VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

URČOVÁNÍ POLOHY ROBOTA NA ZÁKLADĚ MĚŘENÍ ZE SENZORŮ

ROBOT POSITIONING BASED ON SENSOR MEASUREMENTS

BAKALÁŘSKÁ PRÁCE
BACHELOR‘S THESIS

AUTOR PRÁCE
AUTHOR

Ondrej Čakloš

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Jaroslav Rozman, Ph.D.

BRNO 2017

Abstrakt

Cieľom tejto práce je vytvoriť program, ktorý bude prijímať merania zo senzorov robota a previesť ich fúziu. Následne použiť fúziu vybraných senzorov na vyhodnotenie polohy daného robota.

Ako riešenie týchto problémov som použil znalosti Kalmanovho filtra, resp. jeho rozšírenej podoby. Kalmanov filter dokáže pri správne formulovaných správach o meraniach previesť fúziu týchto meraní a zároveň jeho výsledkom je žiadaná poloha robota. Filter dokáže prijímať merania z viacerých zdrojov a to aj duplicitných.

Výsledné hodnoty stavov sa preukázali ako dostatočne presné pre úspešné lokalizovanie robota v priestore.

Abstract

The goal of this thesis is to create a program, that will be receiving measurements from robot's sensors and fuse them together. Afterwards use this data fusion of chosen sensors to estimate location of a robot.

As a solution for these problems I've used my knowledge of Kalman filters, especially extended one. If messages from sensor measurements are well formulated, Kalman filter can perform fusion of measurements together with estimating the actual position of a robot. Filter can receive measurements from multiple sources and even from duplicities.

The estimated states have proven themselves reasonably accurate for successful robot localization in space.

Klíčová slova

Fúzia dát, Global Positioning System, Kalmanov filter, Rozšírený Kalmanov filter, Lokalizácia robota, Navigačný zásobník, Odometria, Pravdepodobnostná robotika, ROS

Keywords

Data fusion, Global Positioning System, Kalman filter , Extended Kalman filter, Robot localization, Navigation stack, Odometry, Probabilistic robotics, ROS

Citace

Čakloš Ondrej: Určování polohy robota na základě měření ze senzorů, bakalářská práce, Brno, FIT VUT v Brně, 2017

Určovanie polohy robota na základe meraní zo senzorov

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Ing. Jaroslava Rozmana, Ph.D. a s použitím uvedenej literatúry a prameňov.

……………………
Ondrej Čakloš
Datum (17.5.2017)

Poděkování

Rád by som poďakoval svojmu vedúcemu bakalárskej práce Ing. Jaroslavovi Rozmanovi, Ph.D. za odborné vedenie, za pomoc a rady pri spracovaní tejto práce.

© Ondrej Čakloš, 2017
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Obsah 1
1 Úvod 3
1.1 Neistota v robotike 3
1.2 Pravdepodobnostná robotika 4
2 Teoretická časť 6
2.1 Merania zo senzorov 6
2.2 Transformácia robota 6
2.2.1 Účel transformácií 6
2.3 Odometria 7
2.4 Fúzia senzorov 7
2.4.1 Kalmanov filter 7
2.4.2 Rovnice a vysvetlenie 9
3 Navigačný zásobník 11
3.1 Hardwarové požiadavky 11
3.2 Nastavenie robota 11
3.2.1 ROS 11
3.2.2 Konfigurácia transformácie 11
3.2.3 Vysielanie senzorových prúdov cez ROS 12
3.2.4 Odometrické informácie 12
3.2.5 Ovládač základne 12
3.3 Nastavenie navigačného zásobníka 12
3.3.1 Vytvorenie packagu 12
3.3.2 Vytvorenie spúšťacieho konfiguračného súboru pre robota 12
3.3.3 Konfigurácia máp s oceneniami 13
3.3.4 Konfigurácia základného lokálneho plánovača 13
3.3.5 Vytvorenie spúšťacieho súboru pre navigačný zásobník 13
4 Návrh implementácie 14
4.1 Prehľad 14
4.1 Vytvorenie modelu robota 14
4.1.1 Základné nastavenie 14
4.1.2 Vytvorenie modelu a transformácií 15
4.2 Práca s ROS-om 15
4.3 Kalmanov filter 15
4.3.1 Vytvorenie základu 15

4.3.2 Generalizácia 16
4.4 Presnosť lokalizácie..16
5 Závěr 20

1 Úvod

Pri určovaní polohy robota v priestore sa potrebujeme zoznámiť so senzormi používaných v robotike. Zistíme, že používané senzory sa menia od robota k robotovi. Od jednoduchých enkodérov či sonarov až po laserové merače vzdialenosti skoro v 360° okolí zvaný Lidar či rôzne verzie kinectu ktoré vytvárajú mračno bodov, kde poloha každého bodu je určená v troch súradných osách. Zistili sme že medzi najpoužívanejšie patria enkodéry, ktoré poskytujú odometriu, IMU, poskytujúce hneď niekoľko senzorov pohromade, globálny pozičný systém GPS a už spomínaný lidar alebo kinect.

Ďalej sa budeme zaoberať meraniami zo senzorov a hlavne fúziou týchto dát. K tomu budeme potrebovať nejaký spôsob fúzie, nejaký filter, ktorý si poradí s viacerými senzormi a nelineárnym pohybom robota. Ak fúzie prebehne v poriadku využijeme ju na určenie polohy robota.

Problém pri určovaní polohy robota v prostredí je v nepresnosti nameraných hodnôt zo senzorov. Z dôvodu týchto nepresností a predpokladu, že lokalizačný systém s týmito nepresnosťami počíta, môže sa niekedy javiť ako by sa robot nachádzal na viacerých miestach súčasne. Preto musíme zabezpečiť čo najväčšiu presnosť lokalizácie.

1.1 Neistota v robotike

Robotika je veda o vnímaní a manipulovaní s fyzickým prostredím skrz počítačom riadeným mechanickým zariadením. Robotické systémy majú spoločnú myšlienku a to že sa nachádzajú vo fyzickom svete, vnímajú prostredie pomocou senzorov a pracujú s prostredím pohyblivými časťami robota.

Neistota vzniká ak robot má nedostatok podstatných informácií pre prevedenie jeho úlohy. Táto neistota vzniká na základe piatich rôznych faktoroch:

1. **Prostredie.** K fyzickému svetu neoddeliteľne patrí jeho nepredvídateľnosť. Aj keď úroveň neistoty v upravenom prostredí ako výrobné pásy je nízka, prostredia ako napríklad diaľnice alebo obytné priestory sa dynamicky menia a sú nepredvídateľné.
2. **Senzory.** Každý senzor je stavaný tak aby vnímal iba určité dáta. Najväčšie dopad na obmedzenia senzorov majú dva faktory. Prvý, vzdialenosť a rozlíšenie senzorov na základe fyzikálnych zákonov. Za druhé, senzory sú vystavené šumom, ktoré rušivo pôsobia na merania, nepredvídateľným spôsobom a tým vyznačujú hranice pre informácie, ktoré dokážeme získať zo senzorových meraní.
3. **Robot.** Ovládanie robotov s motormi je, aspoň do určitej miery, nepredvídateľné. Dôvodom je šum ovládania a opotrebenia. Niektoré ovládania, ako napríklad priemyselné robotické ramená, sú vcelku presné. Naopak nízkorozpočtové pohyblivé roboty môžu byť extrémne nepresné.
4. **Model.** Model je abstrakcia reálneho sveta. Pre zjednodušenie sa modelujú hlavne dôležité aspekty, zatiaľ čo tie menej dôležité sa vypúšťajú. Takýto model sa stáva do určitej miery nepresným. Nedostatky modelu sú zdrojom neistoty, ktorá je v robotike do veľkej miery ignoruje.
5. **Výpočty.** Roboti pracujú v reálnom čase, čo obmedzuje množstvo času použiteľný na výpočty. Veľa najmodernejších algoritmov sú aproximované a získavajú lepšiu časovú odozvu za cenu presnosti.

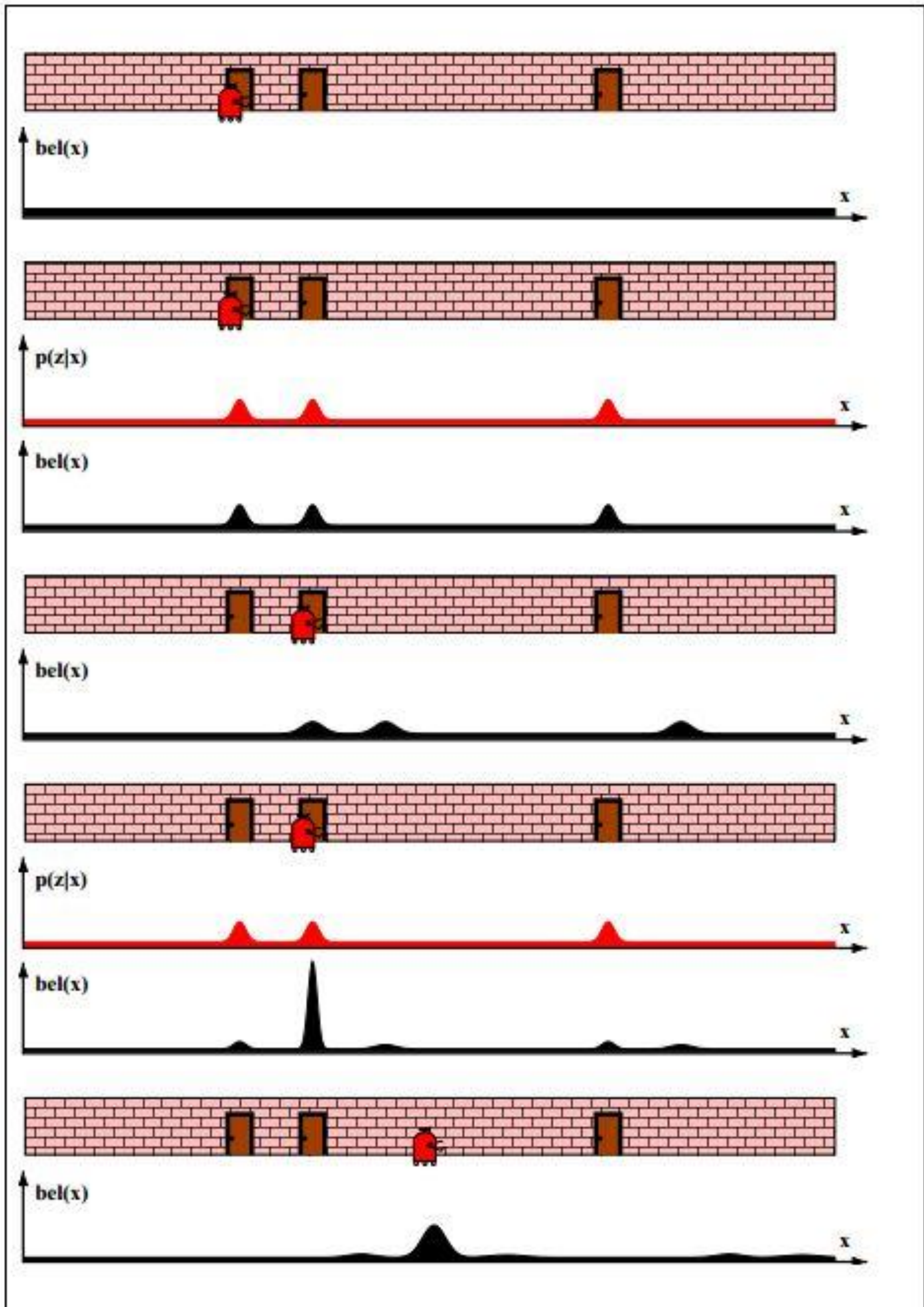
Všetky tieto faktory zvyšujú neistotu v robotike. Bežne sa neistota v robotike ignoruje ale jak sa postupne roboti dostávajú z im prispôsobeného prostredia do viac a viac nepredvídateľného, schopnosť zvládať neistotu je zásadnou pre tvorbu úspešných robotov.[1]

1.2 Pravdepodobnostná robotika

Pravdepodobnostná robotika je nový spôsob prístupu k robotike, ktorá berie v úvahu neistotu vo vnímaní a činnostiach robota. Kľúčovou myšlienkou pravdepodobnostnej robotiky je vyhodnocovať neistotu výhradne používaním výpočtov pravdepodobnostnej teórie.

