsem, že překážky jsou polygonální, jelikož pro nepolygonální překážky tuto metodu použit nelze. Tato metoda je velmi jednoduchá a z toho vyplývá, že i poměrně rychlá.

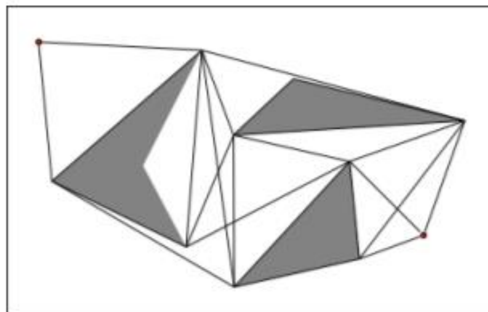
b) Graf viditelnosti

Zástupcem druhé zmiňované skupiny, tedy skupiny, která vytváří mapu cest, je právě graf viditelnosti. Tato metoda mi dokáže najít nejkratší cestu. Vrcholy zde tvoří start, cíl a vrcholy překážek. Všechny vrcholy spojím tak, aby žádná úsečka neprocházela žádnou překážkou.



Obrázek 4.5: Graf viditelnosti

Z obrázku je zřejmé, že vzniklých hran (úseček) je obrovské množství. Jak ovšem poznat, které hrany odstranit a které ne? Inu musím vědět, které jsou důležité. Důležité jsou takové, které můžu protáhnou za vrchol, aniž by mi zasáhly do překážky. Všechny ostatní, které do překážky vniknou, odstraníme. [5]



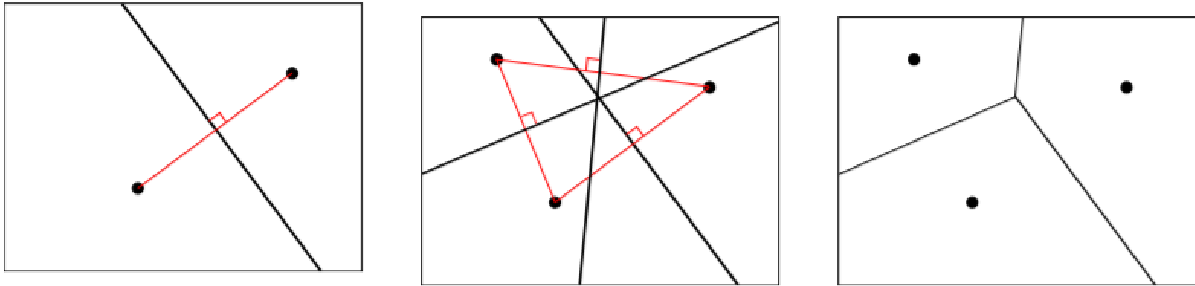
Obrázek 4.6: Redukovaný graf viditelnosti o nepotřebné hrany

Výsledné hrany mi v podstatě dávají samotnou cestu pro robota. Zde vidím, že je vlastně počítáno s robotem jako s bodem, což není úplně dobré. Další nevýhodou budou kulaté překážky. Tato metoda počítá pouze s hranatými. Lze udělat z kulatého tělesa polygon, ale tím nám vznikne obrovské množství hran, a jak lze jednoduše odvodit, čím více hran, tím více výpočtů.

c) Voroného diagramy

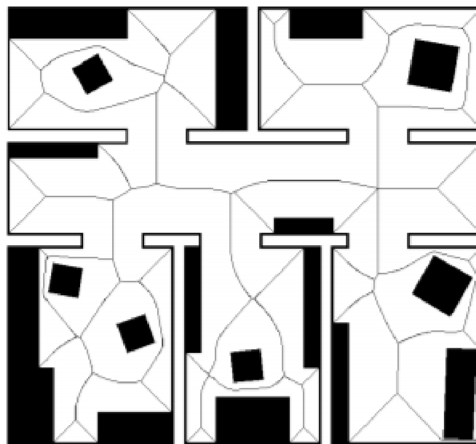
V předchozích metodách mi v podstatě šlo hlavně o to najít cestu nejkratší. To je běžné i v normálním životě. Být, co nejrychleji využitím nejkratší cesty v cíli. Ovšem i v robotice můžu potřebovat najít, co nejbezpečnější cestu. Tedy neriskovat, že narazím do překážky, či třeba nespadnu ze skály.

Pokusím se zde popsat, v čem spočívají Voroného diagramy. Představím si, že budu mít více bodů, které mi něco udávají. Například policejní stanice. Budu mít tři policejní stanice a budu chtít pomocí těchto diagramů rozdělit okolí, které bude spadat pod jednotlivé policejní stanice. Rozdělím je tak, že spojnicí bodů povedu osu. Tímto se mi prostor rozdělí na části. [5] Jako je vidět na obrázku 4.7.



Obrázek 4.7: Tvorba Voroného diagramu

Mám Voroného diagram. Vím, jak ho zkonstruovat, ale jak ho budu používat pro plánování trasy? V podstatě body budou moje překážky. Musím brát v potaz, že mapa je pro mě grafem, kde mám vyznačené body, moje překážky. No a tímto vlastně dostávám cestu mezi překážkami. Jak vidím, nebo si domyslím, tak tato cesta nebude nejrychlejší, jak jsem předpokládal, ale zaručeně bude ta nejbezpečnější.



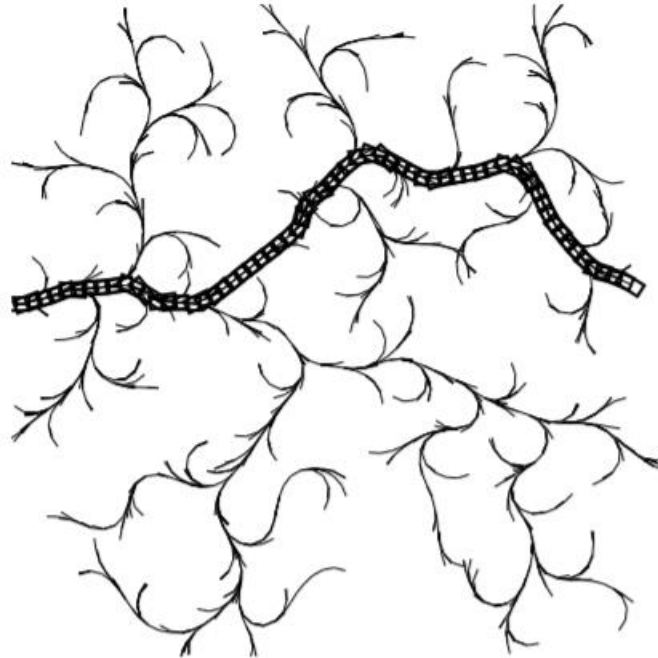
Obrázek 4.8: Voroného diagramy použité v místnosti

4.3) Pravděpodobnostní plánování

K tomuto typu plánování budu velmi stručný, je to hlavně z důvodů složitosti a z důvodů nepoužitelnosti pro moji relativně jednoduchou praktickou část. Tím však rozhodně nehodlám snižovat význam toho typu plánování. Jedná se o typ, který se používá pro roboty, kteří nejsou konvexní a musí se tedy otáčet např. auta. Další možnost využití je, pokud budu brát v potaz, že robot je v 3D světě a to on opravdu je. Všechny metody jsou více složité, a tudíž se musí nějak zjednodušit. Takže v podstatě velmi často převádíme popis z exaktního do pravděpodobnostního. U těchto metod záleží dost i na konfiguračním prostoru a na tom, jak si ho zvolíme. Rád bych zde zmínil, alespoň okrajově, dvě metody.

První je Cfree, kde zjednodušeně funguje algoritmus tak, že nejprve vygeneruje mapu cest a poté hledám cestu. Mapu musím brát spíše jako strom cest.

Druhou, kterou zmíním je Rapidly Exploring Random Trees – RRT. Používá stejné generování mapy jako Cfree, avšak na každé pozici, jež určím, generuje mapu další, která je připojena k té prvotní, která začíná v samotném startu robota. Úplně stejně jako u Cfree, zde generuji strom cest, jak ten prvotní (základní) tak i ostatní, které na něj připojuji viz. obrázek 4.9. [2]



Obrázek 4.9: Příklad stromu u RRT

Jak vidím, metod pro různé plánování je velké množství. Každá má svoje plusy a mínusy. Ale to má ostatně v životě cokoli. Samotné plánování, jak už jsem zmiňoval, souvisí hodně s mými požadavky na robota a samotným prostředím, kde budu plánovat. Lépe řečeno všechno souvisí se vším.

5) Závěr rešeršní části

Pomalu ale jistě jsem na konci prezentace možností, jak vyřešit náš úkol. Teď už mi „jen“ zbývá vybrat si pro mě nejlepší a proveditelné metody. Skloubit je dohromady. K tomu postavit robota tak, aby odpovídal požadavkům jednotlivých metod. Nakonec vytvořit program, který robot pochopí a bude mu rozumět. Jdu se tedy pustit do řešení problémů, které mi nastanou a že jich určitě nebude málo.

