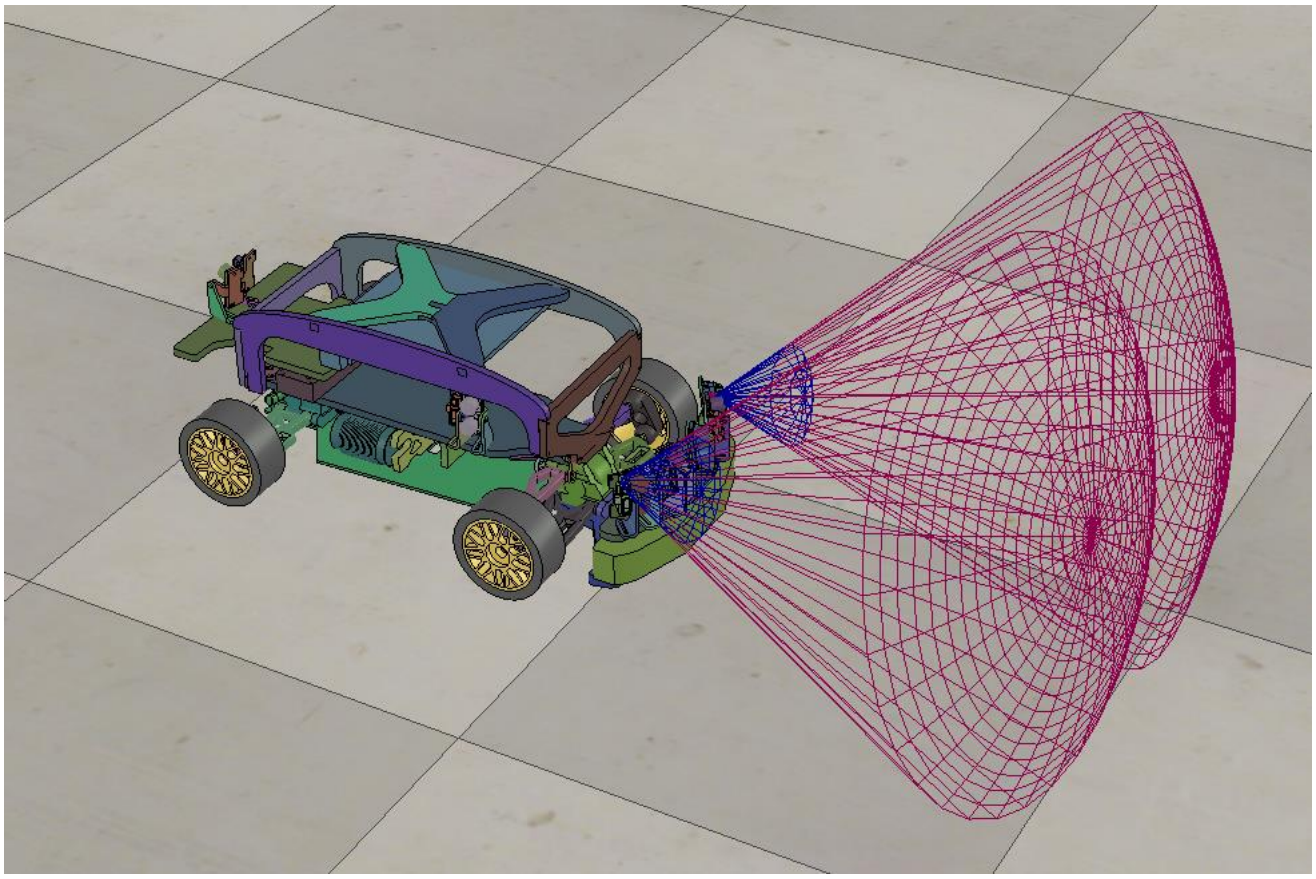
CHALMERS
UNIVERSITY OF TECHNOLOGY



Simulation av Gulliver

En virtuell robotmiljö för skalade autonoma fordon

Kandidatarbete inom Data- och Informationsteknik

Sebastian Dädeby
Adam Eriksson
Pontus Khosravi
Kristian Onsjö
Klas Sandell

Instutitionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2015

En virtuell robotmiljö för skalade autonoma fordon

S. Dädeby
A. Eriksson
P. Khosravi
K. Onsjö
K. Sandell

Examiner: Arne Linde

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Bilden på framsidan visar den virtuella modellen av Gulliver-bilen som tagits fram som del av projektet. Mer informarion finns på kapitel 5.2.

Department of Computer Science and Engineering
Göteborg, Sweden June 2015

Abstract

Autonomous cars are the next step in the transportation sector in the hopes of reducing the environmental impact and increase safety, but they are expensive to develop and test. In order to facilitate the development and reduce test costs, a virtual test environment has been developed and tested via implementation of an autonomous steering function.

In order to develop this virtual environment the simulation tool V-REP and the framework ROS were used, since both of these tools are developed for robot development. What has been developed is a virtual test environment as well as an interface for communication between simulation and framework.

The result of this project is a virtual test environment where filters and steering algorithms can be calibrated and tested for usage in a physical environment, as well as support for the possibility of basic network communication between multiple cars.

The virtual environment works well for testing the model. Simulator and framework have reached a functional level, but small errors occur which may demand further development to compensate for.

Sammanfattning

Autonoma bilar är nästa steg inom transport och fordonssektorn för att möjliggöra minskning av miljöpåverkan och öka säkerheten, men dessa är dyra att utveckla och testa. För att underlätta utvecklingen och reducera testkostnader har därför en virtuell testmiljö utvecklats och testats via implementation av en autonom styrfunktion.

För utveckling av den virtuella modellen användes simulationsverktyget V-REP samt ramverket ROS då båda dessa verktyg är utvecklade för robotutveckling. Det som utvecklats är en virtuell testmiljö samt ett gränssnitt för kommunikation mellan simulator och ramverk.

Detta projekt resulterade i en virtuell modell där filter och styralgoritmer kan kalibreras och testas för bruk i en fysisk miljö, samt stöd för möjligheter till grundläggande nätverkskommunikation mellan flera bilar.