Pre lepšie znázornenie si uvedieme príklad lokalizácie mobilného robota. Lokalizácia je problém vyhodnocovania súradníc robota v externých referenčných rámcoch zo senzorových dát, na mape prostredia v ktorom sa robot nachádza. Obrázok 1.1 ilustruje pravdepodobnostnej prístup k lokalizácií mobilného robota. Tento konkrétny prípad sa nazýva *globálna lokalizácia*, kde robot je vložený do priestoru o ktorom nemá žiadne informácie. V pravdepodobnostnej paradigme sa aktuálne vyhodnotenie nazývane *belief*, je vyobrazené ako funkcia hustoty pravdepodobnosti pre všetky pozície na mape. To nám zobrazuje prvý diagram na obrázku 1.1, ktorý zobrazuje rovnomerné rozloženie zodpovedajúce maximálnej neistote. Povedzme, že robot vykonal meranie a zistil, že sa nachádza vedľa dverí. Zmena sa premietne do belief-u pridaním vysokej šanci na lokácie s dverami a nízke šance kdekoľvek inde. Keďže ale v našom prostredí sa nachádzajú troje dvere robot predpokladá, že môže stáť pred ľubovoľnými z nich. Z dôvodu neistoty v senzoroch sa aj na nedverových lokáciách nachádzajú nenulové hodnoty. Tretí diagram na obrázku 1.1 nám zobrazuje vplyv pohybu robota na bilief. Bilief sa posunie v smere pohybu robota. Taktiež sa trochu znížia pravdepodobnosti z dôvodu neistoty pohybu. Po následnom nasnímaní zistí blízkosť dverí s následným dopadom na belief ako zobrazuje štvrtý diagram obrázku 1.1. Algoritmus pridá pravdepodobnosti na lokácie s dverami a robot si už začína byť vcelku istý kde sa nachádza.

V príklad sme si predviedli problém vnímania ako problém vyhodnotenia pozície a ako lokalizačný algoritmus sme použili *Bayesov filter*.[1]



Obrázok 1.1 – Základná myšlienka Markovej lokalizácie: Mobilný robot počas globálnej lokalizácií.[1]

2 Teoretická časť

Pre úspešné fungovanie lokalizácie je si musíme najskôr vysvetliť niekoľko pojmov. Nastaviť robota a jeho komponenty tak aby vedel kde aký snímač má. Pripraviť ho na odosielanie správ zo senzorov a „krokomeru".

Následne preveríme spôsob zlučovania informácií, ktoré nám senzory budú poskytovať a nakoniec samotné vyhodnotenie pozície robota v priestore.

2.1 Merania zo senzorov

Každý senzor, ktorý chceme používať na určovanie polohy robota musíme nastaviť tak aby posielal správy cez ROS. Každá z týchto správ nesie informáciu z jedného senzoru zabalené v štruktúre vhodnej k zasielaniu. Generovanie týchto správ prebieha priamo v počítači robota (napr. Arduino).

K odosielaniu správ cez ROS sú určené publishery. Ako prvé treba oznámiť ROS-u, že budeme posielať správy na určitý topic. ROS vytvorí daný topic a následne môže publisher posielať vygenerované správy na topic. Zasielanie prebieha periodicky v určitom časovom intervale tak aby informácie zo senzorov boli vždy čo najviac aktuálne.

Na prijímanie správ potrebujeme vytvoriť subscriber. Ten sa „podpíše" na prijímanie správ z určeného topicu. subscriber musí mať určenú funkciu, ktorú bude volať keď dostane správu z topicu. Takéto funkcie sa nazývajú „callback". Samotný subscriber musí byť v pohotovosti na prijímanie správ. Na to slúži napríklad `ros::spin()`. Táto funkcia v podstate cykluje program a pri zmene na niektorom z topicov, ktoré majú v programe svoj subscriber, volá callback daného subscribera.

2.2 Transformácia robota

Keďže robot sa skladá z veľkého množstva častí, jak pohyblivých tak statických, je potrebné týmto častiam nastaviť ich pozíciu v súradnom systéme. Tieto systémy nazývame rámce. Robot v základe rozlišuje niekoľko rámcov (súradných systémov). Najdôležitejším z nich je rámec sveta často označovaný anglickým slovom „world". Na tento rámec priamo nadväzuje rámec základne robota. Ďalšie rámce potom môžu byť napríklad: rámec odometrie, sonaru, lidaru, atď.. Tieto rámce vytvárajú stromovú štruktúru, kde rámec sveta je vlastne najvyššie postaveným rámcom. Podmienka je, že každý rámec môže mať len jediného rodiča. Môžu ale zato mať ľubovoľný počet potomkov.

Vytváranie rámcov je možné pomocou transformačného broadcasteru. Do broadcasteru nastavíme posun a orientáciu nového rámca oproti rodičovskému rámcu. Tieto informácie následne vysiela do ROS-u na topic transformácie. Naslúchanie potom prebieha podobne ako pri senzoroch s tým, že potrebujeme transformačný listener. Ten nám podľa jeho nastavenia získava z topicu transformácie informácie o požadovanom rámci.

2.2.1 Účel transformácií

Transformácie sa vytvárajú aby sme dokázali určiť kde sa ktorý senzor nachádza a jak je otočený vzhľadom na základňu robota. To znamená že ak nám nejaký senzor pošle informáciu, že sa nachádza prekážka priamo pred ním, vieme určiť vzhľadom na transformáciu kde sa daná prekážka nachádza od robota.

Príklad: Ak máme senzor ktorý je upevnený na čele robota s transformáciou x: 0, y: 0,2 a z: 0,1 v metroch a je orientovaný v smere robota (nulové hodnoty) oproti základni. Ak nám tento senzor nameria prekážku, ktorá je vzdialená 0,5 metra od senzoru, vieme transformovať tento údaj aby odpovedal vzdialenosti prekážky od základne robota. V tomto prípade postačí pripočítať hodnoty transformácie k nameranému bodu. Nameraný bod (x: 0, y: 0,5, z: 0) + transformácia (x: 0, y: 0,2 a z: 0,1) = pozícia bodu od základne robota (x: 0, y: 0,7 a z: 0,1).

Keď použijeme túto metódu na všetky merania, dokážeme určiť pozíciu každého bodu v priestore jak oproti robotovi tak aj v súradnom systéme mapy.

2.3 Odometria

Odometria je vlastne meranie zasielané prvkami, ktoré ovplyvňujú pohyb robota. Alebo inač je to vektor určujúci zmenu pozície robota na základe pohybových prvkov.

Správy prispôsobené na posielanie odometrických informácií obsahujú hodnoty vyjadrené v základných súradných systémoch a to pozíciu, orientáciu, rýchlosť a uhlovú rýchlosť v x, y, z osiach. Je možne prijať aj hodnoty vzdialenosti a smeru (uhol), v základe to ale znamená iba prepočet týchto hodnôt tak aby sme dostali štandardný odometrický vektor hodnôt.

Samozrejme aj odometriu je potrebné transformovať aby zodpovedala potrebnému rámcu. Prevažne to bude na rámec sveta (mapy v ktorej sa robot pohybuje).

2.4 Fúzia senzorov

Keď už máme namerané hodnoty z rôznych senzorov a odometrie potrebujeme spraviť fúziu informácií aby sme ich mohli použiť pre vyhodnotenie polohy robota. V našom prípade sa o obidva problémy postará Kalmanov filter resp. jeho rozšírená verzia.

2.4.1 Kalmanov filter

Kalmanov filter je algoritmus, ktorý spracováva sériu meraní za čas, obsahujúci štatistický šum a iné nepresnosti, a produkuje odhad neznámych premenných s väčšou presnosťou ako metódy založené na jednom samostatnom meraní. Použitím Bayesovho záveru a odhadom spojeného rozdelenia pravdepodobnosti nad premennými pre každý časový rámec.

Kalmanov filter má vysoké využitie v technike. Najbežnejšie použitie je v navigácií, navádzaní a kontrolovaní vozidla a čiastočne v lietadlách. Ďalej je koncept Kalmanovho filtra dosť rozšírený pri analýze údajov závislých od času ako spracovanie signálov alebo ekonometrií. Pre túto prácu najdôležitejším použitím Kalmanovho filtra je plánovanie a kontrola pohybov robota.

Algoritmus pracuje v dvoch krokoch. A to krok predpovedania, kde Kalmanov filter vytvorí odhady premenných v aktuálnom stave spolu s ich neurčitosťou. Po vyhodnotení nasledujúceho merania, aktualizuje odhady použitím váženého priemeru (väčšia váha pre odhady s vyššou určitosťou). Algoritmus je rekurzívny. Môže fungovať v reálnom čase, len za použitia aktuálnych hodnôt, hodnôt vypočítaných v minulých krokoch a matice neurčitosti.

Je dôležité uvážiť relatívnu určitosť merania a aktuálneho odhadu stavu, bežne používaným termínom je Kalmanov nárast (gain). Kalmanov nárast je relatívna váha pre meranie a aktuálny odhad stavu a môže byť naladený pre dosiahnutie určitého výkonu. S vysokým nárastom, filter prikladá väčšiu váhu na najnovšie meranie a tým pádom je citlivejší na zmeny. Naopak nízky nárast lepšie nasleduje predpovede modelu.

Pri prevádzaní reálnych výpočtov sú hodnoty odhadu stavu a rozptyl sú zapísané v matici aby sa zvládlo viacero dimenzií vypočítať v jednej sade výpočtov. Toto umožňuje reprezentáciu lineárnych vzťahov medzi rôznymi premennými v ľubovoľnom prechodovom modely alebo rozptyle.

2.4.1.1 Pozadie modelu dynamického systému

Kalmanov filter je založený na lineárnom dynamickom systéme diskretizovaný v čase. Je modelovaný na Markovom reťazci postavenom na lineárnych operátoroch prerušovaných chybami medzi ktoré patrí napríklad Gaussov šum. Stav systému je reprezentovaný ako vektor reálnych čísel. Každým posunom v diskrétnom čase sa aplikuje lineárny operátor na stav pre vygenerovanie nového stavu s primiešaným šumom, navyše môže obsahovať informácie z ovládačov systému ak sú známe. Následne ďalší lineárny operátor je zmiešaný s väčším šumom a vytvorí výstup z pravého (skrytého) stavu. Kalmanov filter môže byť považovaný za analogický k skrytému Markovmu modelu. Hlavný rozdiel je že premenné zo skrytého stavu získavajú hodnoty v spojitom priestore. Medzi skrytým Markovým modelom a Kalmanovým filtrom je vysoká dualita.