6) Praktická část

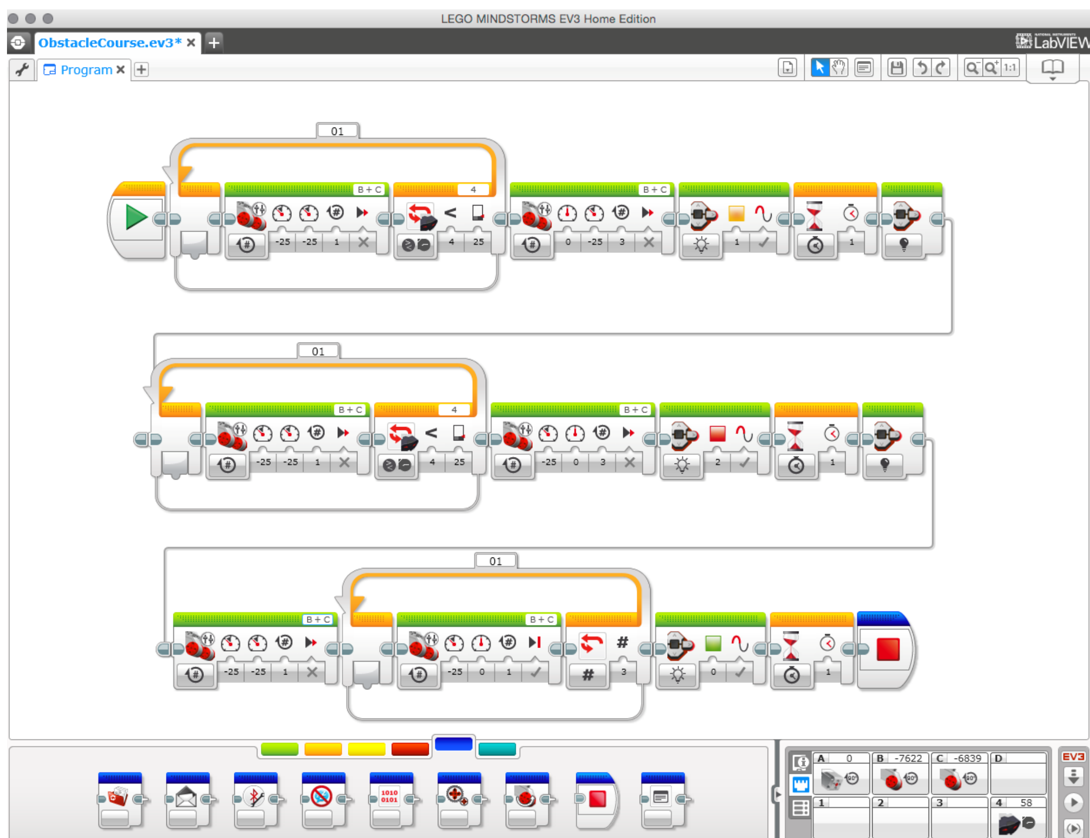
6.1) Výběr lokalizace a mapy pro našeho robota

Zde jen popíši, jak si asi představuji fungování mého robota. Podle prostředí a podle složitosti, se mi zdálo nejlepší vybrat si odometrii. S tím souvisí i samotný výběr mapy a hledání cesty. Mapu použiji klasickou mřížkovou, která pro mé pole bude naprosto stačit. K mřížkové mapě se dobře hodí potenciálové pole, které mi přišlo i myšlenkově blízké.

S tímto výběrem samozřejmě souvisí konstrukce robota a také programovací jazyk. Takže veškerou konstrukci budeme muset podřídit odometrii. Pro jednoduchou demonstraci nám postačí derivačně řízený robot se dvěma poháněnými koly. Ideálně, který se dokáže otočit kolem své osy na místě, jelikož jsem to zkoušel i s konstrukcí, která to nezvládla a nebyl to nejlepší nápad. Výběr a s ním spojené věci jsou nastíněné. Můžu se tedy pustit do samotné konstrukce a programování.

6.2) Problém se softwarem od Lego Mindstorms

Začnu se samotným programováním. Je zde dobré zmínit, že Lego Mindstorms a software určený k programování jsou prvotně dělané pro děti. Ano i v tomto jednoduchém prostředí se dá vytvořit solidní program, ale nepůjde zde vše, co bych si představoval. Jedná se v podstatě o spojování a nastavování různých bloků viz obrázek 6.1.



Obrázek 6.1: Příklad programu vytvořeným v softwaru od Lega

Myslel jsem si, že odometrie by byla skoro nenaprogramovatelná v tomto softwaru. Spletl jsem se. Na internetu jsem nalezl vytvořenou odometrii v tomto softwaru. Popravdě řečeno od začátku jsem počítal v programování v jazyce programátorském, například v pythonu. Takže jsem tuto odometrii ani příliš nerozebíral, ale pro srovnání a pro ukázku, zde přikládám odkaz: https://github.com/ilovenetruinos/ev3g-odometry.

Znovu, ale zdůrazňuji, že přeci jenom zde nejde udělat vše a lépe se pracuje s programovacím jazykem, kde si budete moci udělat vše po svém. Já tedy zapomenu na tento program a budu muset vyřešit, jak udělat to, aby kostka dokázala číst například Python nebo jiný programovací jazyk.

6.3) Kompilace Pythonu do jazyka Lego Mindstorms

V předchozí části jsem poukázal na to, že software od Lego Mindstorms pro mě nebude úplně nejlepší. Má své výhody i nevýhody, ale nevýhody mnohonásobně, pro náš úkol, převyšují klady. Nejlepší by bylo, mít nějaký kompilátor, který by kompiloval z programátorského jazyka např. Pythonu do jazyka, který používá Lego Mindstorms. Byly dvě možnosti, buď takový kompilátor vytvořit nebo už vytvořený najít a použít. Našel jsem tedy už vytvořený kompilátor, lépe řečeno operační systém, který se nahraje do kostky. Tento software mi umožní programovat v mnou vybraném jazyce. Jelikož jsem nenašel žádný český návod na „instalaci" a používaní, tak jsem se rozhodl, že ho zpracuji zde do bakalářské práce. Návod v angličtině a samotné věci potřebné jsou k nalezení na stránkách: http://www.ev3dev.org.

a) Návod na nahrání Debianu Linuxu do kostky Lego Mindstorms

Jak je z nadpisu zřejmé, nejde o nic jiného než od Debian, tedy budeme ho muset nějak nahrát do kostky. Dále budou rozepsané jednotlivé kroky, součásti a programy potřebné k nahrání a zprovoznění Debianu na kostce.

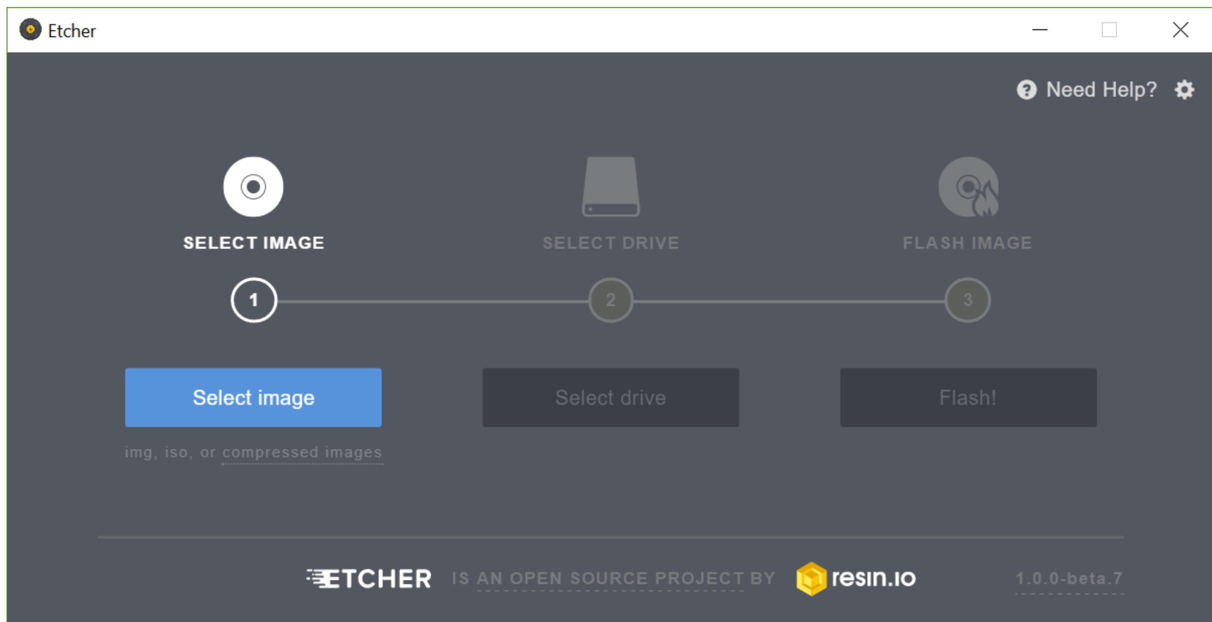
I) Potřebné věci k nahrání Debianu

Za prvé určitě musíte mít samotnou kostku. Za druhé paměťovou kartu typu microSD. Zde si musíte dát pozor na velikost. Měla by vám stačit velikost 2 GB, ovšem může být i větší. Pozor! Větší microSD než 32 GB nejsou kompatibilní s kostkou. Já sám používám 16 GB microSD. Po té samotný počítač a adaptér k paměťové kartě, abyste mohli paměťovou kartu načíst v počítači. Jako poslední součástí je kabel, který slouží k propojení samotné kostky s počítačem.

II) Stažení potřebných souborů

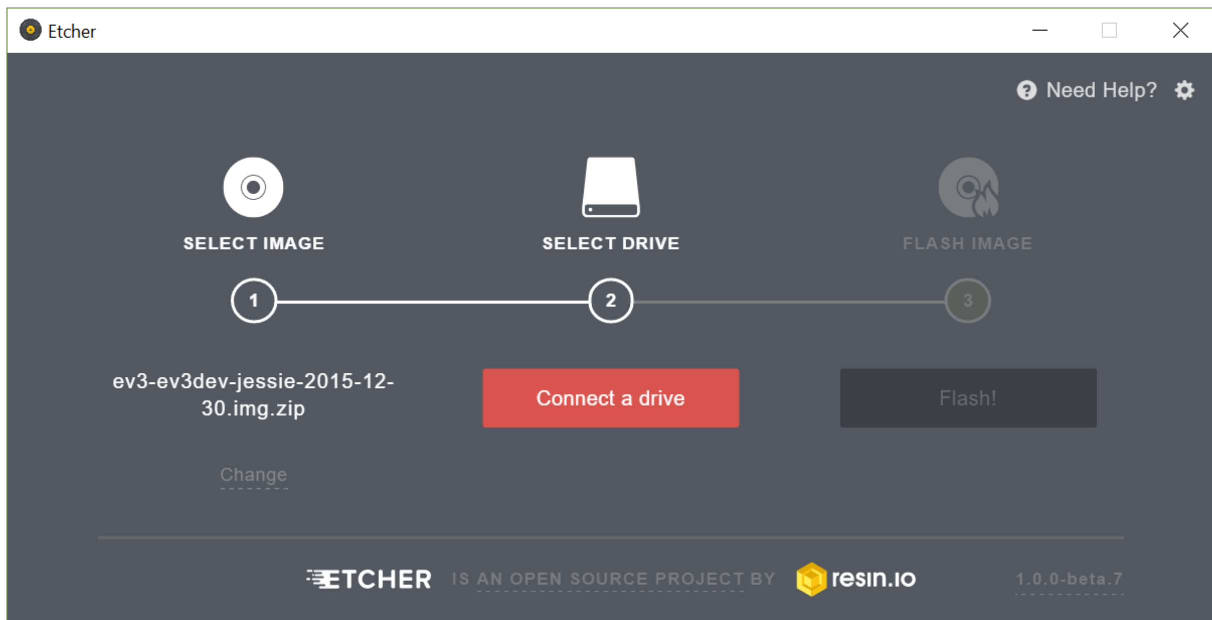
Nejdříve si budete muset stáhnout samotný „image file" Debianu. Naleznete ho na výše uvedených stránkách. Poté musíte rozbalit tento soubor přímo do paměťové karty. To uděláte

pomocí bezplatné verze programu Etcher (https://etcher.io). Tento program nainstalujete a otevřete. Po otevření se vám zobrazí hlavní okno programu.