Den virtuella miljön lämpar sig bra för att testa den önskade modellen. Simulator och ramverk har uppnått en funktionell nivå, men små fel uppstår som kan kräva vidareutveckling för att kompenseras för.

Nyckelord: autonom, bil, Gulliver, ROS, simulation, självkörande, testmiljö, testplattform, virtuell, V-REP

Förord

Ett stort tack till Elad Schiller och Thomas Petig för den expertis som de har bidragit med, både i genomförandet av projektet samt skrivning av rapporten. Vi vill också tacka Viktor Nilsson och Herman Fransson för den autonoma styrfunktion som har legat till grund för det som har byggts och implementerats i projektet. Även ett stort tack till Claes Ohlsson från Fackspråk för hans handledning med rapporten.

Förkortningslista/ordlista

Ankare	En statiskt placerad RCM.
Autonomt	Självstyrande.
Child script	Ett mindre program som utför en funktion i V-REP.
Färdriktning	Vinkel på bilen relativt referenskoordinatsystemet.
Gulliver-bilen	Den fysiska bilen.
Gulliver-modellen	Den modellerade bilen.
IMU	Inertial Measurement Unit.
Joystick	En standard XBOX-kontroll.
Level-of-Service	Kvalitet på nätverkskommunikation.
Meddelande	Strukturen för data som skickas.
Motorregulator	Reglerar hastighet och styrvinkel på bilen.
Noder	Tar emot, behandlar och skickar data i ROS.
RCM	Ranging and Communication Module.
ROS	Ramverk för robothantering
ROS-paket	Programstruktur, innehåller noder.
Publisher	Publicerar (skickar) data, del av noder.
Statisk karta	En vägbeskrivning för den autonoma styrfunktionen.
Styrvinkel	Vinkel på bilens hjul.
Subscriber	Prenumererar på (tar emot) data, del av noder.
Topic	Kanalen som data skickas över.
V-REP	Virtual Robot Experimentation Platform.

Innehåll

1 Inledning

Bilen i alla dess former och storlekar är ett av världens mest använda transportfordon. Miljoner bilar körs varje dag, bara i Sverige, och det ser likadant ut i större delen av världen. Då varje bil kräver en förare finns det ett stort intresse i att låta bilarna köra autonomt och på så sätt eliminera den mänskliga faktorn. Fördelarna med detta är enorma, inte bara en besparing i arbetskraft utan också en förhöjd säkerhet, potentiellt mindre miljöpåverkan samt på sikt besparing av utrymme i storstäder då autonoma bilar klarar av att köra på mindre ytor. Det följande avsnittet behandlar varför projektet är relevant, relaterade arbeten, projektets problembeskrivning och dess avgränsningar.

1.1 Bakgrund

Förutom miljöproblem bidrar bilar också till ett socialt och ekonomiskt problem. Enligt siffror från 2004 dör 1,2 miljoner människor varje år i trafikrelaterade olyckor och hela 20 till 50 miljoner människor blir skadade till följd av trafikolyckor. Dessa siffror har ökat och förväntas fortsätta öka drastiskt de kommande åren. Idag står trafikrelaterade olyckor för ungefär 2,1 procent av alla dödsfall i världen. Detta gör trafikrelaterade olyckor till den elfte vanligaste dödsorsaken i världen. Dessutom kostar alla dessa trafikolyckor stora summor pengar för samhället, beroende på låg- eller höginkomstland, 1 respektive 2 procent av bruttonationalprodukten. [1]

För att kunna minska antalet olyckor måste man minimera problemet, vilket i de flesta sammanhang är människan. Därför bedrivs forskning kring självstyrande bilar, autonoma bilar, som kan fatta egna beslut och köra distanser automatiskt utan människans närvaro. Därigenom hoppas man kunna underlätta användandet av bilar och minska antalet trafikolyckor. [2] Förutom att undvika olyckor finns det också förhoppningar om att autonoma bilar ska underlätta trafiksituationen i många städer och förebygga trafikstockningar.

Genom koordinerad körning mellan fordon, med hjälp av ett trådlöst nätverk, kan man skapa konvojer och undvika onödiga stopp. Därigenom hoppas man kunna minska miljöpåverkan, minska olycksrisken och spara miljonbelopp varje år. [3]

Utveckling och testning av autonoma fordon är kostsamt och tar mycket tid. I syfte att underlätta utveckling startades projektet Gulliver på Institutionen för Data- och Informationsteknik på Chalmers. Gulliver är tänkt att fungera som en testplattform för autonoma system, där funktioner kan testas i stor skala med hjälp av billiga miniatyrfordon. Ett Gulliver-fordon kostar cirka 2000€, vilket är mycket billigare än ett fullskaligt testfordon som kan kosta så mycket som 100,000€. [4]

1.2 Syfte

Syftet med projektet är att underlätta utvecklingen av autonoma styrsystem genom att erbjuda en virtuell testmiljö där parametrar för filter och regulatorer kan testat och optimeras. Genom att använda en sådan testmiljö så kan man minska kostanderna associerade med detta.

1.3 Problemformulering

Problemet för detta projekt är uppdelat i två delar. Den första delen är att modellera den fysiska Gulliver-bilen virtuellt. Modellen skall vara realistisk och ha samma egenskaper som den verkliga Gulliver-bilen. Modellen skall också köra i en liknande inomhusmiljö för att förenkla övergången till en fysisk implementation.
Den andra delen är att skapa en passande mjukvaruarkitektur för implementation av ett redan utvecklat autonomt styrsystem. Denna mjukvaruarkitektur skall vara kompatibel med den fysiska Gulliver-bilen och den skapade Gulliver-modellen.
Följande avsnitt kommer att beskriva varje problem mer detaljerat.

1.3.1 Den virtuella simulationsmiljön

Den virtuella simulationsmiljön skapades helt från grunden. För att återskapa den fysiska miljön virtuellt behövs följande delar:

- En virtuell modell av den fysiska Gulliver-bilen.
- Vägbanor med vägpunkter för bilen att följa.
- Mätningar av bilens acceleration, färdriktning och svängradie i tre dimensioner.
- En motorregulator.
- Mätningar av avstånd till närliggande bilar och hinder som finns i bilens färdriktning.
- Styrning med hjälp av joystick.
- Ankare för positionsmätning.
- Loggning av relevant data.