Nato aby sme použili Kalmanov filter pre odhadnutie vnútorného stavu procesu ak poznáme iba sekvenciu sledovaní so šumom, musíme namodelovať proces zhodný s štruktúrou Kalmanovho filtra. To znamená špecifikovanie nasledujúcich matíc: $\mathbf{F}_k$, model prechodov medzi stavmi; $\mathbf{H}_k$, model pozorovania; $\mathbf{Q}_k$, rozptyl procesového šumu; $\mathbf{R}_k$, rozptyl pozorovaného šumu; a občas $\mathbf{B}_k$, model ovládania vstupu, pre každý krok v čase k.

Kalmanov filter predpokladá, že pravý stav v čase k je odvodený od stavu v čase $(k - 1)$.

$$\boldsymbol{x_k = f(x_{k-1}, u_k) + w_k} \tag{2.1}$$

Kde

- $\mathbf{F}_k$ je model prechodov medzí stavmi, ktorý je násobený s predchádzajúcim stavom $\mathbf{x}_{k-1}$
- $\mathbf{B}_k$ je model ovládania vstupu, ktorý je násobený vektorom ovládania $\mathbf{u}_k$
- $\mathbf{W}_k$ je procesový šum, ktorý predpokladáme, že je získaný z viacrozmernej normálnej distriúcie s rozptylom $\mathbf{Q}_k$

V čase k pozorovanie $\mathbf{z}_k$ pravého stavu $\mathbf{x}_k$ vypočítame ako

$$\boldsymbol{z_k = h(x_k) + v_k} \tag{2.2}$$

Kde $\mathbf{H}_k$ je model pozorovania, ktorý mapuje pravý stav priestoru do pozorovaného priestoru a $\mathbf{v}_k$ je šum pozorovania o ktorom predpokladáme, že má nulovú strednú hodnotu Gaussovho bieleho šumu s rozptylom $\mathbf{R}_k$.

Počiatočný stav a šumové vektory v každom kroku $\{\mathbf{x}_0, \mathbf{w}_1, \ldots, \mathbf{w}_k, \mathbf{v}_1, \ldots, \mathbf{v}_k\}$ sú všetky navzájom nezávislé.

Veľa reálnych dynamických systémov nesedia na tento model. V skutočnosti nenamodelované dynamiky môžu seriózne znížiť výkon filtra, aj keď údajne dokáže pracovať s neznámymi stochastickými signálmi ako vstup. Dôvod tohto správania je závislosť nenamodelovaných dynamikách na vstupe a tak môže priviesť odhadový algoritmus k nestabilite (algoritmus diverguje). Na druhú stranu signál nezávislého bieleho šumu nedonúti algoritmus divergovať. Rozlišovanie medzi šumom merania a nemodelovými dynamikami je náročné. Spracováva sa v teórií riadenia pod štruktúrou robustného riadenia.

2.4.1.2 Detail

Kalmanov filter vyhodnocuje odhady rekurzívne. To znamená, že iba odhad stavu z predchádzajúceho časového kroku a aktuálne meranie je potrebné na výpočet odhadu pre aktuálny stav. Oproti technikám, ktoré počítajú odhad po skupinách, nevyžaduje žiadnu históriu pozorovaní ani odhadov. To čo nasleduje je notácia $\hat{x}_{n|m}$, ktorá reprezentuje odhad **x** v čase *n* pre dané pozorovania do času *m*, vrátane ($m \leq n$).

Stav filtru je reprezentovaný dvoma premennými:

- $\hat{x}_{n|m}$, koncový (posteriori) stav odhadu v čase *k* pre dané pozorovania do času *k*,
- $P_{n|m}$, matica koncových rozptylov chyby (miera presnosti odhadu stavu)

Kalmanov filter môže byť zapísaný ako jediná rovnica. Väčšinou ale býva rozčlenená do dvoch fáz: „Predpoveď“ a „Aktualizácia“. Fáza predpovede využíva odhad stavu z predchádzajúceho kroku na vyhodnotenie stavu v aktuálnom stave. Tento predpovedaný odhad stavu je známy ako *priori*, pretože aj keď vyhodnocuje stav aktuálneho kroku, nezohľadňuje pozorovanie z aktuálneho kroku. V aktualizačnej fáze je aktuálna a *priori* hodnota skombinovaná s aktuálnymi informáciami o pozorovaní a vytvoria odhad stavu. Tento vylepšený odhad sa nazýva *posteriori* odhad stavu.

Typicky sa striedajú dve fáze. Predpovedanie postupu stavu dokým nenastane ďalšia naplánované pozorovanie a aktualizácia zahŕňajúce pozorovanie. Ak je z nejakého dôvodu pozorovanie neprístupne tak sa môže byť preskočených niekoľko fáz aktualizácie zatiaľ čo sa prevedie viacnásobná predpoveď, naopak ak je dostupných viacero pozorovaní súčasne, môže sa previesť viacnásobná aktualizácia.

2.4.2 Rovnice a vysvetlenie

Kalmanov filter pracuje v dvoch základných krokoch: predpoveď a aktualizácia. V kroku predpovede pracuje filter s hodnotami z minulého vyhodnotenia a vypočíta predpokladanú pozíciu a kovarianciu. Hodnoty z tohto kroku sa nazýva „priori“. Druhý krok, aktualizácia, vezme hodnoty priori a upraví ich podľa nameraných hodnôt. Veľkosť akou namerané hodnoty ovplyvnia výsledok udáva Kalman gain. Optimálny gain sa vypočíta pomocou kovariancií. V základe určuje význam alebo dôveryhodnosť nameraných hodnôt. Po kroku aktualizácie dostaneme „posteriori“ hodnoty, ktoré by mali zodpovedať polohe robota v priestore.

Nepresnosti, ktoré sa môžu počas procesu vyskytnúť nazývame hluk (angl. noise). Rozlišujeme dva druhy hluku a to: Procesový hluk a Hluk pri meraní. Hluk sa pripočítava do kovariancie. Procesový hluk aj pri velice nízkych nenulových hodnotách zlepšiť vyhodnocovanie a napomáha udržať výsledky v norme. Hluk pri meraní sa naopak doporučuje nastaviť na vysoké hodnoty aby sa mohol filter inicializovať na správne hodnoty.

Keďže my potrebujeme aby filter spracovával viacero senzorov súčasne a väčšina senzorov odosiela viac ako jednu hodnotu, potrebujeme filter upraviť tak aby bol schopný spracovať viacero hodnôt. Riešením sú vektory, matice a ich operácie.

Rozšírením Kalmanovho filtra prevedieme filter do nelineárnej hladiny. Aby sme previedli lineárny Kalmanov filter na nelineárny potrebujeme vytvoriť maticu parciálnych derivácií. Takáto matica sa nazýva Jacobian.

Teraz si vypíšeme rovnice Kalmanovho filtra a popíšeme si jednotlivé členy rovníc. Pre zjednodušenie si môžeme rozdeliť rovnice na tri kroky:

Model:

$$\boldsymbol{x_k} = \boldsymbol{f}(\boldsymbol{x_{k-1}}, \boldsymbol{u_k}) + \boldsymbol{w_k} \tag{2.3}$$

$$\boldsymbol{z_k = h(x_k) + v_k} \quad \textbf{(2.4)}$$

Rovnica **Chyba! Nenalezen zdroj odkazů.** – Stavovo prechodový model, vypočítava nové stavy priori z Jacobianu vynásobeného predchádzajúcim stavom a ovládacým vstupom. Nakoniec sa pripočíta hluk. Často sa výsledok funkcie $f(x) = x$ tým pádom aj jej Jacobian je identitná matica.

Rovnica 2.2 – Model merania, je vlastne spôsob ktorým prinášame hodnoty z merania do filtra. Funkcia h je obdobne ako f dosť často len identitná ako aj jej Jacobian. Pre zjednodušenie sa môžeme na maticu H pozerať ako na nejakú „výberovú“ maticu, ktorá určí ktoré stavy budeme vyhodnocovať.

Predpoveď:

$$\boldsymbol{\hat{x}_k = f(\hat{x}_{k-1}, u_k)} \quad \textbf{(2.5)}$$

$$\boldsymbol{P_k = F_{k-1} P_{k-1} F_{k-1}^T + Q_{k-1}} \quad \textbf{(2.6)}$$

Predpoveď stavu v rovnici 2.3 je v podstate len prepis 2.1 len vo vektorovom zápise. Zatiaľ čo 2.4 je výpočet priori kovariancie. Kde F je Jacobian funkcie f a Q je kovariancia procesového hluku. Čím väčšie hodnoty v Q tým presnejšie bude nasledovať veľké zmeny v dátach.

Aktualizácia:

$$\boldsymbol{G_k = P_k H_k^T \left(H_k P_k H_k^T + R\right)^{-1}} \quad \textbf{(2.7)}$$

$$\boldsymbol{\hat{x}_k = \hat{x}_k + G_k\left(z_k - h(\hat{x}_k)\right)} \quad \textbf{(2.8)}$$

$$\boldsymbol{P_k = (I - G_k H_k) P_k} \quad \textbf{(2.9)}$$

V kroku aktualizácie si potrebujeme vypočítať optimálny Kalmanov gain 2.5. Pre jeho výpočet pripočítavame hluk pri meraní, ktorého veľkosť určuje dôveryhodnosť meraní. Čím väčšie hodnoty tým menšej bude filter brať v úvahu meranie. Nakoniec upravíme stavový vektor pomocou gainu. Taktiež kovarianciu upravíme gainom. Matica *I* v rovnici je identitná matica o rozmeroch *počet_stavov * počet_stavov*.

3 Navigačný zásobník

Základná pre-rekvizita je robot s ROS, ďalej potrebujeme mať vytvorený `tf` transformačný strom a publikované dáta zo senzorov správnym typom ROS správ. Pre zaručenie čo najlepšej funkčnosti navigačného zásobníka musíme nakonfigurovať tvar a rozmery robota.

3.1 Hardwarové požiadavky

Aj keď je navigačný zásobník vytvorený tak bol čo naviac všestranný ako len ide, existuje tri hlavné požiadavky na hardware:

1. Je použiteľný jak pre diferenciálny pohon tak aj na holonimický pohon. Predpokladá že pohybová základňa je ovládaná posielaním rýchlostnými príkazmi vo forme: rýchlosť v ose x, rýchlosť v ose y a uhlová rýchlosť theta.
2. Vyžaduje namontovaný rovinný laser niekde na pohybovej základni. Využíva sa na mapovanie budov a lokalizáciu.
3. Navigačný zásobník bol vyvinutý na robotovi v tvare štvorca, preto jeho výkon je najlepší na robotoch ktorý sú približne štvorcový alebo okrúhly.

3.2 Nastavenie robota

3.2.1 ROS

Ako som už písal vyššie, základnou pre-rekvizitou je robot s ROS.