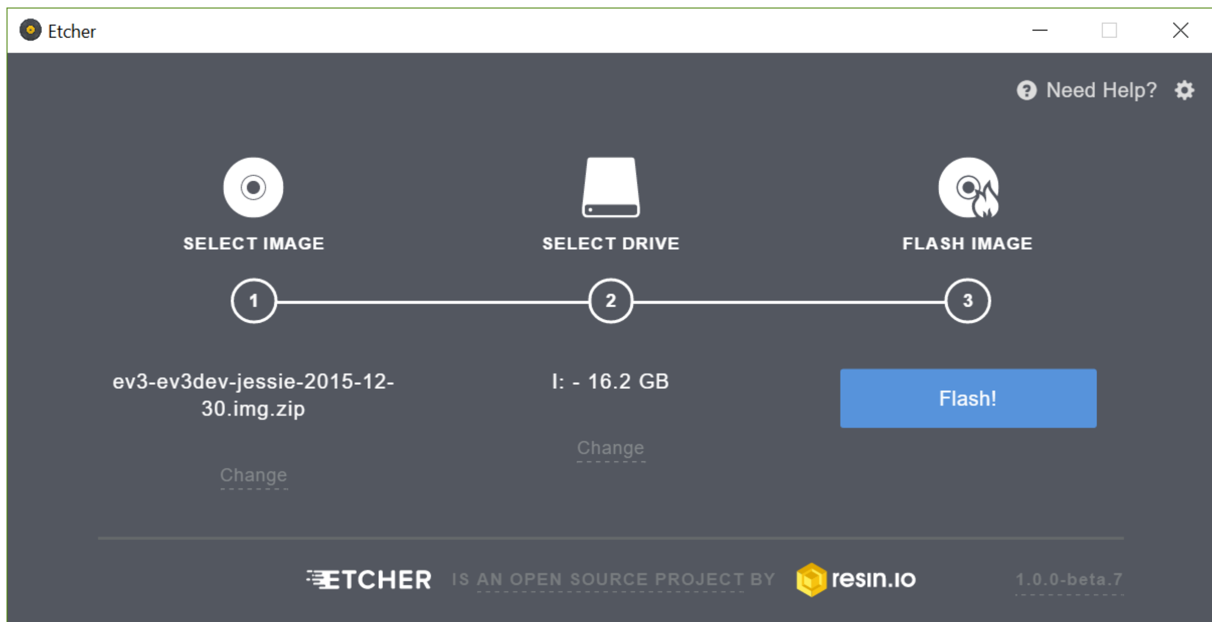
Obrázek 6.2: Hlavní okno programu Etcher

Dále kliknete na „select image“ a vyberete stažený soubor, který chcete nahrát na paměťovou kartu.



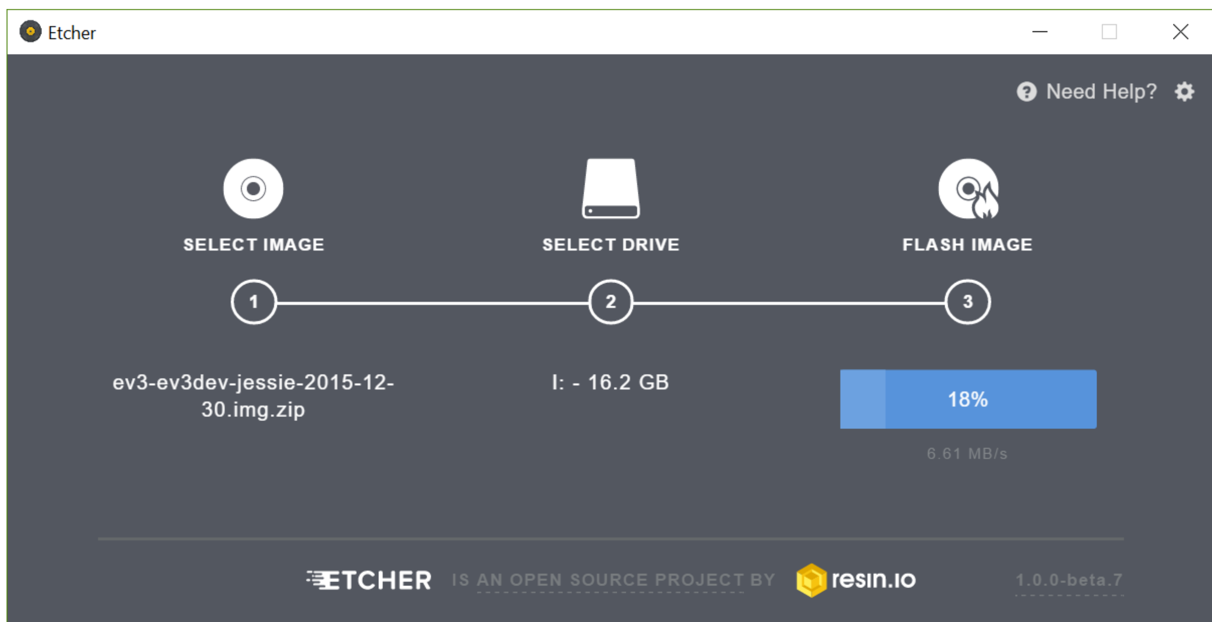
Obrázek 6.3: Vložený "image file" v programu Etcher

Nejdříve připojíte paměťovou kartu a poté kliknete na „connect a drive“ a zvolíte paměťovou kartu na kterou budete nahrávat.



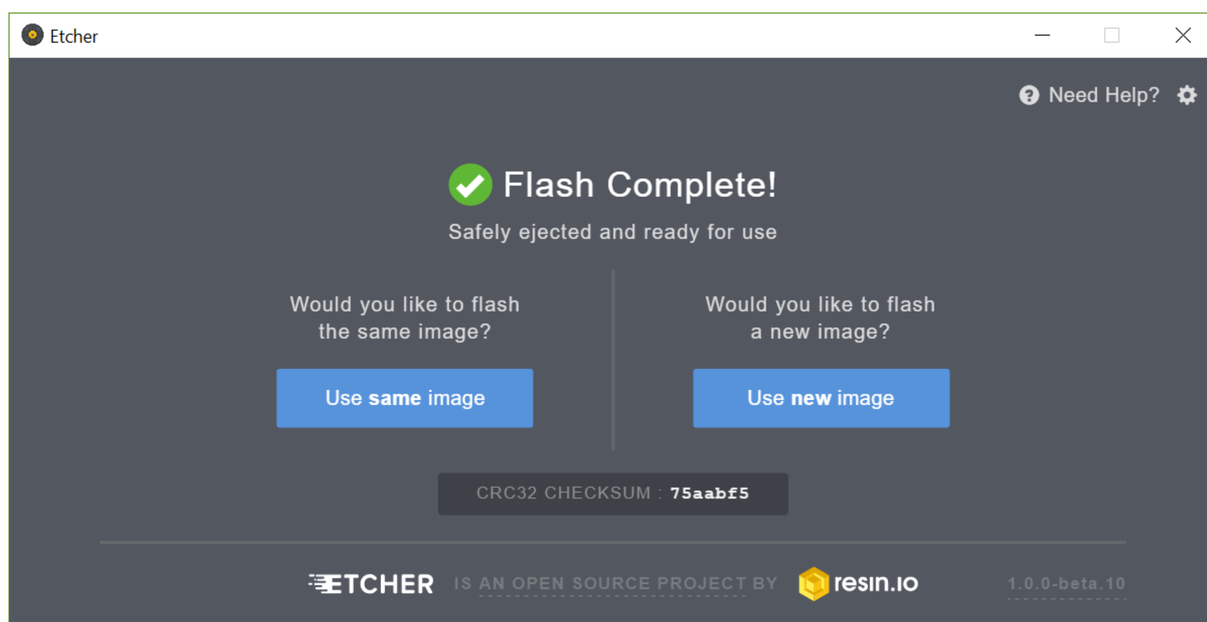
Obrázek 6.4: Okno po výběru paměťové karty v programu Etcher

Nakonec to celé spustíte tlačítkem „flash!“.



Obrázek 6.5: Průběh nahrávání Debianu na paměťovou kartu v programu Etcher

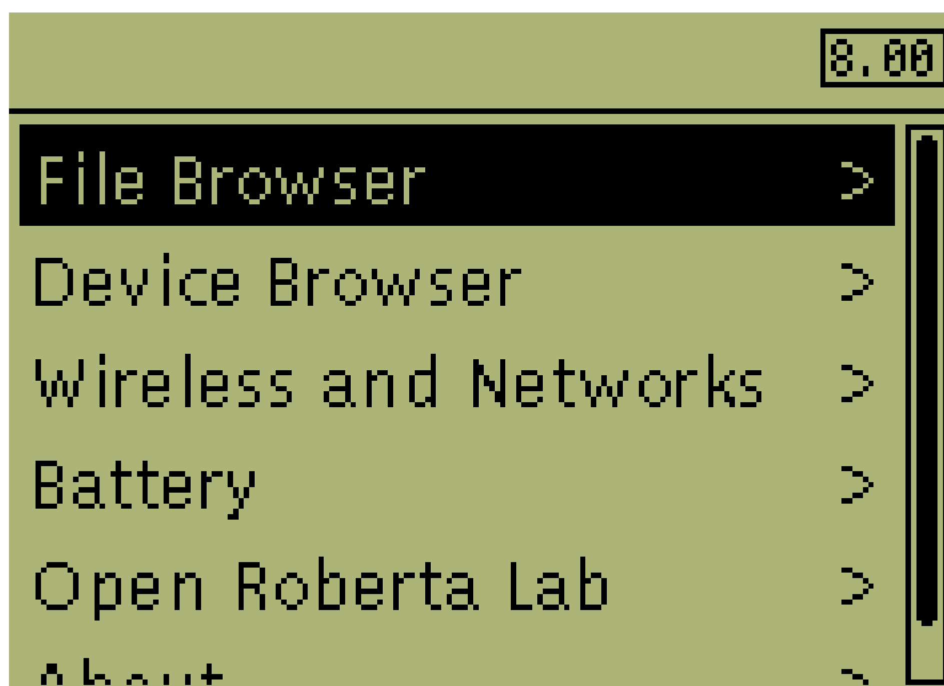
Počkáte, než se vám objeví okno, které je na obrázku 6.6. Tedy až přesun bude hotov.