Bilen skall bete sig precis som Gulliver-bilen. Bilen kommer inledningsvis att styras med hjälp av en xbox-kontroller för test av modellens motorregulator, för att sedan implementera en autonom styrning.

1.3.2 Mjukvara för autonomt styrsystem

För att skapa en passande mjukvaruarkitektur behövs följande delar:

- Lagring av en statisk karta, kartan hämtas från den virtuella modellen.
- Integration av positionsuppskattning, bilen ska veta ungefär var den är på den statiska kartan.
- Integration av vägplanering, bilen planerar hur den ska ta sig från punkt A till punkt B.
- Kollisionsundvikning, avgöra om en bil är för nära ett hinder och bromsa om så är fallet.

- Förberedelse för framtida implementation av kommunikation mellan bilar.
- Kompatibelt gränssnitt mot den virtuella och fysiska modellen.
- Loggning av relevant data.

De listade punkterna ger en bra grund för ett autonomt system för utveckling och testning. Vissa av dessa delar är redan utvecklade av ett tidigare projekt och kommer användas, samt vid behov modifieras. Gränssnittet ska fungera både mot den virtuella miljön och den fysiska miljön. Modellen skall även ha möjligheten att kunna hantera flera bilar samtidigt i simulationen för att skapa konvojer.

För att flera bilar skall kunna köra tillsammans autonomt på ett effektivt sätt, räcker det inte bara med bilarnas lokala lösningar. Det krävs även att bilarna kommunicerar med varandra så att information kan skickas mellan dem, exempelvis för att minska bromstider och kunna anpassa sin rutt baserat på trafiksituationen.

1.4 Relaterade arbeten

Projektet bygger på ett tidigare arbete, skrivet av Herman Fransson och Viktor Nilsson, vars syfte var att implementera autonom styrning av Gulliver-bilen. Arbetet var en del av Gulliver-projektet. [5]

Parallellt med detta projekt utförs ett annat kandidatarbete [6], vars uppgift är att skapa en mjukvaruarkitektur för att simulera autonoma fordon som kommunicerar över valfritt nätverksprotokoll. Projektet förbereder integration för att få säker kommunikation mellan flera bilar, därför har det varit ett pågående samarbete mellan grupperna.

Ett annat relaterat arbete är Bridging Physical and Digital Traffic System Simulations with the Gulliver Test-Bed [7] som behandlar en länkning mellan en virtuell och en fysisk miljö för Gulliver-plattformen. Detta för att skapa en gemensam testmiljö där både fysiska och virtuella Gulliver-bilar ingår.

1.5 Metodöversikt

Projektet inleddes med en inlärningsfas där material och verktyg som fanns tillgängligt studerades. En stor del av denna fas var att förstå tidigare projekt och hur verktygen ska användas. Detta följdes av en implementationsfas där varje del av problembeskrivningen implementerades och testning utfördes i den mån möjligt vid brist av data från fysiska tester. Implementationsfasen delades upp i flera mindre etapper så att tidsplaneringen enklare kunde följas och för att underlätta testning. I denna fas fanns dock en hel del flexibilitet för att kunna åtgärda eventuella problem som uppstår. Implementationsfasen följdes av en test- och resultatfas där projektet testades och resultat avgavs för denna rapport. Genom hela projektet användes den agila arbetsmetoden scrumban [8].

1.6 Avgränsningar

Projektet avgränsas genom att den virtuella modellen endast innehåller två enklare statiska kartor; en cirkel och en karta som ser ut som en åtta. Mjukvaruarkitekturen testas bara i den virtuella modellen. Komponenterna på Gulliver-bilen kommer inte utvärderas på något sätt. Dessutom kommer det autonoma styrsystemet endast testas på en instans av den virtuella modellen.

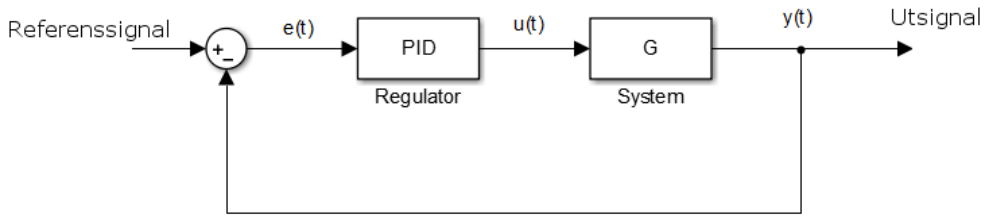
2 Teori

Detta kapitel förklarar de tekniker som används genom att beskriva grundläggande principer och ger även matematiskt stöd. Dessa delar är viktiga komponenter i mjukvaran för projektet.

2.1 PID-regulator

Det finns flera olika reglertekniker men de har alla samma uppgift, att styra systemet på önskat sätt genom att bestämma storleken på olika styrsignaler. Den vanligaste regulatorn är den så kallade PID-regulatorn *(proportionell, integrerande och deriverande)*. Det är dock vanligt att man inte använder den deriverande delen då det är svårt att derivera med avseende på brusiga mätsignaler. [9]

En PID-regulator fungerar rent matematiskt enligt ekvation (1) men för att få en bättre förståelse används figur 1. Först beräknas reflerfelet, som är referenssignalen subtraherat utsignalen. Felet som uppstår regleras sedan i PID-regulatorn och skickas till systemet. Systemets utsignal är nu kompenserat för en del av felet beroende på vilken skalning som används. Detta system har också en återkoppling vilket leder till att processen itereras för varje ny utsignal.



Figur 1: Ett enkelt reglersystem med återkoppling. Lägg särskilt märke till var i reglersystemet regulatorn används.