3.2.2 Konfigurácia transformácie

Pre navigáciu potrebujeme transformačný strom publikovaný pomocou knižnice `tf`. Transformačný strom popisuje posun, v mieste aj rotácii, medzi jednotlivými koordinačnými rámcami. Pre vytvorenie transformačného stromu si potrebujeme definovať (namerať) posuny medzi rámcami. Potom vytvoríme vzťah medzi rámcami a uložíme ich do transformačného stromu. V zásade každý uzol v transformačnom strome zodpovedá jednému koordinačnému rámcu a hrany zodpovedajú transformácii, ktorú treba previesť pre presun z aktuálneho uzla do jeho potomka. Stromová štruktúra sa využíva aby medzi ľubovoľnými dvoma koordinačnými rámcami vždy viedla len jedna hrana a predpokladá, že tieto hrany sú vždy orientované smerom od rodiča k potomkovi.

Keď už máme vytvorený transformačný strom potrebujeme tieto informácie vysielať do systému kde budú k dispozícií k naslúchaniu a ich následnému využitiu.

Ako prvý musíme vytvoriť uzol, ktorý bude vysielať transformáciu do nášho systému. K tomu si potrebujeme vytvoriť objekt typu `TransformBroadcaster`, ktorý pre odoslanie potrebuje nasledujúcich päť argumentov. Prvý je rotácia rodičovského rámca od potomka. Druhý je posun medzi rámcami. Ako tretie je časová značka. Štvrtý a piaty sú názvy rodičovského a potomkovho uzla v tomto poradí.

Ďalej potrebujeme uzol, ktorý bude naslúchať dátam vysielaným cez ROS a používať ich k transformácií bodov. Vytvoríme funkciu, ktorá bude transformovať bod z rámca potomka pomocou dát vysielaných do systému do rámca rodiča. Ako zdroj bodu je senzorový prúd vysielaný cez ROS.

3.2.3 Vysielanie senzorových prúdov cez ROS

Tieto prúdy môžu obsahovať buďto správu typu `LaserScan` alebo `PointCloud`. Hlavička takýchto správ sa skladá z troch polí a to z poľa `seq`, ktorý sa po každom poslaní správy automaticky inkrementuje, `stamp` s časovou pečiatkou a `frame-id` pre názov rámcu, ktorý správu vygeneroval. Obsah správy je naplnený dátami z daného senzora.

Navigačný zásobník využíva informácie zo senzorov na vyhýbanie sa prekážkam vo svete.

3.2.4 Odometrické informácie

Navigačný zásobník využíva `tf` na vyhodnotenie pozície robota. Ale z dôvodu, že `tf` neposkytuje informácie o rýchlosti robota, navigačný zásobník potrebuje publikované jak transformácie tak aj odometrické správy cez ROS, ktoré ich obsahujú.

Odometrické správy obsahujú odhad pozície a rýchlosti robota vo voľnom priestore. Hodnota `pose` obsahuje odhad pozície robota v odometrickom rámci. Do hodnoty môžeme pridať neurčitosť pre odhad pozície. Hodnota `twist` potom odpovedá rýchlosti robota v potomkovom rámci. Takisto môže obsahovať neurčitosť odhadu rýchlosti.

Odometrické zdroje musia vysielať informácie pomocou knižnice `tf` o koordinačnom rámci, ktorý ho spravuje.

3.2.5 Ovládač základne

Navigačný zásobník predpokladá, že môže ovládať pohyb robota príkazmi o rýchlosti zasielaním `geometry_msg/Twist` správ s predpokladom, že sa nachádza v základnom koordinačnom rámci s prístupom k topicu `cmd_vel`. To znamená, že musí existovať uzol, ktorý dokáže premeniť hodnoty rýchlosti (x,y,theta) na pohybové príkazy a poslať ich na pohybovú základňu.

3.3 Nastavenie navigačného zásobníka

Predpokladá, že všetky požiadavky uvedené vyššie sú splnené. To znamená, že každý senzor, ktorý budeme využívať bude vysielať informácie o koordinačnom rámci pomocou `tf`. Ďalej, vysielanie odometrických informácií pomocou `tf` a odometrických správ a prijímanie príkazov na rýchlosť a ich zasielanie na pohybovú základňu.

3.3.1 Vytvorenie packagu

Musíme vytvoriť package, ktorý bude obsahovať všetky súbory nastavení a spúšťacie súbory pre navigačný zásobník. Je dôležité vytvoriť balíček so závislosťami na každom balíčku použitom k nastaveniu robota ako aj na `move_base` balíčku tak aby bolo možné úspešne spustiť navigačný zásobník.

3.3.2 Vytvorenie spúšťacieho konfiguračného súboru pre robota

Po vytvorení pracovného priestoru pre naše konfiguračné a spúšťacie súbory potrebujeme vytvoriť súbor `roslaunch`, ktorý si vyvolá všetky potrebné hardwarové a transformačné publikácie. Ďalej potrebujem daný súbor nastaviť na nášho konkrétneho robota. Pre každý senzor, ktorý bude navigačný zásobník využívať. Treba vyvolať balíček s ROS ovládačom, typ ovládaču, názov uzla a ďalšie

parametre potrebné pre spúšťaný uzol. Podobne je potreba spustiť aj odometický uzol a nakoniec aj transformačnú konfiguráciu.

3.3.3 Konfigurácia máp s oceneniami

Navigačný zásobník využíva dve mapy s oceneniami (ďalej costmap) na ukladanie informácií o prekážkach vo svete a to: globálne a lokálne. Globálnu mapu používa pre globálne plány (dlhodobé plány naprieč celým prostredím) a lokálnu mapu, ktorá slúži na vyhýbanie sa prekážkam v blízkom okolí. Máme nastavenia, ktoré chceme aby obidva typy máp dodržiavali a potom individuálne nastavenia pre každý typ zvlášť.

Obecne navigačný zásobník využíva costmapy na ukladanie informácií o prekážkach vo svete. Aby to mohlo fungovať potrebujeme nastaviť costmapy tak aby naslúchali aktualizáciám z topicov senzorov. K tomu potrebujeme nastaviť prahy na ktoré budú senzory vyhodnocovať prekážky alebo voľný priestor. Ďalej potrebujeme nastaviť rozmery robota. Tie nastavujeme s predpokladom, že stred robota má hodnotu (0.0, 0.0). Zadávanie je možné v smere alebo v protismere hodinových ručičiek. Pre prípad kruhového tvaru robota sa zadá hodnota polomeru. Ešte potrebujeme zadať hodnotu inflačného polomeru, ktorý by mal byť nastavený na maximálnu vzdialenosť od prekážky v ktorej je ohodnotenie costmapy navýšené. Nakoniec potrebujem vytvoriť zoznam senzorov, ktoré budú dodávať informácie costmapám, a nastaviť parametre samotným senzorom zo zoznamu. Tieto parametre sú: názov koordinačného rámca senzora, typ správy podľa topicu, povolenie pridávať a odstraňovať informácie o prekážkach.

Pre nakonfigurovanie globálnej costmapy potrebujeme definovať rámec v ktorom bude spustená a rámec, ktorý bude odkazovať na základňu robota. Potom už len obnovovacia frekvencia v Hz.

Konfigurovanie lokálnej costmapy prebieha podobne ako pri globálnej, sú tam ale tieto parametre navyše. Parameter určujúci frekvenciu vysielania vizualizačnej informácie. Nastavenie centrovania mapy na robota a výšku, šírku a rozlíšenie costmapy.

3.3.4 Konfigurácia základného lokálneho plánovača

Lokálny plánovač je zodpovedný za prepočítavanie príkazov o rýchlosti a ich zasielanie do pohybovej základne. Potrebujeme upraviť nastavenia podľa špecifikácií pre konkrétneho robota.

Lokálny plánovač poskytuje ovládač, ktorý vedie pohybovú základňu v rovine. Tento ovládač slúži ako prepoj medzi trasovacím plánovačom a robotom. S použitím mapy, plánovač vytvára kinematickú trajektóriu pre robota od štartu k cieli. Počas cesty plánovač ohodnocuje okolie robota pomocou funkcie, reprezentované mriežkovou mapou. Funkcia ohodnocuje prechody medzi bunkami mriežky. Úlohou ovládača je využívať tieto informácie a vyhodnotiť rýchlosti dx, dy ,dtheta a posielať ich robotovi.

3.3.5 Vytvorenie spúšťacieho súboru pre navigačný zásobník

Keď už máme všetky potrebné konfigurácie hotové, potrebujeme všetko spojiť dohromady do spúšťacieho súboru pre navigačný zásobník.

4 Návrh implementácie

V tejto kapitole sa pokúsim čo najpodrobnejšie vysvetliť a popísať postup implementácie systému pre fúziu dát zo senzorov a teda rozšírený kalmanov filter.

Pre testovanie filtra som používal simulačné prostredie Gazebo v ktorom som si vytvoril model robota s rôznymi senzormi.

4.1 Prehľad

Z môjho výskumu som zistil, že pre úspešné určenie polohy a navigovanie je potrebné najprv vytvoriť koordinačné rámce a transformačný strom. To umožňuje robotovi určiť rozmiestnenie a rotáciu senzorov (a iných prvkov snímajúcich pohyby robota a jeho okolie) oproti základni robota. Keď už robot pozná umiestnenie senzorov dokáže dátam, posielaných zo senzorov, priradiť kontext. To znamená, že ak vie že má senzor na pravej strane tak prekážky nasnímané senzorom sa nachádzajú napravo do neho (robota).

Ďalej potrebujeme zaistiť posielanie správ zo senzorov v správnej forme (napr. PointCloud). Musíme zaistiť pravidelné vysielanie týchto správ. Keďže navigácia ako aj polohovací systém pracuje v reálnom čase je potrebné označovať tieto správy časovou pečiatkou.

Obdobne potrebujeme zaistiť aj dáta z odometrie. Z dôvodu že skrz transformácie nie je možné zasielať informácie o rýchlosti, potrebujeme súčasne vysielať aj správy cez ROS. Transformácia z odometrického rámcu nám poskytne kontext kde odometrické správy určia polohu a rýchlosť robota.

Pre vyhodnotenie smerodajných informácií o polohe potrebujeme najprv dáta z rôznych senzorov zlúčiť. Navyše každá správa o meraní obsahuje aj šum ktorý znižuje presnosť merania. Riešenie pre tieto problémy predstavuje Kalmanov filter, ktorý s veľkou úspešnosťou vyhladí šum v meraniach a prevedie fúziu dát meraní. Výhodou Kalmanovho filtra je, že dokáže spracovať dohromady serie meraní závislých na čase a jeho schopnosť pracovať iba s predpoveďou (vypočítanou z minulého kroku) a aktuálne nameranou hodnotou.

4.1 Vytvorenie modelu robota

Z dôvodu testovania potrebujem vytvoriť jednoduchý model robota do simulácie. Model bude obsahovať senzory, ktoré budú vysielať simulované dáta pre kontrolu a testovanie lokalizačného programu. Prvky modelu robota je potreba nastaviť tak aby boli schopné čo najvernejšie simulovať realitu.