Obrázek 6.6: Okno, které nám říká, že celé nahrávání je úspěšně dokončeno

Po zobrazení tohoto okna, by instalace měla být úspěšně dokončena, a tedy hotová k použití. Program Etcher tedy můžete zavřít, jelikož už ho dále potřebovat nebudete.

Pokud teď vložíte paměťovou kartu do kostky, tak by se vám při zapínání měla rozsvítit červená LED kontrolka u tlačítek a poté problikávat oranžová LED kontrolka na stejném místě. Obrázek 6.7 nám ukazuje, jak by mělo vypadat hlavní menu v kostce po úspěšném zapnutí operačního systému.

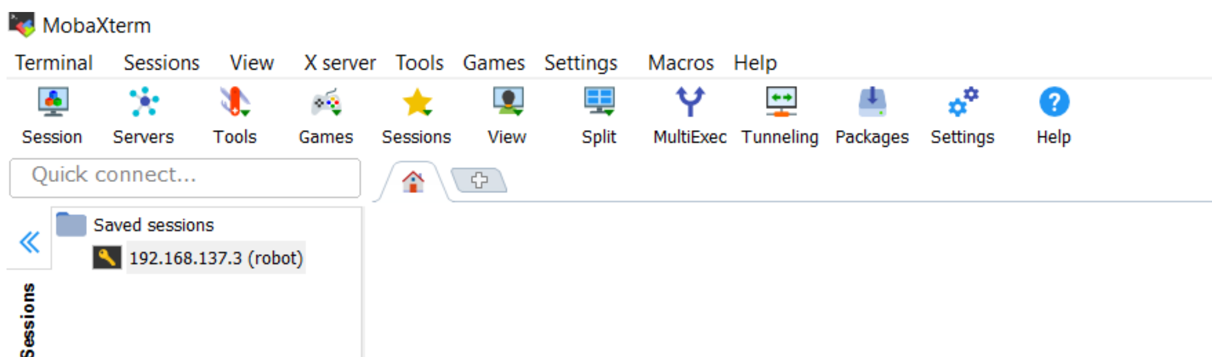


Obrázek 6.7: Hlavní menu po zapnutí kostky s paměťovou kartou

b) Propojení kostky s počítačem

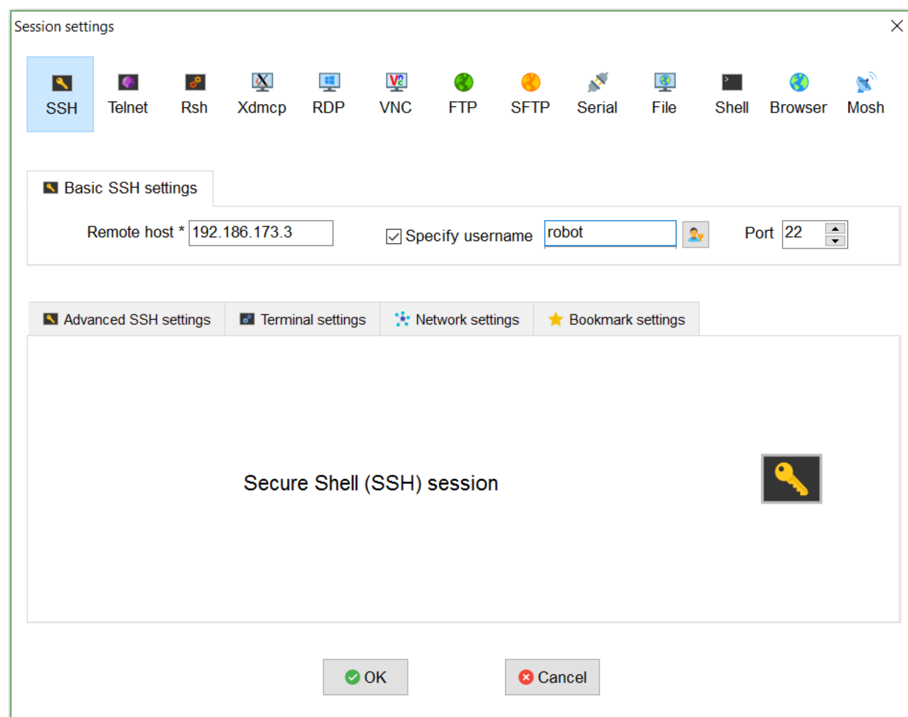
Dalším důležitým krokem je propojení kostky se samotným počítačem, jelikož ji potřebujete ovládat skrze příkazovou řádku v počítači a následně vytvářet samotný program v počítači. Nejjednodušší možností je kabel, který propojí kostku a počítač. Dále je možné propojit kostku pomocí Bluetooth nebo Wi-Fi. Já jsem použil propojení přes kabel, jelikož jiné možnosti sebou přinášejí další úskalí.

Kostku tedy zapnete a propojíte s počítačem pomocí kabelu. Dále budete potřebovat program, který vám udělá SSH (Secure shell) propojení mezi kostkou a počítačem. Takových programů je dost. Osobně jsem použil program MobaXterm, který se bezplatně dá stáhnout na stránkách výrobce (http://mobaxterm.mobatek.net). Na stránkách(http://www.ev3dev.org) v návodu sice používají jednoduší program PuTTY, ale mě více zaujal právě MobaXterm, jelikož do něj lze nahrát samotný PyCharm, ale to už moc předbíhám. Tento program nainstalujete a zapnete.



Obrázek 6.8: Hlavní okno po zapnutí programu MobaXterm (pouze levý horní roh obrazovky)

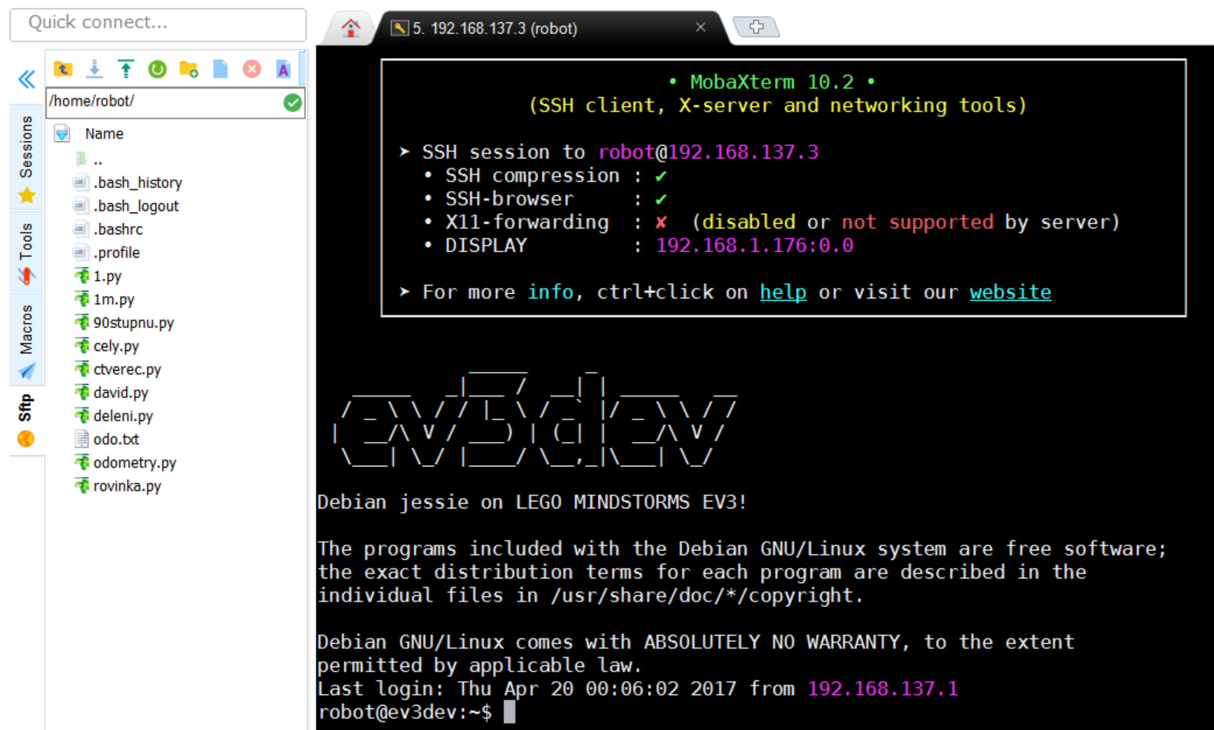
Samotné nastavení je jednoduché. Nejdřív v hlavním okně klinete na „Session".



Obrázek 6.9: Okno, kde využívám SSH k propojení s robotem

Dále se vám zobrazí okno, kde si vyberete, jaký typ propojení chcete. Jak jsem zmiňoval výše, rozkliknete SSH viz. Obrázek 6.9.

Údaje vyplníte podle své kostky. Jako IP adresu použijete tu, která se vám zobrazila po připojen kostky na jejím displeji a jméno je stejné jako na obrázku 6.9, tedy robot. Pokud potvrdíte, tak budete muset vyplnit i heslo do příkazové řádky. Pokud jste ho již dříve nezměnili, mělo by být „maker“. Teď budete mít přístup, jak k souborům v samotné kostce (v levém sloupci), tak také samotnou příkazovou řádku, kterou budete robota ovládat.