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

Där $K_p e(t)$ är den proportionella delen, $K_i \int_0^t e(\tau)d\tau$ är den integrerande delen och $K_d \frac{d}{dt} e(t)$ är den deriverande delen av PID-regulatorn. K_p, K_i och K_d är skalningskonstanter som sätts för att systemet ska bli så optimerat som möjligt. Det finns flera olika optimeringsmetoder, men ett sätt som enkelt tillämpas med en simulerad testmiljö är att testa sig fram till bästa värde. [9]

2.2 Kalmanfilter

Kalmanfilter används i stor utsträckning för robotnavigering och systemintegration. För att en robot ska kunna fungera autonomt måste den veta var den är. Exakt lokalisering är en viktig förutsättning för framgångsrik navigering i storskaliga miljöer, speciellt när globala modeller används, såsom kartor, ritningar, topologiska beskrivningar och CAD-modeller. [10]

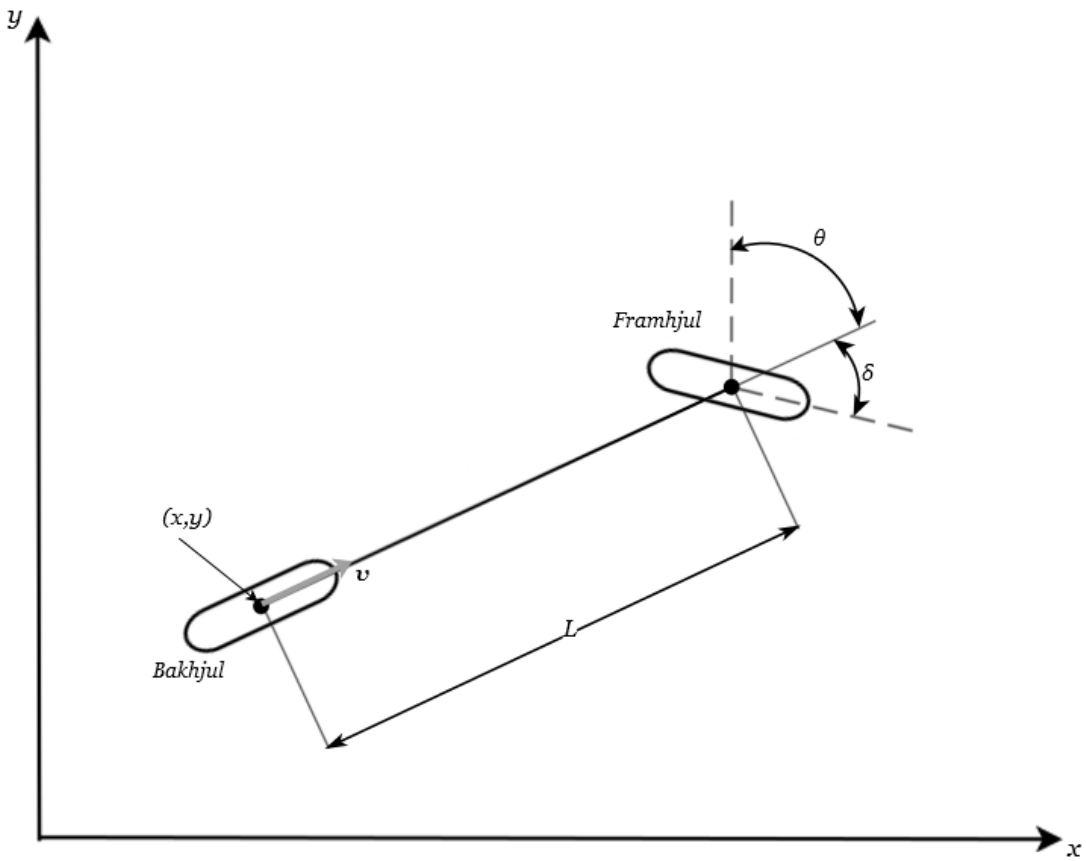
Kalmanfiltret uppfanns på 1950-talet av Rudolf Emil Kalman och är en matematisk algoritm för att approximera tillståndet hos linjära system. Filtret är en typ av gaussfilter, en rekursiv tillståndsestimator. Algoritmen använder metoder från såväl reglerteknik som sannolikhetslära. Principen är att systemets momentantillstånd beskrivs som normalfördelningar, som representerar tilltron till modellens överensstämmelse med verkligheten. Större varians betyder större osäkerhet. Systemets tillstånd är alltså inte känt explicit, men kan approximeras till normalfördelningarnas väntevärden. Att styra ett system på detta sätt är användbart när systemet har många frihetsgrader och en inneboende slumpmässighet. När osäkerheter för det nuvarande tillståndet är kända kan de kompenseras för i styrningen av systemet och högre precision kan uppnås. [11]

När momentantillståndet ska uppdateras i Kalmanfiltret används det tidigare momentantillståndet som insignal, tillsammans med ny data. Baserat på tidigare tillstånd, mätningar och kunskap om systemets egenskaper kan ett nytt momentantillstånd beräknas. [11]

Algoritmen kan också hantera osäkerheter, exempelvis brus hos sensorer eller dålig precision hos motorer, genom att de inkluderas i modellen, förutsatt att de är kända eller kan approximeras. [10], [11]

2.3 Cykelmodell

Ackermannstyrning är ett geometriskt arrangemang av kopplingar i styrningen av en bil eller annat fordon som syftar till att lösa problemet med att hjul på insidan och utsidan av en sväng behöver följa cirklar med olika radie. Cykelmodellen är en förenklad matematisk modell för att modellera ett bakhjulsdrivet fordon med ackermannstyrning. Modellen approximerar fordonet som en cykel med samma avstånd mellan fram- och bakhjul, placerad mitt i mellan hjulen. Det finns två olika typer av modellen, en dynamisk som används för att beräkna dynamiska krafter i systemet, och en kinematisk som används för att beräkna rörelser i systemet. Detta projekt använder en kinematisk modell. Figur 2 visar hur en kinematisk modell kan utformas. En tillståndsmodell för systemet ställs upp enligt ekvation (2). $\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix}$ är nästa momentanvärde. Δd är $\int_{k}^{k+1} v$. [12]