4.1.1 Základné nastavenie

Aby sme mohli začať, musíme zhodnotiť koľko a akých koordinačných rámcov budeme potrebovať. Každý senzor by mal mať vlastný rámec. Ďalej potrebujeme rámec pre odometriu a samozrejme rámec základne. Keď už máme potrebné rámce potrebujeme zistiť vzťahy medzi rámcami. Tieto vzťahy zodpovedajú rozdielu polohy a rotácie medzi rámcami v osách x, y a z (v metroch) pre polohu a sklon, natočenie a výchylka (v radiánoch) pre rotáciu. Vzťahy sa skladajú z rodiča a potomka kde hociktorý potomok môže mat práve jedného rodiča. Zo vzťahov vytvorím transformácie a z nich následne vznikne transformačný strom. Aby mohol tento strom vzniknúť musia rámce svoje transformácie vysielať (publikovať) a zároveň musí existovať uzol ktorý týmto vysielaniam naslúcha.

4.1.2 Vytvorenie modelu a transformácií

Keď už máme náhľad aké potrebujeme rámce začneme vytvárať samotného robota. Model robota pre simulátor budeme písať v jazyku *SDF*, ktorý používa syntax *xml.*

Pre vytvorenie nového modelu si vytvoríme súbor s názvom robota *.config* a *.sdf*. Do súboru *.config* si nastavíme fyzické atribúty modelu a jeho počiatočnú pozíciu na mape. Tieto atribúty vpisujeme pod značky *visual*. Aby však model dokázal aj interagovať s okolím musíme nastaviť aj atribúty v *collision.* Collision by mal byť len zjednodušenou verziou vizualu aby zbytočne nezaberal výpočet a modelovanie kolizných parametrov výpočetnú silu simulátora.

Preto pre našeho robota zvolíme „bublinu" okolo robota ako kolizný atribút. Ako vizuál zvolíme jednoduchý hranol ako chassis s dvomi oválnymi kolečkami, na každej strane jedno. Kolečká pripojíme k chassis pomocou *joint* značky. Ešte pridáme nejaké senzory a máme robota na simuláciu.

Nato aby mohol byť model úspešne premietnutý na mape simulátora musíme ešte do súboru *.sdf* priradiť transformácie robota na svet. Taktiež nastavíme vysielanie odometrie na požadovaný topic (*/odom*). Senzory, ktoré sme si pridali na modle je potrebné tak isto nastaviť aby vysielali svoje správy na topicy. K správnemu fungovaniu ale senzory potrebujú ovládač. Ten priradíme k senzoru pod značkou *plugin* ako aj topic na ktorý bude senzor vysialať.

Nutnou značkou každého modelu v simulátore je *inertia*, ktorá určuje fyziku modelu.

4.2 Práca s ROS-om

Každý senzor odosiela určitú správu s dátami. ROS umožňuje tieto správy posielať pomocou topicov. Jedna správa na jeden topic. Aby sme mohli začať s fúziou potrebujeme dostať dáta zo senzorov do jedného uzla. Vytvoríme preto viacnásobný subsrciber, ktorý bude prijímať merania zo všetkých senzorov, ktoré chceme využiť na lokalizáciu. Subscriber nám volá pre každý prípad callback funkciu. V tejto funkcií sa dáta pripravia na spracovanie filtrom.

4.3 Kalmanov filter

Implementácia Kalmanovho filtra je v zjednodušene povedané iba napísanie rovníc ktoré sa cyklicky volajú s novými hodnotami merania. Výsledok filtra sa potom publikuje na určitý topic. O niečo náročnejšie je vytvoriť Kalmanov filter tak aby dokázal prijať ľubovoľný počet rôznych senzorov a posielať na výstup iba požadované dáta.

Aj keď napísanie rovníc znie jednoducho, naplnenie a nastavenie matíc tak aby operácie nad maticami boli možné je niekedy problém. Pre prácu s maticami som si vybral knižnicu Eigen, ktorá poskytuje potrebné operácie s maticami.

4.3.1 Vytvorenie základu

Ako základ som si teda vytvoril triedu *kalman_filter*. Pri inicializácií triedy sa nastavia hodnoty filtra tak aby matice neporušovali pravidlá operácií nad maticami.

Pri prvom priechode cyklu filtra si nastavíme hodnotu stavového vektoru podľa nameraných hodnôt zo senzorov. Tým zabezpečím aby hodnoty získané po vyhodnotení celého cyklu boli korektné. Následne a v každej ďalšej iterácií sa volá funkcia predpovede *prediction*. Funkcia obsahuje rovnice 2.4, 2.5 a 2.6 pomocou ktorých sa vypočíta „priori" stavový vektor a procesová kovariancia.

Po výpočte „priori“ hodnôt sa zavolá funkcia *correction* ktorá má opraviť predpoveď predchádzajúcich výpočtov. Pre výpočet optimálneho gainu sa vypočíta rovnica 2.7. Takto získaný gain nám pomôže pri vyhodnotení dôveryhodnosti meraní v rovnici 2.8 a korekcií „posteriori“ stavového vektoru ktorý je výsledkom jak tejto rovnice tak aj filtra. Nakoniec ešte opravíme kovarianciu a celý cyklus môže začať odznova.

4.3.2 Generalizácia

Keď nám už funguje základ filtra môžeme ho začať rozširovať a generalizovať aby dokázal prijať čo najväčšie množstvo rôznych správ od rôznych senzorov.

4.3.2.1 Pozícia

Vo funkcii *setPosition_estimation* je možné nastaviť ktoré hodnoty bude filter využívať na určenie polohy. Môžu to byť pozícia v x, y, z osiach, rýchlosť týchto osiach a akcelerácia. Funkcia nastavuje stavovo-tranzitívnu maticu tak aby hodnoty správne ovplyvňovali výstup z filtra. V základe je to identitná matica rozšírená o rovnice pre vyhodnotenie pozície z rýchlosti alebo akcelerácie.

Táto funkcia dokáže prijímať a spracovávať aj hodnoty vzdialenosti a náklonu, na základe ktorých je možné výslednú pozíciu vypočítať. Tento spôsob vyhodnocovania je z podľa mňa dosť zastúpený, aj keď sa nejedná o štandardný postup, a tak si vyslúžil miesto v mojom programe.

Ďalšia funkcia, ktorá sa týka pozície, je jej výpis na výstup resp. či bude filter vyhodnocovať stav robota v určitých osách. Je to funkcia *setPosition_output* a mení maticu meraní tak aby zodpovedala výstupu, ktorý požadujeme.

4.3.2.2 Orientácia

Podobne ako pre pozíciu aj pre orientáciu sú funkcie, ktoré nastavujú filtrovanie s hodnotami orientácie.

Tu je vzniká problém s výpočtami. Je to z dôvodu že orientácia sa počíta buď pomocou Eulerových uhlov alebo Quaternionov. Keďže ale správy v ROS-e používajú prevažne quaterniony na vyjadrenie orientácie v trojrozmernom priestore, primárne aj môj filter počíta s nimi. Zároveň ale dokáže pracovať aj s Eulerovými uhlami.

4.3.2.3 GPS

Dáta prijímané z GPS je náročnejšie na spracovanie. Keďže GPS posiela zemepisnú dĺžku a šírku, potrebujeme tieto merania previesť na rámec mapy v ktorej sa robot pohybuje. Po transformácií na rámec našej mapy nám GPS poskytuje hodnoty, ktoré môžeme priamo použiť vo filtri až na orientáciu v theta,.

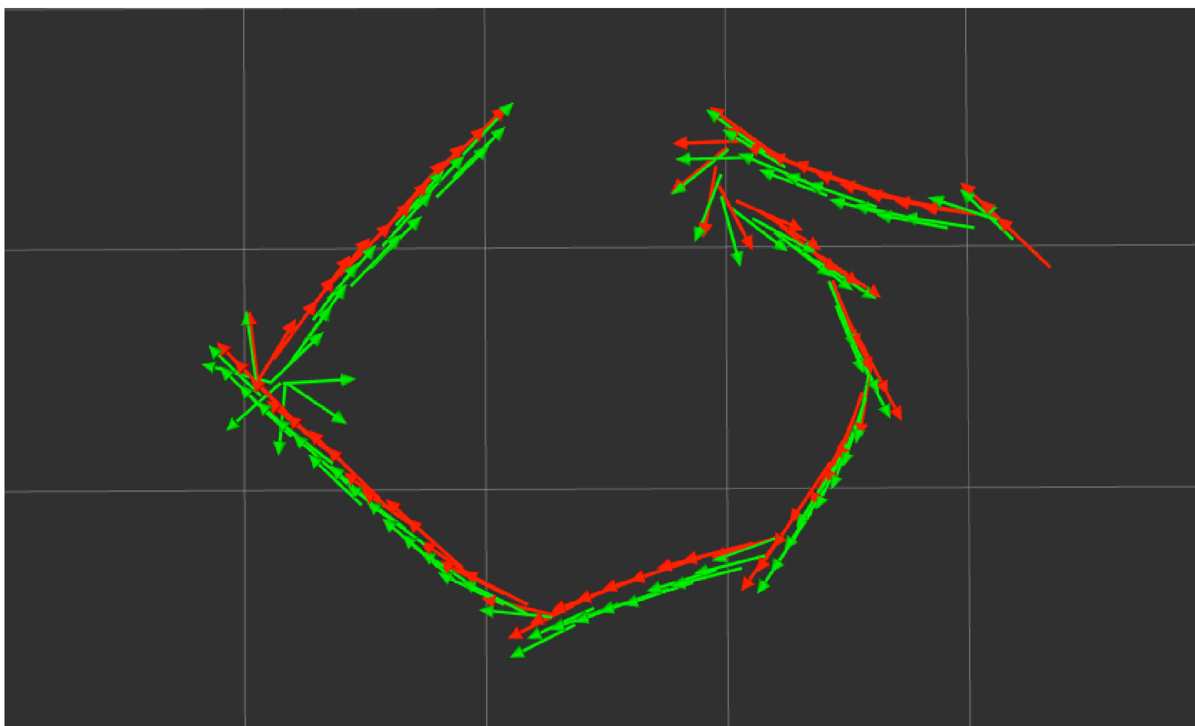
4.4 Presnosť lokalizácie

Pre otestovanie presnosti lokalizácie sme použili program RViz, ktorý prijíma správy a umožňuje ich vykreslovanie na mapu. Pre naše testovanie sme použili odometrické správy, pretože zobrazujú jak pozíciu a orientáciu tak aj predchádzajúce stavy čo nám umožňuje porovnať presnosť vyhodnotenia Kalmanovým filtrom oproti reálnej pozícií robota. Robot a jeho dáta sú simulované pomocou simulátora Gazebo.