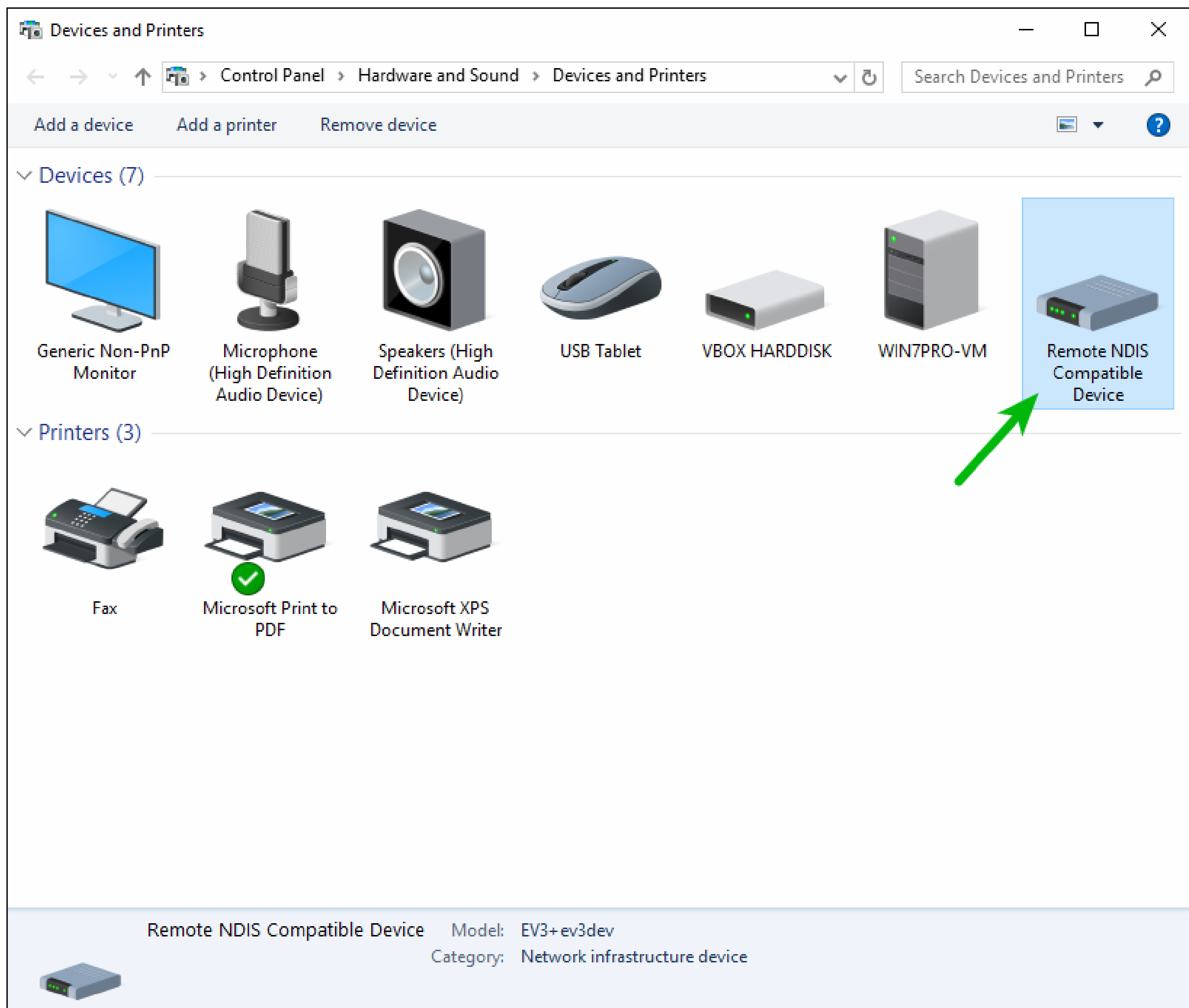


Obrázek 6.10: Příkazový řádek v programu MobaXterm

Zde nastává první úskalí. Po každém připojení a odpojení robota se vám změní i IP adresa. Tu tedy budete muset nastavit pevnou. Jak tedy na to?

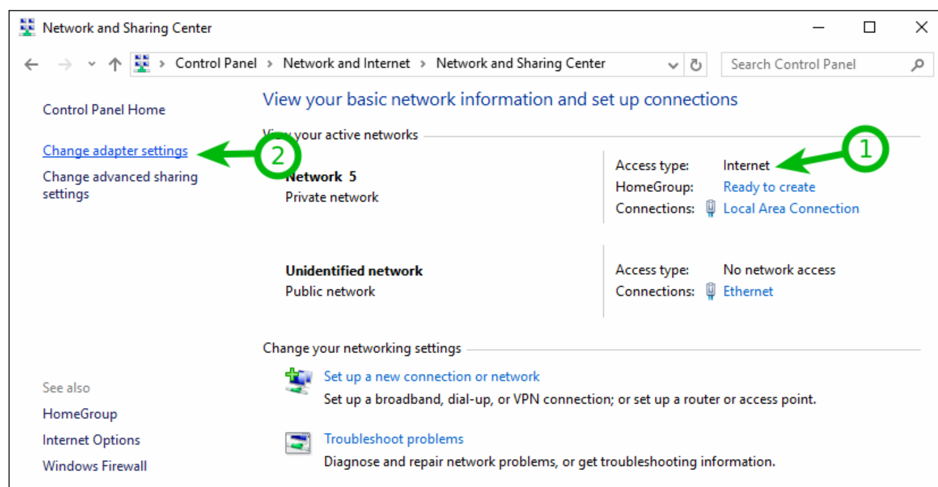
c) Nastavení statické IP adresy

Kostku tedy máte připojenou k počítači. Musíte nejdříve v ovládacích panelech rozkliknout záložku „zařízení a tiskárny“. Tam by se vám mělo objevit zařízení s názvem „Remote NDIS Compatible Device“.



Obrázek 6.11: Zařízení a tiskárny

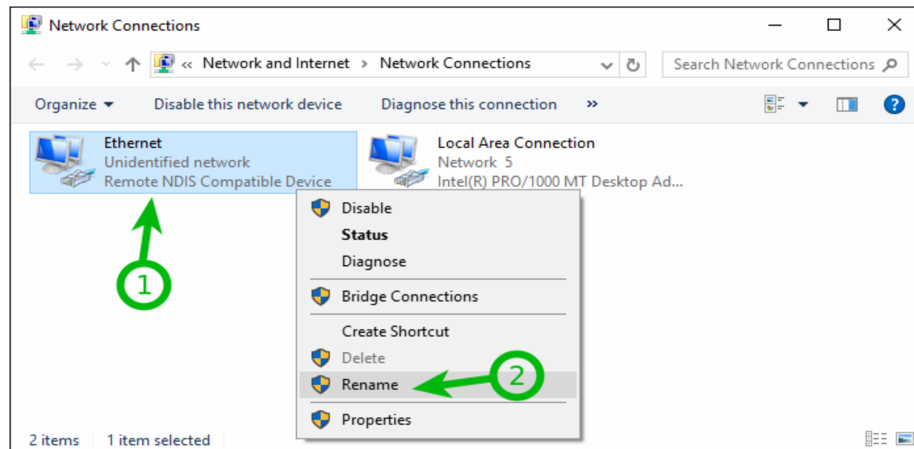
Pravým tlačítkem na něj klinete a vyberete „Nastavení sítě“. Zobrazí se vám okno viz. Obrázek 6.12.



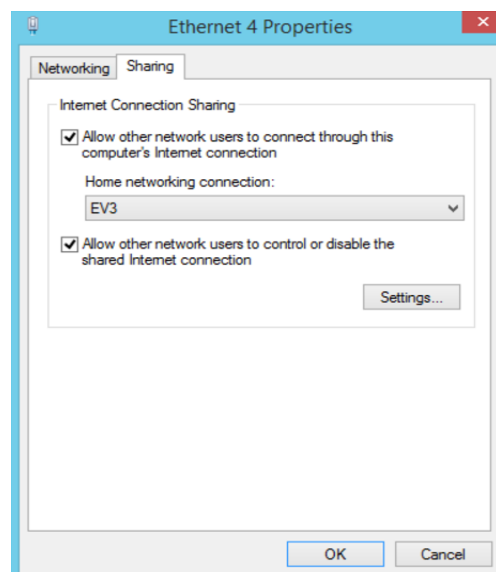
Obrázek 6.12: Nastavení sítě

Zde nejdříve zkontrolujete připojení k internetu a poté kliknete na „Změnit nastavení adaptéru“.

Teď byste si měli přejmenovat spojení robota na např. EV3. U vašeho internetového připojení rozkliknout „vlastnosti“. Kde obě položky musí být zaškrtlé, tedy povolit sdílení viz. Obrázek 6.15.



Obrázek 6.14: Nastavení sdílení internetu



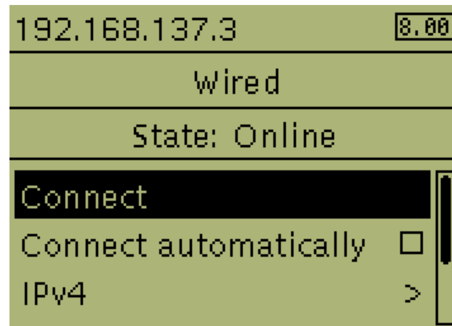
Obrázek 6.13: Změna nastavení adaptéru

Pokud máte více internetových propojení, tak musíte vybrat ten, kterému to dovolujete. Tím by mělo být vše připravené na počítači a teď zbývá nastavit samotnou kostku.

Rozkliknete na kostce „Wireless and Networks“, dále pak „All networks connections“ a měl by se vám tam zobrazit Wired, který rozkliknete.

Obrázek 6.15: All network connections na kostce

Poté zvolíte IPv4 pro nastavení samotné statické IP adresy. Zvolíte „Change“ a „Load windows defaults“. Po chvilce byste měli vidět, že už nejste jen connected, ale i online viz. Obrázek 6.16.