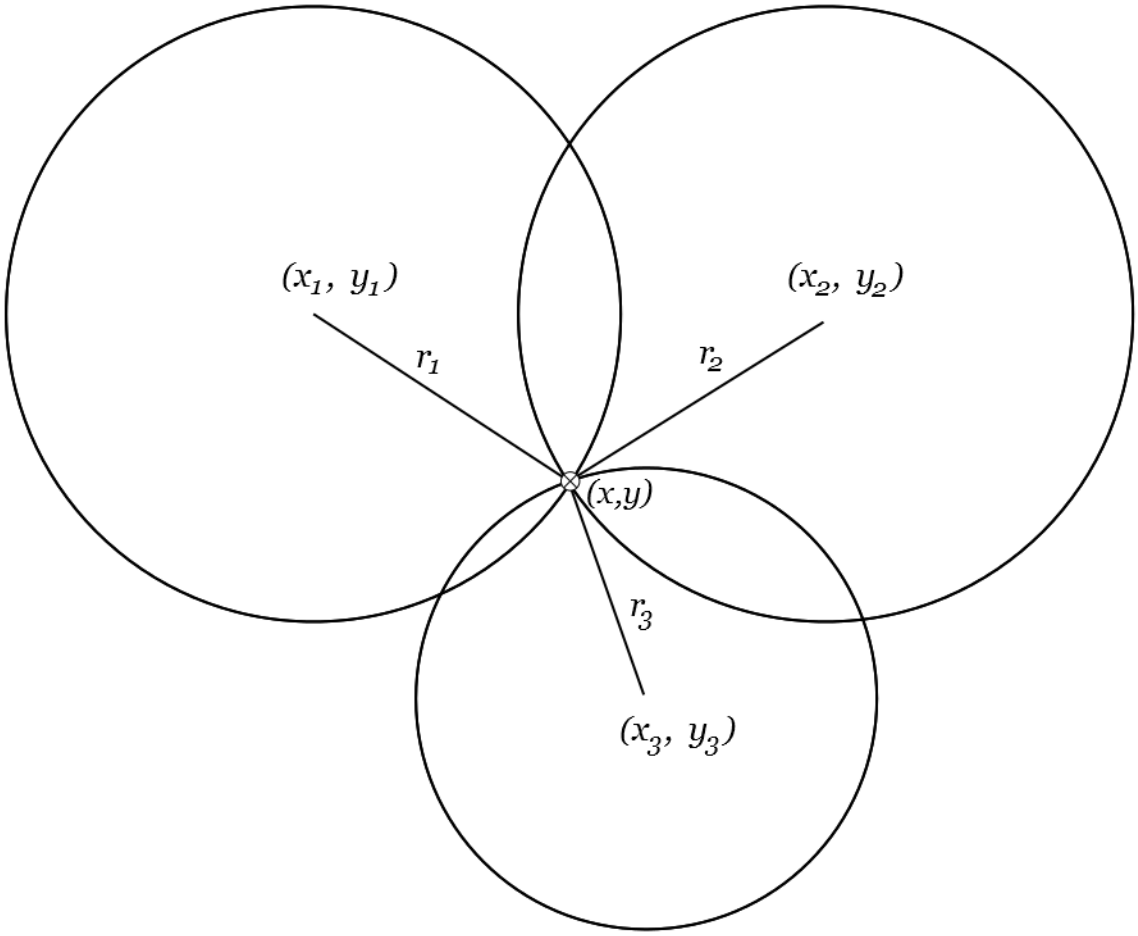
Figur 2: Cykelmodell med längd L, vinkel relativt y-axeln θ, bakhjulets position (x,y) samt hastighet v i färdriktningen.

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + \Delta d \cdot sin(\theta_{k+1}) \\ y_k + \Delta d \cdot cos(\theta_{k+1}) \\ \theta_k + \frac{\Delta d + tan(\delta)}{L} \end{bmatrix} \tag{2}$$

2.4 Trilateration

Trilateration är en metod för att bestämma absoluta eller relativa positioner med hjälp av de geometriska egenskaperna hos cirklar, sfärer och trianglar. Till skillnad från triangulering, som används för GPS, behöver inga vinklar mätas. På grund av de svåra beräkningar som måste göras har trilateration ofta avfärdats till fördel för triangulering. Modern teknik möjliggör däremot en tillräckligt snabb och säker mätnings- och beräkningskapacitet för att kunna tillämpa metoden på mindre skala för experimentella tester. [13]

För positionsbestämning i två dimensioner kan en position beskrivas som skärningspunkten för tre cirklar med centrum på olika positioner, så som illustreras i figur 3. Genom att mäta avståndet till dessa positioner, och därmed även radien på cirklarna, kan den relativa positionen fastställas. Om centrumpunkterna för cirklarna har kända koordinater kan en absolut position fastställas genom ekvationssystem (4) som är en vidareutveckling av ekvationsystem (3). [13]



Figur 3: trilaterationsproblemet utgörs av att hitta positionen (x,y), givet avståndet till de tre punkterna (x_1, y_1), (x_2, y_2) och (x_3, y_3).

$$\begin{cases} (x-x_1)^2 + (y-y_1)^2 &= r_1^2 \\ (x-x_2)^2 + (y-y_2)^2 &= r_2^2 \\ (x-x_3)^2 + (y-y_3)^2 &= r_3^2 \end{cases} \tag{3}$$

$$\begin{cases} (x-x_1)^2 + (y-y_1)^2 &= r_1^2 \\ x(x_2-x_1) + (y_2-y_1) &= \frac{r_1^2 - r_2^2 + a^2}{2} \\ x(x_3-x_1) + (y_3-y_1) &= \frac{r_1^2 - r_3^2 + b^2}{2} \end{cases} \tag{4}$$

$$a = \sqrt{(x_1-x_2)^2 + (y_1-y_2)^2}, b = \sqrt{(x_1-x_3)^2 + (y_1-y_3)^2}$$

3 Tekniska förutsättningar

Följande avsnitt beskriver delvis de verktyg som används under projektet, samt de sensorer som finns på Gulliver-bilen och relevanta paket som fåtts från diskussion med H. Fransson och V. Nilsson [5].