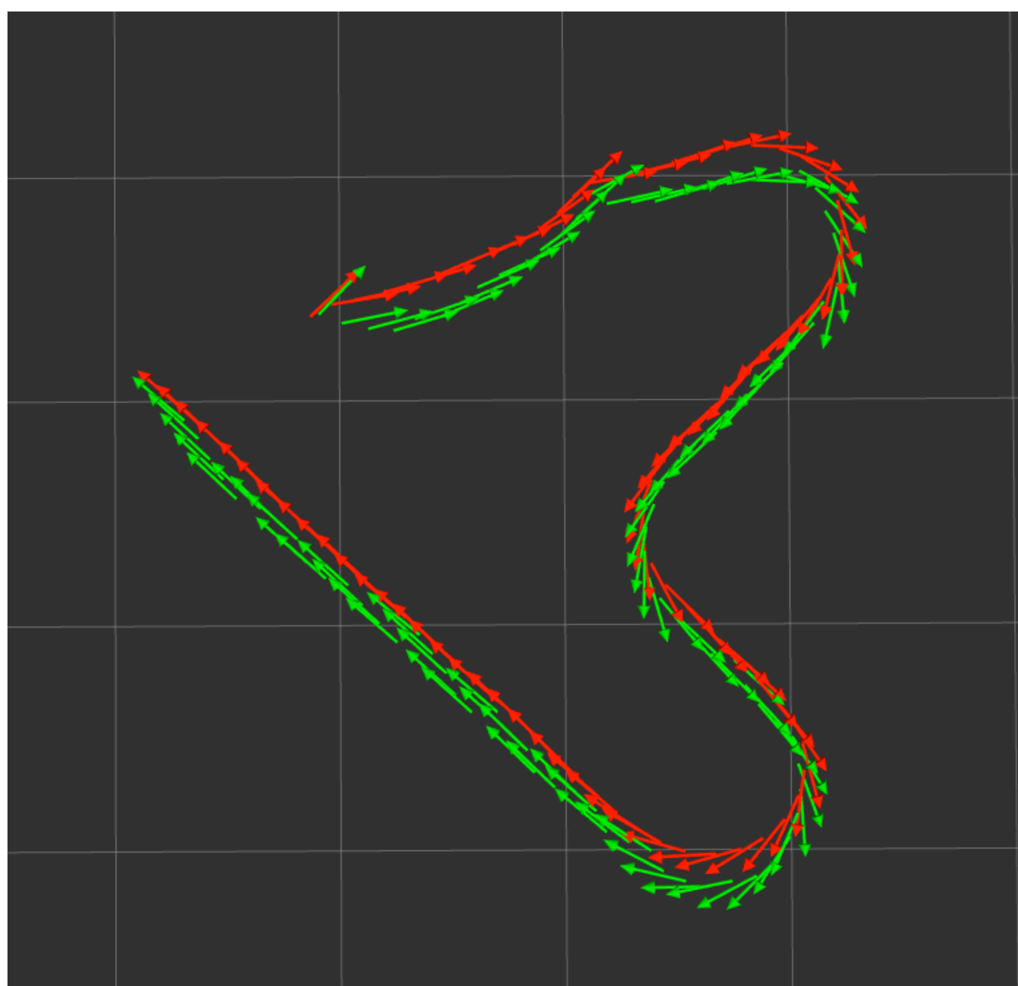
Na Obrázku 4.1 môžeme vidieť že vyhodnotená pozícia (zelená) má malé odchýlky od reálneho stavu (červená). Na tomto obrázku sme testovali hodnoty primárne na priamom pohybe a ostrých zmenách smeru. Až na posledné 90° otočenie, pri ktorom na krátku dobu výrazne chyboval v orientácií,

je vyhodnotená pozícia velice presná. Pri druhej simulácií sme zobrazenej na Obrázku 4.2 sme vyskúšali ako si poradí s postupnejšou zmenou smeru počas pohybu vpred. Ani tu si filter neviedol zle. Nakoniec sme otestovali ako lokalizácia zvládne násilný presun robota do inej polohy s inou orientáciou. Na Obrázku 4.3 vidíme odozvu filtru počas presunu robota (časť viacerých zelených šípok bez červených v okolí). Podľa toho usudzujem, že simulátor odosielal dáta aj počas tohto časového úseku a filter sa tak bez problémov zorientoval.

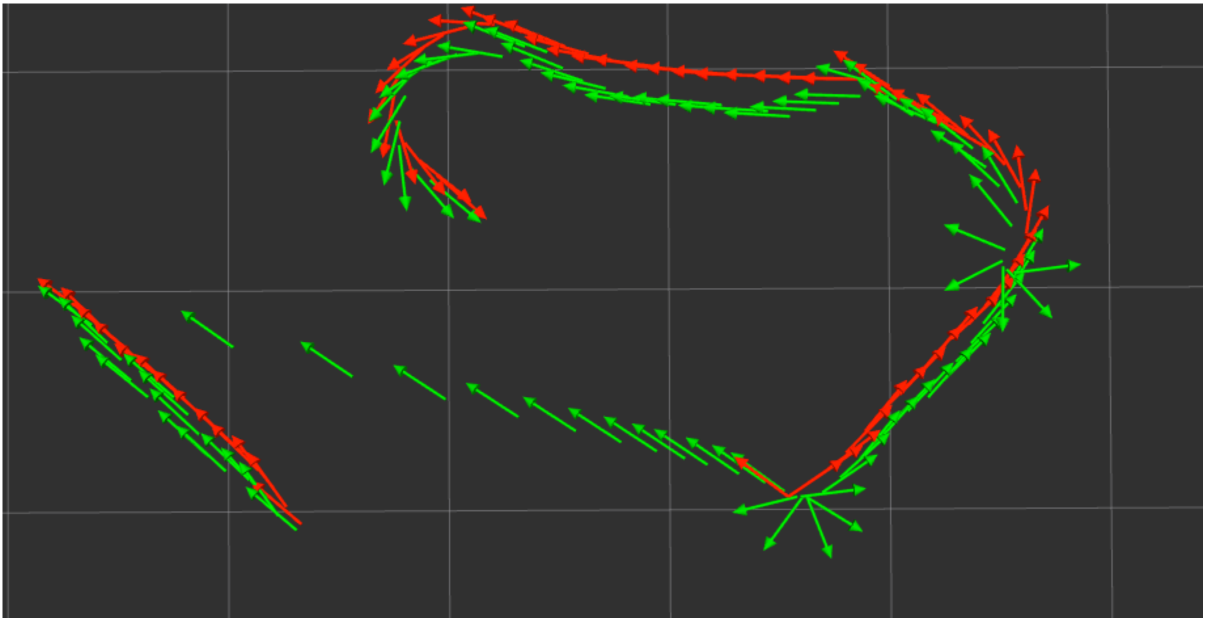
Z testovania v simulovanom prostredí môžeme zhodnotiť, že presnosť vyhodnotených údajov je dostatočne presná aby sa podľa nej robot orientoval. Hodnoty získané z filtra až na niekoľko výnimočných stavov líšia len o malú časť. Keďže ale simulátor neberie v úvahu šum vznikajúci pri meraniach, ďalšie testovanie na reálnom robotovi by bolo viac než vhodné.



Obrázok 4.1 – Testovanie presnosti pri priamom pohybe



Obrázok 4.2 – Testovanie presnosti pri zmene smeru počas pohybu



Obrázok 4.3 – Testovanie správania filtra pri dislokácií robota

5 Závěr

Zistili sme aké senzory sú potrebné k lokalizácií. Taktiež sme sa dozvedeli, že senzory využívajú rôzne správy, ktoré obsahujú dôležité informácie o meraní zo senzoru. ROS poskytuje spôsob šírenia týchto správ pomocou topicov. Pre fúziu dát zo senzorov sme použili rozšírený kalmanov filter.

Vytvorený Kalmanov filter je riešením jak problému s fúziou dát tak aj následným určovaním polohy robota v priestore. Preukázalo sa, že použitie tohto filtra je elegantným spôsobom riešenia, keďže počet senzorov, ktorých merania budeme posielať do filtru, je teoreticky obmedzený iba formuláciou nameraných hodnôt. Súčasne s možnosťou prispôsobenia výstupu sa stáva program dostatočne variabilným a prispôsobivým pre použitie na vyhodnocovanie polohy robota.

Testovaním v simulátore som zistil, že presnosť vyhodnocovania pozície je dostatočne vysoká aby dokázala navigovať robota v priestore. Taktiež prejavil schopnosť sa vzchopiť z nečakaného presunu. Aj keď pri simulovaní som prakticky ignoroval šum prostredia, verím že by sa filter preukázal schopným aj reálnom prostredí.

Pre ďalší vývoj si predstavujem rozšírením kompatibility prijímaných správ a podpory väčšej škály senzorov. Pretože namerané hodnoty senzorov majú rôznorodú štruktúru je prakticky nemožne sa prispôsobiť všetkým. Je ale možné pokračovať v generalizácií programu a tak zaistiť čo najmenšie úsilie potrebné k nastaveniu filtra a začatiu jeho používania.

Literatura

[1] THRUN, Sebastian, Wolfram BURGARD a Dieter FOX. *Probabilistic robotics*. London, England: MIT Press, 2006. ISBN 978-0-262-20162-9

[2] BUCKL, Katharina. *Sensor Fusion using the Kalman Filter* [online]. 8.3.2005. [Cit. 14.5.2017]. Dostupné z: http://campar.in.tum.de/Chair/KalmanFilter

[3] LEVY, Simon D. *The Extended Kalman Filter: An Interactive Tutorial* [online]. 19.12.2016. [Cit. 14.5.2017]. Dostupné z: https://home.wlu.edu/~levys/kalman_tutorial/

[4] Články. *Bzarg.com: How a Kalman filter works, in pictures* [online]. 11.8.2015. [Cit. 14.5.2017]. Dostupné z: http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/

[5] RISTIC, Branko., Sanjeev. ARULAMPALAM a Neil GORDON. *Beyond the Kalman filter: particle filters for tracking applications*. Boston, MA: Artech House, 2004. ISBN 158053631X.

[6] WELCH, Greg a Gary BISHOP. *An Introduction to the Kalman Filter* [online]. Posledná úprava 24.6.2006. [Cit. 14.5.2017]. Chapel hill: Department of Computer Science University of North Carolina at Chapel Hill. Dostupné z: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[7] ZACHAIRE, Mbaitiga. *GPS and Discrete Kalman Filter for Indoor Robot Navigation* [online]. 2011. [Cit. 14.5.2017]. Dostupné z: http://web.cecs.pdx.edu/~mperkows/temp/KALMAN/KALMAN-PAPERS/GPS=Kalman-Good.pdf

[8] *Robot operating system (ROS).* ISBN 9783319549262.

[9] QUIGLEY, Morgan, Brian GERKEY a William D. SMART. Programming robots with ROS. ISBN 1449323898.

[10] MARTINEZ, A. a FERNÁNDEZ E.. *Learning ROS for Robotics Programming*. Pact Publishing. 25.9.2013. ISBN 9781782161455.