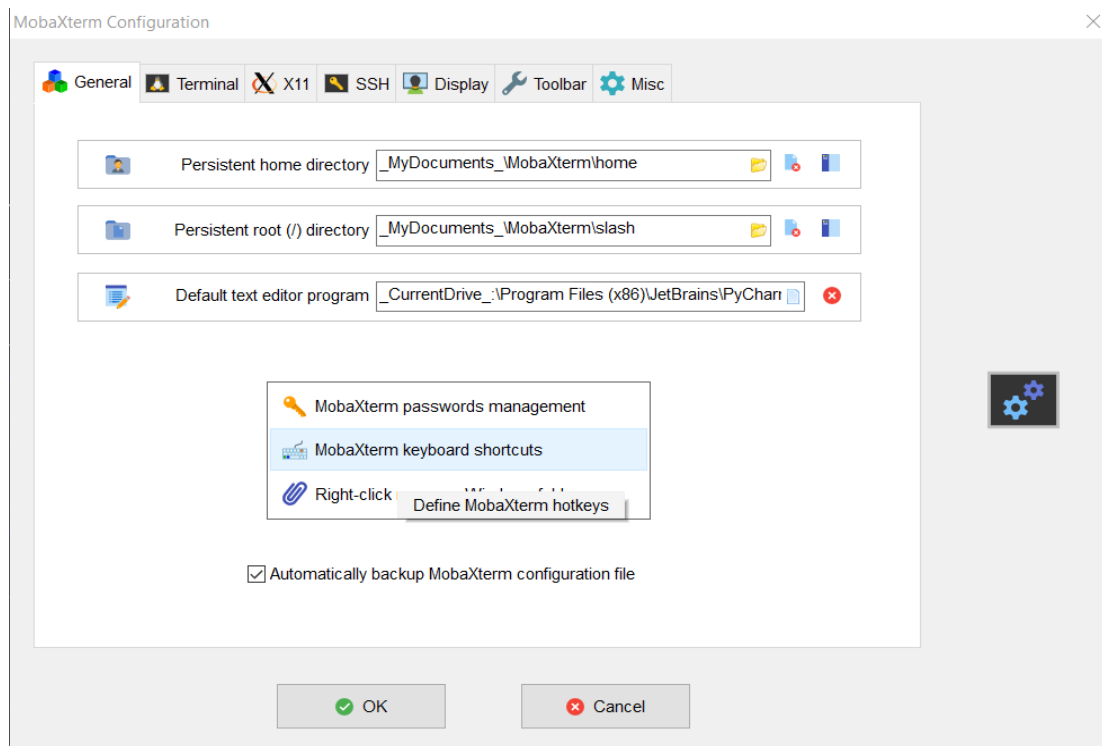


Obrázek 6.16: Obrazovka po nastavení statické IP adresy

Nyní by se vám už neměla adresa libovolně měnit a tím pádem nemusíte pokaždé vytvářet nové SSH spojení.

d) Nastavení výchozího programu pro úpravu skriptů – PyCharm (JetBrains)

Jak jsem již zmiňoval, hodí se mít v MobaXterm nahraný nějaký program pro úpravu skriptů. Ideálně je to právě s PyCharm, který mám osobně odzkoušený. Nainstalujete si tedy PyCharm a poté v MobaXterm v nastavení pouze zvolíte, že chcete jako váš editorovací program používat právě PyCharm viz obrázek 6.17.



Obrázek 6.17: Nastavení PyCharm v MobaXterm

e) Zapínaní samotných skriptů z Lego kostky

Díky této činnosti jsem se setkal s dalším zádrhelem, který jsem řešil velmi dlouho. Takže ho zde ještě uvedu. Pokud budete chtít zapínat skripty přímo z kostky, a ne z příkazového řádku z počítače, tak musíte udělat hned několik zásadních věcí. Do každého skriptu budete muset přidat jako první řádek přesně toto „*#!/usr/bin/env python3*“.

Dále pak musíte dát skriptu povolení, že se může spouštět z kostky. To uděláte tak, že v příkazové řádce zadáte „*chmod +x jmeno_souboru.py*“ a potvrdíte. Zda to proběhlo poznáte tak, že na kostce u jména skriptu je hvězdička, která značí, že se může spouštět z kostky.

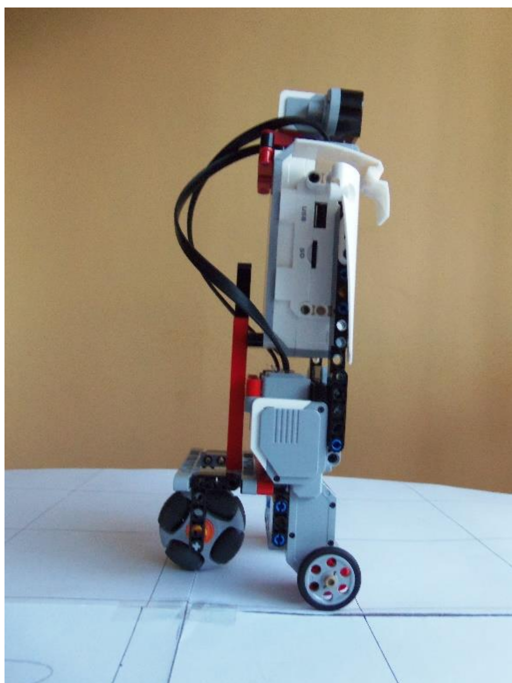
Poslední věcí, která se nikde neuvádí a kterou jsem nakonec vyřešil, souvisí s problémem v oddělování řádků. Jelikož PyCharm odděluje automaticky řádky podle operačního sytému počítače. Tedy pokud je váš operační systém Windows, tak je bude oddělovat jako Windows. S tím má však Linux problém, a tedy v pravém dolním rohu musíte místo CRLF(Windows) nebo CR (Mac OS) změnit na LF. Zdá se to jako maličkost, ale opravdu jsem nad tímto problémem strávil mnoho času.

Popravdě jsem žádný další problém neobjevil. Takže zde je návod na základní instalaci a vyřešení problémů, které mě potkali. Dále už záleží jen na schopnosti programovat. Samotné používání senzorů a jiných součástek je skvěle zpracováno na stránce: https://sites.google.com/site/ev3python.

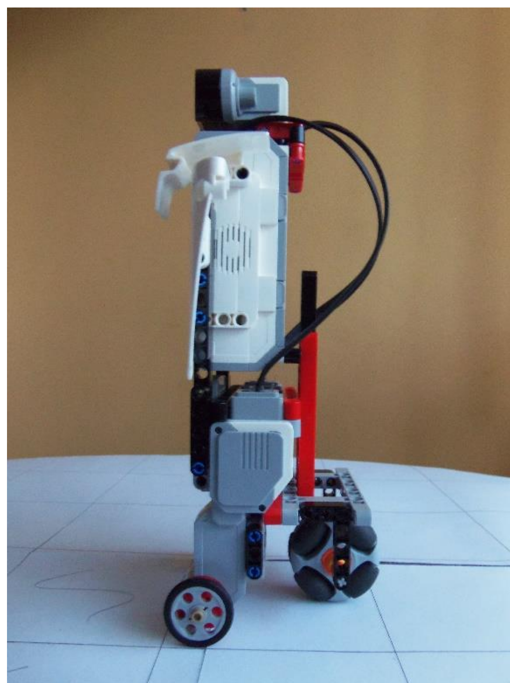
6.4) Samotná konstrukce robota

Zde budu muset brát v potaz odometrii. Tedy, aby se otočil kolem své osy na místě. Myslím si, že zde není asi moc, co psát. Obrázky vypovídají nejlépe o konstrukci robota. Použil jsem pouze kostky ze stavebnice Lego Mindstorms. Největším problémem bylo umístění samotné kostky, která je velká a dost překáží. Na obrázcích níže uvidíte, jak jsem to vyřešil.

Obrázek 6.18: Pohled na konstrukci podvozku

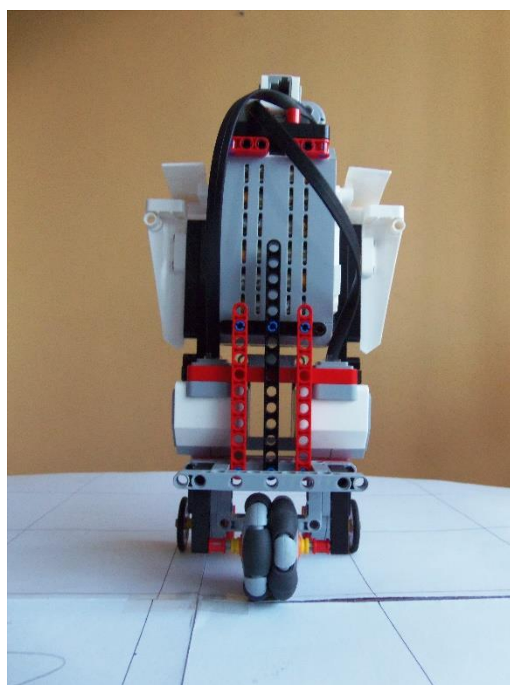


Obrázek 6.22: Pohled na konstrukci pravého boku



Obrázek 6.21: Pohled na konstrukci levého boku

Obrázek 6.20: Pohled na čelní konstrukci robota



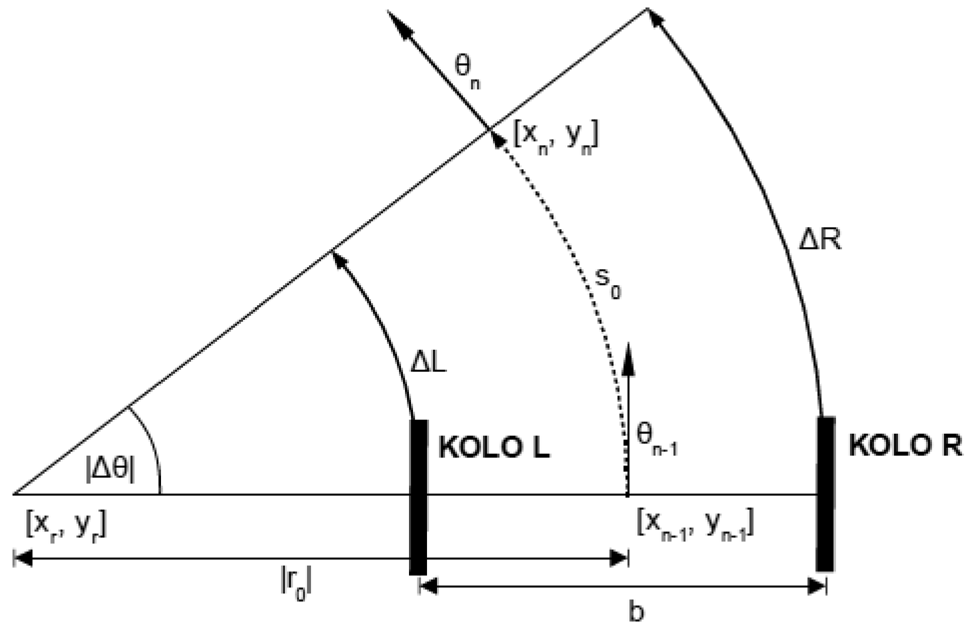
Obrázek 6.19: Pohled na zadní konstrukci robota

Jak na obrázcích vidíte, vyřešil jsem problém tak, že jsem robota natáhnul do výšky, což trochu zhoršuje jeho stabilitu. Na druhou stranu nebyla moc jiná možnost a dělá se to v praxi celkem běžně.

Poslední věcí, o které bych se rád zmínil, je všesměrové kolečko, které používám jako opěrné. Díky tomuto řeším problém, jak se stabilitou, tak také s možností jezdit všemi směry.