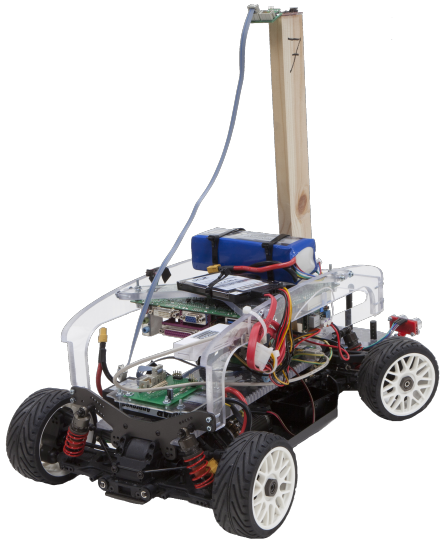
3.1 Robot Operating System

Robot Operating System (ROS) är ett ramverk för operativsystemet Ubuntu för att skriva och implementera mjukvara för robotar. Det är en samling verktyg, bibliotek och konventioner för att underlätta skapandet av komplexa och robusta beteendemönster hos robotar över en stor samling robotplattformar. Ramverket skapades på grund av svårigheter med att skriva robust och allmän robotmjukvara. Det som för en människa kan verka trivialt kan för en robot vara otroligt komplext och att implementera detta kan kräva resurser som en enskild forskningsgrupp inte kan bidra med. [14]

ROS skapades år 2007 som ett kollektivt samarbete mellan forskningsgrupper och privata utvecklare för att vidareutveckla robotprogramvara med gemensamma resurser och resultat. Initiativet har växt till att idag innefatta ett stor mängd paket som näst intill helt modulärt kan kombineras för att skräddarsy programvara. [14]

ROS-ramverket är uppbyggt av paket, som är en organisationsenhet. Syftet med paketstrukturen är att tillhandahålla lättillgänglig funktionalitet så att programvaran kan återanvändas. Ett paket kan innehålla exempelvis bibliotek, script eller ROS-noder, skrivna i programmeringsspråken C++ eller Python [15]. En ROS-nod är en process som utför beräkningar eller översätter meddelanden. Ett styrsystem består ofta av ett flertal noder som tillsammans styr roboten [16].

Noder kommunicerar med varandra med meddelanden, en enkel datastruktur bestående av typade fält. En nod kan bara ta emot ett meddelande av en fördefinierad typ, och bidrar därmed till ökad kommunikationssäkerhet. Meddelanden skickas mellan noder över namngivna kommunikationsbussar kallade *topics* [17]. En nod bryr sig normalt sett inte om vilken annan nod den kommunicerar med, istället prenumererar (*subscribe*) en nod på en topic som skickar önskad meddelandetyp. Noder som tillhandahåller denna meddelandetyp publicerar (*publish*) meddelanden till dessa topic. Det finns ingen gräns för hur många publishers eller subscribers en topic kan ha. Om det är viktigt att två noder kommunicerar direkt med varandra, exempelvis när en process ska anropas hos en annan nod, används *services*. En service fungerar som en funktion med inparametrar som returnerar ett värde baserat på inparametrarna. [18]



Figur 4: Gulliver-bilen

3.2 Fysiska komponenter på Gulliver-bilen

Gulliver-bilen är en miniatyrbil framtagen i Gulliverprojektet. Den fungerar som modulär testplattform för framtagning av autonoma styrsystem. Ombord på bilen finns en dator som kan köra styrsystemet, och sensorer för att känna av omgivningen. Figur 4 visar hur bilen ser ut. Underkapitlet beskriver de komponenter på den fysiska Gulliver-bilen som modellerats i projektet.

3.2.1 Ranging and Communication Module (RCM)

På Gulliver-bilen finns en enhet för de mätningar som krävs för att kunna utföra trilateration. Enheten är en *Ranging and Communication Module* (RCM) av modell PulsON 410 från Time Domain [5]. En RCM-enhet kan kommunicera med andra RCM enheter genom radiofrekvens-teknik och används med fördel i högreflektiva miljöer där GPS har problem, exempelvis inomhus. Enheterna används främst för att mäta avstånd, men kan också användas för enklare kommunikation [19], något som inte utnyttjas i denna tillämpning. Genom att positionera tre sändare med sina relativa positioner kända skapas ett lokalt trilaterationssystem för den fjärde sändaren som finns monterad på Gulliver-bilen [5].

3.2.2 Inertial Measurement Unit (IMU)

På Gulliver-bilen finns en enhet för att mäta fysiska krafter, en *Inertial Measurement Unit* (IMU) [5]. IMU:n består av två skilda enheter, en accelerationsmätare och ett gyroskop, men brukar betraktas som ett mätinstrument. Accelerationsmätaren består av tre linjära accelerometrar som mäter acceleration längs axlarna i ett kartesiskt koordinatsystem, samt tre magnetfältsensorer längs samma axlar. Med magnetfältsensorerna kan IMU:n

mäta jordens magnetfält och på så sätt användas som kompass. Gyroskopet kan mäta vinkelhastigheten för rotation kring koordinataxlarna, vilket gör det möjligt att med derivering och integration avgöra om enheten roterar eller lutar relativt ett referensplan. [20], [21]

3.2.3 Ultraljudssensor

I fronten på Gulliver-bilen finns två ultraljudssensorer av typen SRF08 monterade [5]. Sensorerna används för att upptäcka hinder framför bilen och kan också mäta avstånd till objekt framför bilen. Detta görs genom att mäta tiden mellan att en puls skickas ut och att ekot kommer tillbaka från det reflekterande objektet. [22]

3.3 Paket från tidigare arbeten

I detta underkapitel beskrivs de viktigaste ROS-paketen som skapats av H. Fransson och V. Nilsson för Gulliver-bilen [5]. Endast de paket som används för autonom styrning beskrivs, de övriga paketen berör hårdvara och används enbart för att bestämma struktur på meddelanden som ska användas i mjukvaruarkitekturen.