Seznam příloh

Příloha 1. Manuál
Příloha 2. Zdrojové texty
Příloha 3. CD/DVD

```
#include <ros/ros.h>
#include <iostream>
#include <Eigen/Dense>
#include <nav_msgs/Odometry.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/TransformStamped.h>
#include <sensor_msgs/Imu.h>
#include <math.h>

class kalman_filter
{
  private:
    bool first; //defines first walkthrough of filter


    bool position_set;
    bool velocity_set;
    bool acceleration_set;
    bool odometry_set;

    int position_output_order;
    int velocity_output_order;
    int acceleration_output_order;
    int odometry_output_order;

    bool angles_set;
    bool angular_velocity_set;
    bool angular_acceleration_set;

    int angles_output_order;
    int angular_velocity_output_order;
    int angular_acceleration_output_order;

    bool pose_publish_set;
    bool odometry_publish_set;

    ros::Publisher pose_publish;   //publisher for state
    ros::Publisher odometry_publish;

  public:
    std::string world_name;
    std::string base_link_name;
    int state_number;   //number of states
    int input_number;   //number of inputs
    int output_number;
    double time;      //current time
    double dt;        //timestep
```

```
   Eigen::VectorXd state_vector;         //contains all estimated values
   Eigen::MatrixXd state_transition_matrix;    //defines dependencies of states between themself
   Eigen::MatrixXd process_noise_covariance;   //noise generated by calculation

   Eigen::VectorXd current_measurement_vector;   //values of new measurements
   Eigen::MatrixXd measurement_matrix;       //which states will be affecting which states
   Eigen::VectorXd measurement_state;       //resulting measurement data
   Eigen::MatrixXd measurement_noise_covariance; //noise generated by measurment

   Eigen::MatrixXd covariance;          //estimation covariance

   //Constructor
   kalman_filter(  int position_estimation_set = 0, //1 - x,y,z; 2 - +velocity x,y,z; 3 - +acceleration
x,y,z; 4 - range, bearing(odometry)
          int position_output_set = 0, // 1 - x,y,z; 2 - +velocity x,y,z; 3 - +acceleration x,y,z; 4 -
odometry(must be set for working range, bearing tracking)
          int orientation_estimation_set = 0, // 1 - orientation; 2 - +angular velocity; 3 - +angular
acceleration,
          int orientation_output_set = 0 // 1 - orientation; 2 - +angular velocity; 3 - +angular
acceleration
         )
   {
     values_init(); //safety initialization;

     //setting transition for states of position
     setPosition_estimation(position_estimation_set);
     setOrientation_estimation(orientation_estimation_set);

     state_vector.resize(state_number); //initialize during first walkthrough od cycle method
     state_vector = Eigen::VectorXd::Ones(state_number); //safety initialization

     //process noise setting even small numbers increase trustworthiness of filter
     process_noise_covariance.setIdentity(state_number,state_number);
     process_noise_covariance *= 0.05;


     current_measurement_vector.resize(state_number); //vector of measurement states
     current_measurement_vector = Eigen::VectorXd::Zero(state_number); //safety initialization
     //measurement matrix is setting which states will be outputed from filter
     //also sets how will measurement data affect output values
     if(position_estimation_set > 0)
       setPosition_output(position_output_set);
     if(orientation_estimation_set > 0)
       setOrientation_output(orientation_output_set);
     measurement_state.resize(output_number);

     //measurement noise determines how much information from measurement is used
```

```
    measurement_noise_covariance.setIdentity(output_number,output_number);

    //uncertainity of filter - initialized to high numbers
    covariance.resize(state_number, state_number);
    covariance.setIdentity();
    covariance *= 1000;
  }

  //initializes values
  void values_init()
  {
    first = true;

    state_number = 0;
    //input_number = 3;
    output_number = 0;
    time = 0;
    dt = 0.1;

    position_set = false;
    velocity_set = false;
    acceleration_set = false;
    odometry_set = false;
    angles_set = false;
    angular_velocity_set = false;
    angular_acceleration_set = false;

    pose_publish_set = false;
    odometry_publish_set = false;
  }

//Setter functions -------------------------------------------------------------------

  void setTimestep(double t)
  {
    dt = t - time;
  }

  void setWorld_frame(std::string name)
  {
    world_name = name;
  }

  void setPose_publisher(ros::NodeHandle node)
  {
    pose_publish_set = true;
    pose_publish = node.advertise<geometry_msgs::PoseStamped>("estimated_position", 1);
  }
```

```
  void setOdometry_publisher(ros::NodeHandle node, std::string name)
  {
    odometry_publish_set = true;
    base_link_name = name;
    odometry_publish = node.advertise<nav_msgs::Odometry>("estimated_odometry", 1);
  }

  void setPosition_estimation(int set)
  {
    if(set >= 1) {position_set = true;}
    if(set >= 2) {velocity_set = true;}
    if(set >= 3) {acceleration_set = true;}
    if(set == 4) {odometry_set = true;}

    if(position_set && velocity_set && acceleration_set)
    {
      //sets order of states in state vector
      position_output_order = state_number;
      state_number += 3;
      velocity_output_order = state_number;
      state_number += 3;
      acceleration_output_order = state_number;
      state_number += 3;

      //sets transition matrix for calculating between position velocity and acceleration
      state_transition_matrix.conservativeResize(state_number, state_number);
      state_transition_matrix.block(position_output_order, position_output_order, 9, 9).setIdentity(9,
9);
      state_transition_matrix.block(position_output_order, velocity_output_order, 3, 3) =
state_transition_matrix.block(position_output_order, velocity_output_order, 3, 3).setIdentity() * dt;
      state_transition_matrix.block(position_output_order, acceleration_output_order, 3, 3) =
state_transition_matrix.block(position_output_order, acceleration_output_order, 3, 3).setIdentity() *
((dt*dt)/2);
      state_transition_matrix.block(velocity_output_order, acceleration_output_order, 3, 3) =
state_transition_matrix.block(velocity_output_order, acceleration_output_order, 3, 3).setIdentity() *
dt;
    }
    else
    {
      if(position_set)
      {
        //sets order of states in state vector
        position_output_order = state_number;
        state_number += 3;

        //appendig rows and collumns
        state_transition_matrix.conservativeResize(state_number, state_number);
```

```
        //Zero unwanted interactions
        state_transition_matrix.block(position_output_order, 0, 3, state_number).setZero();
        state_transition_matrix.block(0, position_output_order, state_number, 3).setZero();

        state_transition_matrix.block(position_output_order, position_output_order, 3, 3).setIdentity(3,
3);
      }
      if (velocity_set)
      {
        //sets order of states in state vector
        velocity_output_order = state_number;
        state_number += 3;

        //appendig rows and collumns
        state_transition_matrix.conservativeResize(state_number, state_number);
        //Zero unwanted interactions
        state_transition_matrix.block(velocity_output_order, 0, 3, state_number).setZero();
        state_transition_matrix.block(0, velocity_output_order, state_number, 3).setZero();

        //sets transition matrix for calculating between position velocity
        state_transition_matrix.block(velocity_output_order, velocity_output_order, 3, 3).setIdentity(3,
3);
        if(position_set)
          state_transition_matrix.block(position_output_order, velocity_output_order, 3, 3) =
state_transition_matrix.block(position_output_order, velocity_output_order, 3, 3).setIdentity() * dt;
      }
      if (acceleration_set)
      {
        //sets order of states in state vector
        acceleration_output_order = state_number;
        state_number += 3;

        //appendig rows and collumns
        state_transition_matrix.conservativeResize(state_number, state_number);
        //Zero unwanted interactions
        state_transition_matrix.block(acceleration_output_order, 0, 3, state_number).setZero();
        state_transition_matrix.block(0, acceleration_output_order, state_number, 3).setZero();

        //sets transition matrix for calculating between position velocity and acceleration
        state_transition_matrix.block(acceleration_output_order, acceleration_output_order, 3,
3).setIdentity(3, 3);
        if(position_set)
          state_transition_matrix.block(position_output_order, acceleration_output_order, 3, 3) =
state_transition_matrix.block(position_output_order, acceleration_output_order, 3, 3).setIdentity() *
((dt*dt)/2);
        if(velocity_set)
```

```
            state_transition_matrix.block(velocity_output_order, acceleration_output_order, 3, 3) =
state_transition_matrix.block(velocity_output_order, acceleration_output_order, 3, 3).setIdentity() *
dt;
        }
      }
      if(odometry_set)
      {
        //sets order of states in state vector
        odometry_output_order = state_number;
        state_number += 2;
        //appendig rows and collumns
        state_transition_matrix.conservativeResize(state_number, state_number);
        //Zero unwanted interactions
        state_transition_matrix.block(odometry_output_order, 0, 2, state_number).setZero();
        state_transition_matrix.block(0, odometry_output_order, state_number, 2).setZero();

        state_transition_matrix.block(odometry_output_order, odometry_output_order, 2,
2).setIdentity(2, 2);
      }

    }

    void setOrientation_estimation(int set)
    {
      if(set >= 1) {angles_set = true;}
      if(set >= 2) {angular_velocity_set = true;}
      if(set >= 3) {angular_acceleration_set = true;}

      if(angles_set && angular_velocity_set && angular_acceleration_set)
      {
        //sets order of states in state vector
        angles_output_order = state_number;
        state_number += 3;
        angular_velocity_output_order = state_number;
        state_number += 3;
        angular_acceleration_output_order = state_number;
        state_number += 3;

        //sets transition matrix for calculating between orientation, angular velocity and angular
acceleration
        state_transition_matrix.conservativeResize(state_number, state_number);
        state_transition_matrix.block(angles_output_order, angles_output_order, 9, 9).setIdentity(9, 9);
        state_transition_matrix.block(angles_output_order, angular_velocity_output_order, 3, 3) =
state_transition_matrix.block(angles_output_order, angular_velocity_output_order, 3, 3).setIdentity()
* dt;
        state_transition_matrix.block(angles_output_order, angular_acceleration_output_order, 3, 3) =
state_transition_matrix.block(angles_output_order, angular_acceleration_output_order, 3,
3).setIdentity() * ((dt*dt)/2);
```

```
        state_transition_matrix.block(angular_velocity_output_order,
angular_acceleration_output_order, 3, 3) =
state_transition_matrix.block(angular_velocity_output_order, angular_acceleration_output_order, 3,
3).setIdentity() * dt;
      }
      else
      {
        if(angles_set)
        {
          //sets order of states in state vector
          angles_output_order = state_number;
          state_number += 3;

          //appendig rows and collumns
          state_transition_matrix.conservativeResize(state_number, state_number);
          //Zero unwanted interactions
          state_transition_matrix.block(angles_output_order, 0, 3, state_number).setZero();
          state_transition_matrix.block(0, angles_output_order, state_number, 3).setZero();

          state_transition_matrix.block(angles_output_order, angles_output_order, 3, 3).setIdentity(3, 3);
        }
        if (angular_velocity_set)
        {
          //sets order of states in state vector
          angular_velocity_output_order = state_number;
          state_number += 3;

          //appendig rows and collumns
          state_transition_matrix.conservativeResize(state_number, state_number);
          //Zero unwanted interactions
          state_transition_matrix.block(angular_velocity_output_order, 0, 3, state_number).setZero();