6.5) Odometrie

Jak jsem zmiňoval výše, vybral jsem si diferenčně řízeného robota. Konstrukce je k tomuto typu uzpůsobená, a tedy jediný problém je s rovnicemi. Pro ujasnění, jsou všechny důležité hodnoty vidět na obrázku 6.23.



Obrázek 6.23: Diferenčně řízené vozidlo

Jako důležité hodnoty můžeme brát tyto:

1) Rozchod kol b, b = 100 mm
2) Naměřená ujetá vzdálenost kola pravého ΔR
3) Naměřená ujetá vzdálenost kola levého ΔL
4) Počáteční pozice [x_{n-1}; y_{n-1}] a počáteční úhel θ_{n-1}
5) Ujetá vzdálenost s_0, $s_0 = \frac{\Delta R + \Delta L}{2}$
6) Úhel odbočení Δθ, $\Delta\theta = \frac{\Delta R - \Delta L}{b}$
7) Poloměr středního oblouku r_0, $r_0 = \frac{b}{2} \cdot \frac{\Delta R + \Delta L}{\Delta R - \Delta L}$
8) Konečná poloha [x_n; y_n] a orientace θ_n

Tyto základní parametry a rovnice nám postačí k poskládání celkových rovnic pro souřadnici x, y a úhlu θ. [4]

Poloha středu kružnicového oblouku [x_r; y_r]:

$$x_r = x_{n-1} - r_0 \cdot \sin(\theta_{n-1})$$

$$y_r = y_{n-1} + r_0 \cdot \cos(\theta_{n-1})$$

Vzorec pro novou pozici [x_n; y_n] lze odvodit z rotace bodu v rovině:

$$x_n = \cos(\Delta\theta) \cdot (x_{n-1} - x_r) - \sin(\Delta\theta) \cdot (y_{n-1} - y_r) + x_r$$

$$y_n = \sin(\Delta\theta) \cdot (x_{n-1} - x_r) + \cos(\Delta\theta) \cdot (y_{n-1} - y_r) + y_r$$

Tyto rovnice se dají upravit do následující podoby, které doplníme rovnicí pro natočení robota:

$$x_n = x_{n-1} + r_0(\cos(\Delta\theta) \cdot \sin(\theta_{n-1}) + \sin(\Delta\theta) \cdot \cos(\theta_{n-1}) - \sin(\theta_{n-1}))$$

$$y_n = y_{n-1} + r_0(\sin(\Delta\theta) \cdot \sin(\theta_{n-1}) - \cos(\Delta\theta) \cdot \cos(\theta_{n-1}) + \cos(\theta_{n-1}))$$

$$\theta_n = \theta_{n-1} + \Delta\theta$$

Tyto rovnice se dají zjednodušit na daleko lépe programovatelné rovnice, které nemají takový nárok na výpočetní techniku. V podstatě se kružnici aproximujeme mnohoúhelníkem. Pokud budeme mít dostatečnou frekvenci iterací, tak to bude téměř přesné a tato chyba aproximací bude řádově nižší než chyby jiné. [4]

Aproximované řešení:

$$x_n = x_{n-1} + s_0 \cdot \cos\left(\theta_{n-1} + \frac{\Delta\theta}{2}\right)$$

$$y_n = y_{n-1} + s_0 \cdot \sin\left(\theta_{n-1} + \frac{\Delta\theta}{2}\right)$$

$$\theta_n = \theta_{n-1} + \Delta\theta$$

Právě toto aproximované řešení jsem použil pro programovaní robota. Musím ještě dodat, jak jsem počítal ΔR a ΔL. Tyto hodnoty jsem počítal tak, že jsem si změřil poloměr kola, který jsem označil r. Dále jsem odečítal z motoru hodnotu pulsů neboli počet stupňů, o který se pootočil. Označil jsem tyto hodnoty jako motorR a motorL.

$$\Delta R = m \cdot r \cdot 2 \cdot \pi$$

$$\Delta L = n \cdot r \cdot 2 \cdot \pi$$

$$m = \frac{motorR}{360}$$

$$n = \frac{motorL}{360}$$

Naznačím část skriptu, jen s rovnicemi a s vypsáním hodnot ujeté vzdálenosti, které se mi ukládaly do textového souboru, kvůli lepšímu srovnání hodnot. V tomto skriptu měl robot jet pouze před sebe a čtyřikrát to v cyklu zopakovat. Po každém popojetí jsem srovnával hodnoty.

Část skriptu tedy vypadala takto:

```
mA = LargeMotor('outA')
mD = LargeMotor('outD')

i = 0

while i < 4:
    mD.run_to_rel_pos(position_sp=360, speed_sp=500, stop_action="hold")
    mA.run_to_rel_pos(position_sp=360, speed_sp=500, stop_action="hold")

    mD.wait_while('running')
    mA.wait_while('running')


    motorR = mA.position
    motorL = mD.position

    x = (motorR/360)
    y = (motorL/360)

    deltaR = x * r * 2 * pi
    deltaL = y * r * 2 * pi

    s = (deltaR+deltaL)/2
    deltaTheta = (deltaR-deltaL)/b

    xn = xn + (s * cos( theta + (deltaTheta / 2 )))
    yn = yn + (s * sin( theta + (deltaTheta / 2 )))
    theta = theta + deltaTheta

    with open( "odo.txt", "a" ) as file:
        file.write( "\n Cyklus " )
    with open( "odo.txt", "a" ) as file:
        file.write(str( i ))
    with open( "odo.txt", "a" ) as file:
        file.write(str( "\n\n" ))

    with open( "odo.txt", "a" ) as file:
        file.write(str( a ))
    with open( "odo.txt", "a" ) as file:
        file.write( "    -    Pulsy A\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( d ))
    with open( "odo.txt", "a" ) as file:
        file.write( "    -    Pulsy D\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( deltaR ))
    with open( "odo.txt", "a" ) as file:
        file.write( "    -    deltaR\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( deltaL ))
    with open( "odo.txt", "a" ) as file:
        file.write( "    -    deltaL\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( xn ))
    with open( "odo.txt", "a" ) as file:
```

```
        file.write( "   -   xn\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( yn ))
    with open( "odo.txt", "a" ) as file:
        file.write( "   -   yn\n")

    with open( "odo.txt", "a" ) as file:
        file.write(str( theta ))
    with open( "odo.txt", "a" ) as file:
        file.write( "   -   theta\n")


    i += 1
```

Výsledkem byl tedy soubor hodnot, které jsem vyhodnocoval. Bohužel jsem došel k závěru, že odečítání kol ujeté vzdálenosti se velmi liší a má velkou chybu. Snažil jsem se tuto chybu nějakým způsobem korigovat. Ovšem chyba byla v podstatě náhodná a dosti velká. Na jednu otáčku 2-6 stupňů, což mi v konečném důsledku dávalo to, že robot, pokud jel pouze rovně, tak si myslel, že trochu zatáčí. Zkoušel jsem s tím pracovat dál, ale opravdu to nešlo. Vyplývá z toho, že motory, které jsou u stavebnice nejsou spolehlivé a příliš se nehodí pro odometrii. Musel jsem se tedy smířit s tím, že robot nebude přesně vědět, kde je, ale bude si to pouze myslet, podle toho, v jaké části skriptu se nachází. Myslím, že i přes tuto vadu může být robot úspěšný a projít cestu správně.

6.6) Vytvoření mapy

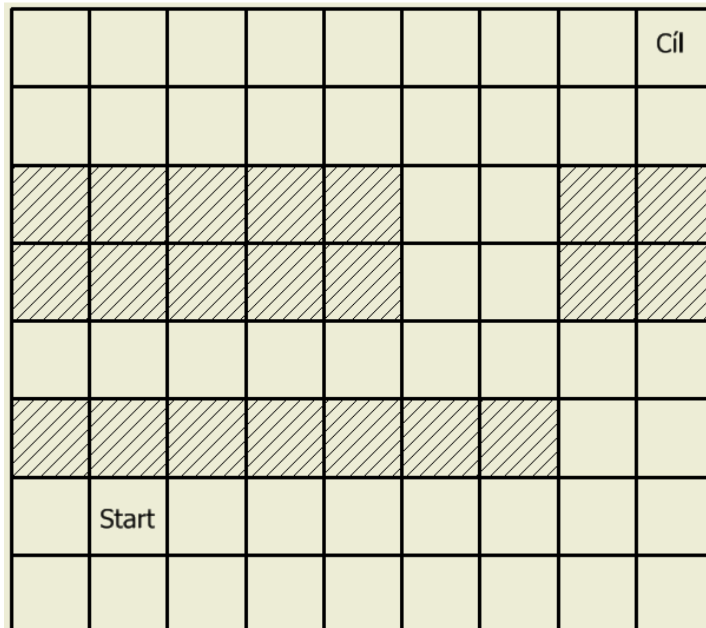
Vytvoření mapy pro mě byl jeden z nejlehčích úkolů. V podstatě mapu s překážkami jsem vložil na mřížkový rastr. Čím menší rastr bude, jinak řečeno, čím menší budou čtverce a čím jich bude více, tím bude mapa přesnější. Pro základní funkci mi však stačily čtverce velké 100 x 100 mm. Na mapě jsem vyznačil start a cíl a samozřejmě překážky, které jsou vybarvené černě. Umístění překážek a samotná velikost mapy je zajisté pouze na mém subjektivním určení. Mapa se v programu robota dá změnit během pár minut.

Opět pro představu vkládám jednu z pracovních map, jak asi takové určení vypadá:

```
x = 8 # sirka mapy
y = 9 # vyska mapy

# Vytvorime dvourozmernou matici naplnenou nulami
M = [ [ 0 for i in range( y ) ] for j in range( x ) ]

# Oznacime si policka, kam robot nesmi vstoupit

M[ 0 ][ 2 ] = -1
M[ 1 ][ 2 ] = -1
M[ 2 ][ 2 ] = -1
M[ 3 ][ 2 ] = -1
M[ 4 ][ 2 ] = -1
M[ 5 ][ 2 ] = -1
M[ 0 ][ 4 ] = -1
M[ 1 ][ 4 ] = -1
M[ 2 ][ 4 ] = -1
M[ 3 ][ 4 ] = -1
M[ 0 ][ 5 ] = -1
```

```
M[ 1 ][ 5 ] = -1
M[ 2 ][ 5 ] = -1
M[ 3 ][ 5 ] = -1
M[ 6 ][ 4 ] = -1
M[ 7 ][ 4 ] = -1
M[ 6 ][ 5 ] = -1
M[ 7 ][ 5 ] = -1

# Inicializujeme pocatecni pozici robota
initial_position = [ 1, 1, 0 ]

# Ulozime si koncovou pozici.
final_position = [ 7, 8 ]
```

Je vidět, že zadávám pouze počáteční pozici, konečnou pozici a místa překážek, které označujeme -1. Pro znázornění přikládám i jak by mapa asi vypadala.