3.3.1 `estimate_position`

Detta paket försöker uppskatta positionen på bilen genom att samla data från de tillgängliga sensorerna. Paketet hämtar data från bilens motorregulator och från IMU:n. Utifrån datan uppskattas en position med hjälp av cykelmodellen beskriven i kapitel 2.3. Denna position kombineras i ett Kalmanfilter med den position som RCM-systemet anger. Resultatet blir den slutgiltiga position som paketet tror att bilen har. Denna process upprepas varje gång det finns ny data att hämta från sensorerna. [5]

3.3.2 `autopilot`

Paketet hämtar positionen från `estimate_position` efter varje uppdatering och jämför mot en vägpunkt. Om målet inte är nått justeras styrvinkeln och skickas till motorregulatorn. Justeringen av vinkeln beräknas som skillnaden mellan nuvarande färdriktning och vinkeln mellan bilens bakaxel och riktningen till målpositionen. Denna skillnad används av en PID-regulator för att beräkna och sätta en ny styrvinkel. [5]

3.4 Virtual Robotics Experimentation Platform

Virtual Robotics Experimentation Platform (V-REP) är ett grafiskt simulationsverktyg utvecklat av Coppelia Robotics och är verktyget som används för att utveckla den virtuella modellen. V-REP är plattformsoberoende och objekt läggs ihop till en enda fil, så kallade scener, vilket gör programmet flexibelt att arbeta med då filen inte behöver kompileras om för att kunna köra simulationen på andra datorer. Programmet är designat runt en mångsidig arkitektur, med flera relativt oberoende funktionaliteter som kan slås på eller av utefter behov. Detta gör det möjligt att genomföra tunga simulationer med relativt lågt prestandakrav, eftersom funktioner som inte används stängs av. [23]

V-REP har ett välutvecklat stöd för samarbete med ROS. Dessutom efterliknar V-REP även den fysiska verkligheten väl, vilket gör det till ett bra simulationsverktyg inför tester i verkligheten. [23]

4 Metod

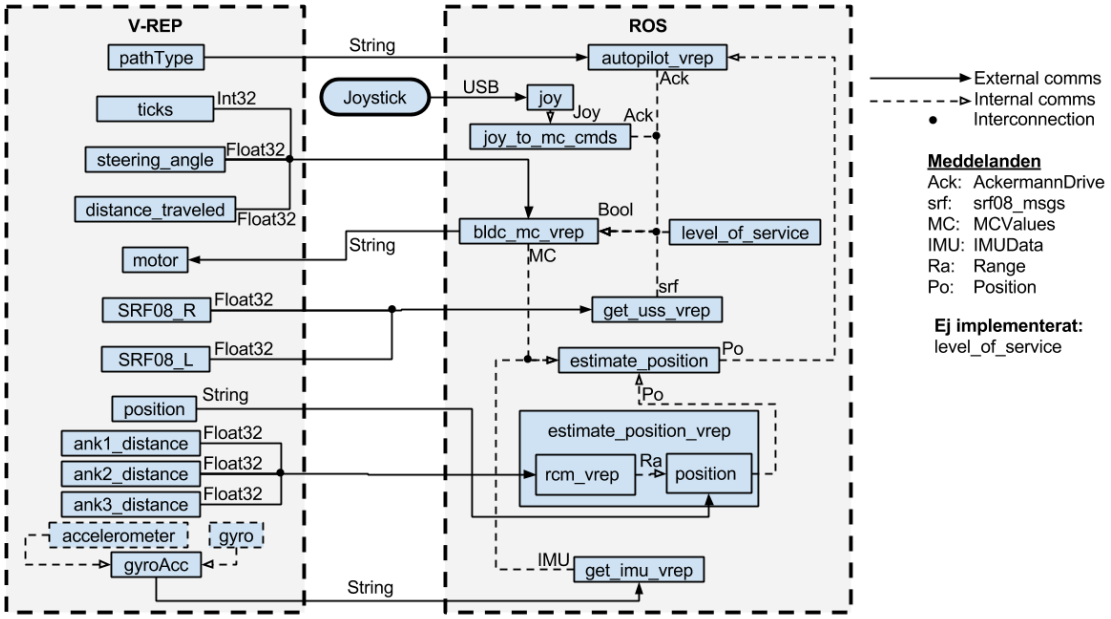
Detta kapitel behandlar i detalj den metod som valdes under projektets gång för att lösa problemformuleringen och tekniska val förklaras samt motiveras. Tillvägagångssätt för testning av systemet tas också upp.

4.1 Mjukvaruarkitektur

En mjukvaruarkitektur beskriver hur alla delar beror på varandra och hur kommunikation sker mellan dessa. Ramverket ROS Indigo används för att skapa mjukvaruarkitekturen och körs på Ubuntu 14.04. Arkitekturen vidareutvecklas från den redan existerande arkitekturen för att behålla kompatibilitet med den fysiska modellen. ROS använder sig av en paketstruktur, som beskrivs i kapitel 3.1, vilket gör ramverket flexibelt och modulärt [24]. Kommunikationsstrukturen *kondenserad publisher/subscriber* används, vilket innebär att flera publisher och subscriber finns i en fil, och minskar antalet filer och noder som körs i ROS. Alla topics i arkitekturen utgör gränssnittet mellan ROS och den virtuella eller fysiska modellen.

4.1.1 Följd mjukvaruarkitektur

Figur 5 visar hur automation- och simulationsdelen kommunicerar med varandra och vilket format datan har. Detta flödesschema var grunden som projektet utgick från när mjukvaruarkitekturen skulle byggas. Figuren visar vilka topics som startas i V-REP och vilka paket och noder som finns i ROS. Paket- och nodnamn är desamma i de flesta av paketen i ROS, med undantag för `estimate_position_vrep` där två noder startas. Utöver detta existerar också en fysisk joystick, denna symboliseras i flödesschemat som *Joystick*.
Detta projekt bygger på tidigare projekt och diskussion med H. Fransson och V. Nilsson [5] och därför byggdes mjukvaruarkitekturen runt de redan existerande paketen.



Figur 5: Flödesschema som beskriver önskad mjuvaruarkitektur

Pilarna i figur 5 beskriver kommunikation och namnet är meddelandetypen. Meddelanden från V-REP görs om till de angivna meddelandetyperna för att behålla kompatibiliteten med den fysiska modellen. Noderna, kommunikationen och de meddelandetyper som används i arkitekturen förklaras i kommande stycken. Vidare finns fullständig kod för paketen i Appendix A.