          state_transition_matrix.block(0, angular_velocity_output_order, state_number, 3).setZero();

          //sets transition matrix for calculating between orientation, angular velocity and angular
acceleration
          state_transition_matrix.block(angular_velocity_output_order, angular_velocity_output_order,
3, 3).setIdentity(3, 3);
          if(angles_set)
            state_transition_matrix.block(angles_output_order, angular_velocity_output_order, 3, 3) =
state_transition_matrix.block(angles_output_order, angular_velocity_output_order, 3, 3).setIdentity()
* dt;
        }
        if (angular_acceleration_set)
        {
          //sets order of states in state vector
          angular_acceleration_output_order = state_number;
          state_number += 3;
```

```
        //appendig rows and collumns
        state_transition_matrix.conservativeResize(state_number, state_number);
        //Zero unwanted interactions
        state_transition_matrix.block(angular_acceleration_output_order, 0, 3,
state_number).setZero();
        state_transition_matrix.block(0, angular_acceleration_output_order, state_number,
3).setZero();

        //sets transition matrix for calculating between orientation, angular velocity and angular
acceleration
        state_transition_matrix.block(angular_acceleration_output_order,
angular_acceleration_output_order, 3, 3).setIdentity(3, 3);
        if(angles_set)
          state_transition_matrix.block(angles_output_order, angular_acceleration_output_order, 3, 3) =
state_transition_matrix.block(angles_output_order, angular_acceleration_output_order, 3,
3).setIdentity() * ((dt*dt)/2);
        if(angular_velocity_set)
          state_transition_matrix.block(angular_velocity_output_order,
angular_acceleration_output_order, 3, 3) =
state_transition_matrix.block(angular_velocity_output_order, angular_acceleration_output_order, 3,
3).setIdentity() * dt;
      }
    }
  }

  void setPosition_output(int set)
  {
    //appends new states for output purposes
    if(set >= 1)
    {
      output_number += 3;
      measurement_matrix.conservativeResize(output_number,state_number);
      measurement_matrix.block(output_number-3, position_output_order, 3, 3).setIdentity(); //default
position output
    }
    if(set >= 2)
    {
      output_number += 3;
      measurement_matrix.conservativeResize(output_number,state_number);
      measurement_matrix.block(output_number-3, velocity_output_order, 3, 3).setIdentity();
    }
    if(set >= 3)
    {
      output_number += 3;
      measurement_matrix.conservativeResize(output_number,state_number);
      measurement_matrix.block(output_number-3, acceleration_output_order, 3, 3).setIdentity();
    }
    if(set >= 4)
```

```
      {
        output_number += 2;
        measurement_matrix.conservativeResize(output_number, state_number);
        measurement_matrix.block(output_number-2, odometry_output_order, 2, 2).setIdentity();
        //setting equations for estimating x,y coordinates from range and bearing
        measurement_matrix(position_output_order, odometry_output_order) =
cos(state_vector(odometry_output_order));
        measurement_matrix(position_output_order, odometry_output_order+1) =
sin(state_vector(odometry_output_order));
        measurement_matrix(position_output_order+1, odometry_output_order) = (-
sin(state_vector(odometry_output_order)))/state_vector(odometry_output_order+1);
        measurement_matrix(position_output_order+1, odometry_output_order+1) =
(cos(state_vector(odometry_output_order)))/state_vector(odometry_output_order+1);
      }
    }

    void setOrientation_output(int set)
    {
      //appends new states for output purposes
      if(set >= 1)
      {
        output_number += 3;
        measurement_matrix.conservativeResize(output_number,state_number);
        measurement_matrix.block(output_number-3, angles_output_order, 3, 3).setIdentity(); //default
position output
      }
      if(set >= 2)
      {
        output_number += 3;
        measurement_matrix.conservativeResize(output_number,state_number);
        measurement_matrix.block(output_number-3, angular_velocity_output_order, 3, 3).setIdentity();
      }
      if(set >= 3)
      {
        output_number += 3;
        measurement_matrix.conservativeResize(output_number,state_number);
        measurement_matrix.block(output_number-3, angular_acceleration_output_order, 3,
3).setIdentity();
      }
    }

//Main Equations ---------------------------------------------------------------------------

    void model(Eigen::VectorXd state_vector_previous)
    {
      //calculates priori state of values
      state_vector = state_transition_matrix * state_vector_previous;
      //defines usege of measurements
```

```
        measurement_state = measurement_matrix * current_measurement_vector;
    }

    //calculates new state - priori
    void prediction()
    {
      //Project the state ahead
      model(state_vector);
      //Project the covariance ahead
      covariance = state_transition_matrix * covariance * state_transition_matrix.transpose() +
process_noise_covariance;
    }

    //makes correction of priori estimation - result posteriori
    void correction()
    {
      Eigen::MatrixXd gain;
      //Compute the Kalman gain
      gain = covariance * measurement_matrix.transpose() * (measurement_matrix * covariance *
measurement_matrix.transpose() + measurement_noise_covariance).inverse();
      //Update state estimate
      state_vector = state_vector + gain * (measurement_state - (measurement_matrix * state_vector));
      //Update the covariance
      covariance = (Eigen::MatrixXd::Identity(state_number, state_number) - gain *
measurement_matrix) * covariance;
    }

//Publishers -----------------------------------------------------------------------------

    void publish_state_estimate_pose()
    {
      geometry_msgs::PoseStamped message;

      //setting header of message
      message.header.stamp = ros::Time::now();
      message.header.frame_id = world_name;
      //filling data into message
      if(position_set)
      {
        message.pose.position.x = state_vector(position_output_order);
        message.pose.position.y = state_vector(position_output_order+1);
        message.pose.position.z = state_vector(position_output_order+2);
      }
      if(angles_set)
      {
        //angles to quaternions
        double t0 = std::cos(state_vector(angles_output_order+2) * 0.5);
        double t1 = std::sin(state_vector(angles_output_order+2) * 0.5);
```

```
    double t2 = std::cos(state_vector(angles_output_order) * 0.5);
    double t3 = std::sin(state_vector(angles_output_order) * 0.5);
    double t4 = std::cos(state_vector(angles_output_order+1) * 0.5);
    double t5 = std::sin(state_vector(angles_output_order+1) * 0.5);

    message.pose.orientation.x = t0 * t3 * t4 - t1 * t2 * t5;
    message.pose.orientation.y = t0 * t2 * t5 + t1 * t3 * t4;
    message.pose.orientation.z = t1 * t2 * t4 - t0 * t3 * t5;
    message.pose.orientation.w = t0 * t2 * t4 + t1 * t3 * t5;
  }

  pose_publish.publish(message);
}

void publish_state_estimate_odometry()
{
  nav_msgs::Odometry message;

  //setting header of message
  message.header.stamp = ros::Time::now();
  message.header.frame_id = world_name;
  message.child_frame_id = base_link_name;
  //filling data into message
  if(position_set)
  {
    message.pose.pose.position.x = state_vector(position_output_order);
    message.pose.pose.position.y = state_vector(position_output_order+1);
    message.pose.pose.position.z = state_vector(position_output_order+2);
  }

  if(angles_set)
  {
    //angles to quaternions
    double t0 = std::cos(state_vector(angles_output_order+2) * 0.5);
    double t1 = std::sin(state_vector(angles_output_order+2) * 0.5);
    double t2 = std::cos(state_vector(angles_output_order) * 0.5);
    double t3 = std::sin(state_vector(angles_output_order) * 0.5);
    double t4 = std::cos(state_vector(angles_output_order+1) * 0.5);
    double t5 = std::sin(state_vector(angles_output_order+1) * 0.5);

    message.pose.pose.orientation.x = t0 * t3 * t4 - t1 * t2 * t5;
    message.pose.pose.orientation.y = t0 * t2 * t5 + t1 * t3 * t4;
    message.pose.pose.orientation.z = t1 * t2 * t4 - t0 * t3 * t5;
    message.pose.pose.orientation.w = t0 * t2 * t4 + t1 * t3 * t5;
  }
  if(velocity_set)
  {
    message.twist.twist.linear.x = state_vector(velocity_output_order);
```

```
      message.twist.twist.linear.y = state_vector(velocity_output_order+1);
      message.twist.twist.linear.z = state_vector(velocity_output_order+2);
    }
    if(angular_velocity_set)
    {
      message.twist.twist.angular.x = state_vector(angular_velocity_output_order);
      message.twist.twist.angular.y = state_vector(angular_velocity_output_order+1);
      message.twist.twist.angular.z = state_vector(angular_velocity_output_order+2);

    }

    odometry_publish.publish(message);
  }

  void cycle()
  {
    if(first)
    {
      state_vector = current_measurement_vector;
      first = false;
    }
    prediction();
    correction();
    if(pose_publish_set)
      publish_state_estimate_pose();
    if(odometry_publish_set)
      publish_state_estimate_odometry();
  }

//Callbacks -----------------------------------------------------------------------------
  void odometry_Callback(const nav_msgs::Odometry& msg)
  {
    if(position_set)
    {
      current_measurement_vector(position_output_order)   = msg.pose.pose.position.x;
      current_measurement_vector(position_output_order+1) = msg.pose.pose.position.y;
      current_measurement_vector(position_output_order+2) = msg.pose.pose.position.z;
    }
    if(velocity_set)
    {
      current_measurement_vector(velocity_output_order)   = msg.twist.twist.linear.x;
      current_measurement_vector(velocity_output_order+1) = msg.twist.twist.linear.y;
      current_measurement_vector(velocity_output_order+2) = msg.twist.twist.linear.z;
    }

    //for test odometry only
    /*if(odometry_set)
    {
```

```
        double range = sqrt(msg.pose.pose.position.x*msg.pose.pose.position.x +
msg.pose.pose.position.y*msg.pose.pose.position.y);
        double bearing = atan2(msg.pose.pose.position.y, msg.pose.pose.position.x);

        current_measurement_vector(odometry_output_order)   = range;
        current_measurement_vector(odometry_output_order+1) = bearing;
      }*/

      time = msg.header.stamp.toSec();
      setTimestep(time);

      cycle();
    }

    void imu_Callback(const sensor_msgs::Imu& msg)
    {
      if(acceleration_set)
      {
        current_measurement_vector(acceleration_output_order) = msg.linear_acceleration.x;
        current_measurement_vector(acceleration_output_order+1) = msg.linear_acceleration.y;
        current_measurement_vector(acceleration_output_order+2) = msg.linear_acceleration.z;
      }
      if(angles_set)
      {
        double qx = msg.orientation.x;
        double qy = msg.orientation.y;
        double qz = msg.orientation.z;
        double qw = msg.orientation.w;

        current_measurement_vector(angles_output_order)   = atan2(2*(qy*qz + qw*qx), qw*qw -
qx*qx - qy*qy + qz*qz);
        current_measurement_vector(angles_output_order+1) = asin(-2*(qx*qz - qw*qy));
        current_measurement_vector(angles_output_order+2) = atan2(2*(qx*qy + qw*qz), qw*qw +
qx*qx - qy*qy - qz*qz);
      }
      if(angular_velocity_set)
      {
        current_measurement_vector(angular_velocity_output_order)   = msg.angular_velocity.x;
        current_measurement_vector(angular_velocity_output_order+1) = msg.angular_velocity.y;
        current_measurement_vector(angular_velocity_output_order+2) = msg.angular_velocity.z;
      }
      time = msg.header.stamp.toSec();
      setTimestep(time);

      cycle();
    }
};
```

```
int main(int argc, char *argv[])
{
  ros::init(argc, argv, "kalman_filter_node");
  ros::NodeHandle node;

  kalman_filter kf = kalman_filter(3,3,2,2); // position estimation, position output, orientation
estimation, orientation_output
  kf.setOdometry_publisher(node, "chassis");
  kf.setWorld_frame("odom");

  ros::Subscriber odometry_subscriber = node.subscribe("/odom", 100,
&kalman_filter::odometry_Callback, &kf);
  ros::Subscriber imu_subscriber = node.subscribe("/trilobot/imu", 100,
&kalman_filter::imu_Callback, &kf);

  ros::spin();

  return 0;
}
```