Obrázek 6.24: Návrh pracovní mapy

6.7) Pohyb robota po mapě – Potenciálová pole

Poslední věcí, kterou bylo za úkol vyřešit, byl samotný pohyb robota po mapě. Museli jsme se obejít bez odometrie. Robot si tedy pouze myslí, že je na určité pozici, ale to ještě neznamená, že tam je.

Hlavní myšlenka se odvíjí od toho, že si v podstatě robot sám očísluje mapu, jako potenciálové pole, aniž by po mapě chodil. Algoritmus funguje tak, že na počáteční pozici si označí sousedy čísly, kam by mohl jet. Pozice, které znázorňují překážky už mají hodnotu svou a to -1. Zavedl jsem si nějakou hodnotu a nazýval ji čas. V předchozí kapitole o mapě můžete vidět počáteční pozici, která se neskládá jen ze souřadnic x, y(pozice), ale je tam i třetí hodnota. Tou je právě můj zavedený čas. Počáteční pozice tedy bude mít čas nula a všechna sousedící pole s počátkem hodnotu 1. Robot bude pokračovat na další pole a opět na ty pole, která ještě nemají přiřazené číslo a sousedí se současným polem, dostanou hodnotu 2. Takto to bude pokračovat až do konečné pozice, tedy do cíle. Poté si robot projde cestu nazpět a ukládá si pozice, kterými musí

projít. Nakonec celou trasu otočí a vydá se na samotnou cestu. Bohužel i tento algoritmus není dokonalý. Nalezená cesta nemusí být nejkratší, a navíc se nám robot může chytit do pasti. Řekněme, ale že na délce nám v tuto chvíli moc nezáleží a naším úkolem není robota zmást.

Dále pak algoritmus navrhne cestu a řekne robotovi, kudy se má vydat. Zde už začíná část s motory. Zde jsem nejdříve robota naprogramoval tak, aby se točil pouze po pravé ruce, což je možná nelogické, ale hlavně mě zajímalo, zda tento algoritmus bude fungovat, ovšem chyba byla velká. Musel jsem tedy robotovi říct, kdy se má otáčet doprava, kdy doleva, aby vždy ujel nejkratší cestu, a tedy snížil chybu způsobenou pohybem.

Celý program je proložen komentáři k jednotlivým krokům, tudíž ho zde neuvádím a nepopisuji. Uvedu zde pouze pár poznatků, na které jsem narazil při práci s motory. Motory jsou vždy přesnější při pomalejších rychlostech, což je logické. Důležité je, aby se motory skutečně otočili a provedli celý pohyb. Proto je dobré požívat mezery mezi pohyby pomocí funkce sleep(), která pozastaví na chvíli samotný program. Pro přesnější pohyb jsou lepší kola menších průměrů. Také je dobré, když mají kola jasně definovaný bod, po kterém se pohybují, tudíž není vhodné používat kola, která jsou široká.

Vzhledem k tomu, že chytrá kostka má i reproduktor, vložil jsem do programu příkaz, který umožní robotovi zahlásit hlasovým projevem nalezenou cestu. Tedy pokud robot cestu našel. V opačném případě, na který je velmi dobré myslet, zahlásí, že cestu nenašel a ukončí program.

7. Závěr

Cílem práce bylo naprogramovat robota, který díky zadané mapě dokáže dojet ze startu do cíle a vyhýbat se překážkám, které jsou vyznačené v mapě.

V rešeršní části jsem nastínil možné druhy lokalizace robota, různé mapy a druhy algoritmů hledání cesty. Z nastíněných variant jsem si pro definici prostředí vybral senzorickou mapu a variantu metody potenciálových polí pro hledání cesty.

Robot je založen na stavebnici Lego Mindstorms s upraveným firmwarem (dodávaný software byl nahrazen operačním systémem Linux). Robot byl naprogramován v jazyce Python. Stroj využívá připravenou mapu prostředí, kde jsou vyznačené překážky. V zadané mapě dokáže sám nalézt cestu do cíle. Robot dokáže identifikovat případy, kdy není možné cestu nalézt, v takovém případě to oznámí. Zařízení je konstruováno tak, aby se dokázalo otočit na místě, což usnadňuje pohyb po ploše.

Další vývoj může směřovat k lepší identifikaci pozice, protože nyní robot svoji polohu zná pouze přibližně, na základě počtu vykonaných příkazů. Dále je možné pokračovat ve vylepšování algoritmu, například implementací nejkratší nebo nejbezpečnější cesty.

Tuto práci jsem chtěl napsat také jako návod a inspiraci dalším studentům. Všechny možné problémy, které mě potkaly, jsem se snažil obsáhnout, popsat a umožnit tím ostatním lépe a snadněji se v této problematice pohybovat.

8. Seznam použité literatury

1. CONRADT, J. A Distributed Cognitive Map for Spatial Navigation Based on Graphically Organized Place Agents. URL <http://www.ini.uzh.ch/~conradt/projects/CogRobNavigation/>.
2. LAVALLE, S. Rapidly-exploring random trees A new tool for path planning. URL <http://msl.cs.uiuc.edu/~lavalle/papers/Lav98c.pdf>.
3. MILFORD, M.; SCHULZ, R. Principles of goal-directed spatial robot navigation in biomimetic models. *The Royal Society*. 2014-09-29. URL <http://rstb.royalsocietypublishing.org/content/369/1655/20130484>.
4. SKALKA, M. *Srovnání lokalizačních technik*. Praha : Univerzita Karlova v Praze Matematicko-fyzikální fakulta, 2011. 86 p. URL <http://marek.sk.sweb.cz/lokalizace/DP-Srovnani-lokalizacnich-technik-Skalka.pdf>.
5. SYKALA, V. *Vizualizace hledání cesty pro robota*. Brno : Vysoké učení technické v Brně, 2007. 25 p. URL <https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=116127>.
6. TAYLOR, G. Path planning. *United states naval academy*. URL <https://www.usna.edu/Users/cs/taylor/courses/si475/class/pathPlanning.pdf>.
7. WINKLER, Z. Plánování na mřízce. *Robotika*. 2003-03-12. URL <https://robotika.cz/guide/gridplan/cs>.
8. A*. *Wikipedie*. URL <https://cs.wikipedia.org/wiki/A*>.
9. Connecting to the Internet via USB. URL <http://www.ev3dev.org/docs/tutorials/connecting-to-the-internet-via-usb/>.
10. Getting Started with ev3dev. URL <http://www.ev3dev.org/docs/getting-started/>.
11. LEARNING TO CODE WITH ROBOTS. URL <http://blog.grunick.com/tag/mindstorm/>.

9. Seznam obrázků

Obrázek 2.1: Ilustrace chyby, která roste s ujetou vzdáleností a která je způsobena drobnými rozdíly mezi nominálním a skutečným obvodem levého a pravého kola. Vzorky označují potenciální polohu robota při jízdě po (dle měření odometrie) přímé dráze, barevně jsou odlišeny jednotlivé kroky 0 až 4 [4] 10
Obrázek 2.2: Diferenčně řízené vozidlo [4] 10
Obrázek 2.3: Vozidlo s pojezdem typu tříkolka [4] 11
Obrázek 2.4: Ackermanovo řízení [4] 11
Obrázek 2.5: Nejjednoduší konfigurace pojezdu všesměrového robota se třemi koly [4] 12
Obrázek 2.6: Aktivní majáky [4] 13
Obrázek 3.1: Převod mapy do topologické místnosti [1] 15
Obrázek 4.1: Ukázka potenciálového pole [3] 16
Obrázek 4.2: Tzv. metoda záplavového vyplňování (wavefront-expansion) [7] 17
Obrázek 4.3: Konec Dijkstrova algoritmu. Všechny cesty mezi uzly jsou spočítané a nejkratší [8] 18
Obrázek 4.4: Prostor rozdělený lichoběžníkovou dekompozicí a výsledný graf souslednosti [5] 19
Obrázek 4.5: Graf viditelnosti [5] 20
Obrázek 4.6: Redukovaný graf viditelnosti o nepotřebné hrany [5] 20
Obrázek 4.7: Tvorba Voroného diagramu [6] 21
Obrázek 4.8: Voroného diagramy použité v místnosti [6] 21
Obrázek 4.9: Příklad stromu u RRT [2] 22
Obrázek 6.1: Příklad programu vytvořeným v softwaru od Lega [11] 23
Obrázek 6.2: Hlavní okno programu Etcher [10] 25
Obrázek 6.3: Vložený "image file" v programu Etcher [10] 25
Obrázek 6.4: Okno po výběru paměťové karty v programu Etcher [10] 26
Obrázek 6.5: Průběh nahrávání Debianu na paměťovou kartu v programu Etcher [10] 26
Obrázek 6.6: Okno, které nám říká, že celé nahrávání je úspěšně dokončeno [10] 27
Obrázek 6.7: Hlavní menu po zapnutí kostky s paměťovou kartou [10] 27
Obrázek 6.8: Hlavní okno po zapnutí programu MobaXterm (pouze levý horní roh obrazovky) 28
Obrázek 6.9: Okno, kde využívám SSH k propojení s robotem 28
Obrázek 6.10: Příkazový řádek v programu MobaXterm 29
Obrázek 6.11: Zařízení a tiskárny [9] 30
Obrázek 6.12: Nastavení sítě [9] 30
Obrázek 6.13: Změna nastavení adaptéru [9] 31
Obrázek 6.14: Nastavení sdílení internetu [9] 31
Obrázek 6.15: All network connections na kostce [9] 31
Obrázek 6.16: Obrazovka po nastavení statické IP adresy [9] 32
Obrázek 6.17: Nastavení PyCharm v MobaXterm 32
Obrázek 6.18: Pohled na konstrukci podvozku 33
Obrázek 6.19: Pohled na zadní konstrukci robota 34
Obrázek 6.20: Pohled na čelní konstrukci robota 34
Obrázek 6.21: Pohled na konstrukci levého boku 34
Obrázek 6.22: Pohled na konstrukci pravého boku 34
Obrázek 6.23: Diferenčně řízené vozidlo [4] 35
Obrázek 6.24: Návrh pracovní mapy 39