`estimate_position_vrep`
Noden `rcm_vrep` tar emot data från V-REP innehållandes avståndet från bilen till aktuellt ankare. Ett meddelande av typen `Range` skapas som innehåller denna data och även ankarets ID. Resterande fält, felestimering och tidstämpel, används inte. Meddelandet skickas sen till noden `Position`.

`Position`-noden använder tre `Range`-meddelanden, ett för varje ankare, för att beräkna bilens uppskattade position genom trilateration. Den beräknade positionen packas i ett meddelande av typen `Position` som innehåller x- och y-värde samt färdriktning och skickas sedan till `estimate_position`. `Position`-noden tar även emot bilens absoluta position som skickas i form av en packad sträng från V-REP. Noden packar upp strängen och får
x-, y- och z-värden för bilens absoluta position. Detta gör det möjligt att istället för att använda trilateration för positionsuppskattning direkt skicka bilens faktiska position till `estimate_position` och därmed undvika möjliga problem som kan uppstå genom beräkning av sensordata vilket underlättar tester av positionsestimeringen. Dessutom tillåter detta möjligheter för att jämföra bilens beräknade position mot bilens sanna position.

`get_imu_vrep`

Paketet tar emot en packad sträng från V-REP innehållandes x- och y-värde från accelerometern och z-värde från gyroskopet. Paketet packar upp strängen och skapar ett meddelande av typen `IMUData` och sätter respektive fält till värdet, resterande fält används inte. Meddelandet skickas sedan till `estimate_position`.

`estimate_position`
Paketet tar emot data från IMU:n, en position estimerad från trilateration och data från motorregulatorn. Paketet använder sedan ett Kalmanfilter för att beräkna en ny position som bör stämma bättre överens med den faktiska positionen. Den nya positionen skickas till `autopilot_vrep` i ett meddelande av typen `Position`.

`autopilot_vrep`
Detta är bilens navigationspaket. Paketet tar emot vilken karta som används från V-REP och kan därefter läsa in de aktuella vägpunkterna från en *csv*-fil. Paketet får även bilens uppskattade position från `estimate_position` och kan då avgöra vilken styrvinkel och hastighet bilen ska ha för att ta sig till nästa vägpunkt. Ett meddelande av typen `AckermannDrive` skapas innehållande styrvinkeln och hastigheten, resterande fält används inte. Meddelandet skickas sedan till `bldc_mc_vrep`.

`level_of_service`
Detta paket är inte implementerat. Paketet ska avgöra om nätverkskommunikationen har hög level of service. Paketet skickar ett meddelande av typen *bool* till `bldc_mc_vrep`. En bool har antingen värdet 1 eller 0, där 1 anger en hög level of service.

4.2 Positionsuppskattning

För att implementera ett sätt att uppskatta en position används det redan existerade ROS-paketet `estimate_position` som filtrerar och beräknar uppskattad position och skickar vidare detta till bilens vägplaneringsalgoritm [5] och finns beskrivet i 3.3.1.

4.3 Vägplanering

Om bilen skall köra autonomt och köra efter en specifik väg behövs en algoritm som kan styra bilen genom att data skickas från algoritmen och sensordata fås från positionsbestämningen. Det ROS-paket som innehåller vägplaneringen fås från [5]. Paketet heter `autopilot` och beskrivs i kapitel 3.3.2. Detta paket modifieras sedan för projektets användningsområde då `autopilot`:s statiska karta inte sammanfaller med projektets testbanor. En statisk karta är en serie koordinater som inte uppdateras med nya värden. Genom att tillåta en automatisk import av statiska kartor från exporterade koordinatsystem i V-REP möjliggörs skapandet av vilken statisk karta som helst bara kartan är modellerad i V-REP och stöd anges för den i paketet. På grund av denna modifikation skapas ett nytt paket som kallas `autopilot_vrep` som tillåter flera testkartor och enkelt kan utvecklas vidare för att inkludera ytterligare kartor.

4.4 Kollisionsundvikning

Den kollisionsundvikning som används är väldigt simpel och bygger på diskussion med H. Fransson och V. Nilsson [5]. Om ultraljudssensorerna upptäcker ett hinder kommer en signal skickas till motorregulatorn som sätter önskad hastighet till noll och bilen kommer då bromsa tills den står still. Kollisionsundvikningen har högsta prioritet vilket gör att andra system, som till exempel autopiloten, stängs av.

4.5 Konvojer och kommunikationssystem

Konvojkörning möjliggörs genom att läsa data från ultraljudssensorerna på bilen. Genom en adaptiv farthållning håller bilarna ett långsamt oscillerande avstånd från varandra. Denna oscillation går att, genom mer komplexa styralgoritmer och filtrering av data, minimeras för att uppnå ett konstant jämnt avstånd mellan bilarna.

Då konvojer innefattar flera bilar krävs även en metod för att på ett så dynamiskt sätt som möjligt möjliggöra för att lägga till önskat antal bilar i en simulation. Detta uppnås genom att använda *namespaces* i ROS, vilket är en isolerad miljö inuti ROS där noder och topics kan startas utan konflikt med noder och topics som ligger utanför namespace. Genom detta går det att starta isolerade instanser av varje bil som kan kommunicera med V-REP ostört av andra instanser av samma noder. Dessvärre så fungerar detta endast om varje instans av bilen använder unika topics i V-REP, till vilket programmet inte erbjuder någon dynamisk lösning.

Konvojkörning optimeras via implementation av systemet från [6] och namespaces i ROS. Systemet anger om det är hög eller låg *level of service*.

Level of service anger kvaliteten på nätverkskommunkationen. Algoritmen som beskrivs i [25] kan användas för att bestämma level of service. Denna information kan sedan användas för att köra bilarna på olika sätt. Exempelvis kan bilarna via nätverkskommunikation få information om resterande bilar, som kan inkludera position och hastighet. Alla bilar bestämmer sedan en säker rutt för de olika graderna av level of service. [26]

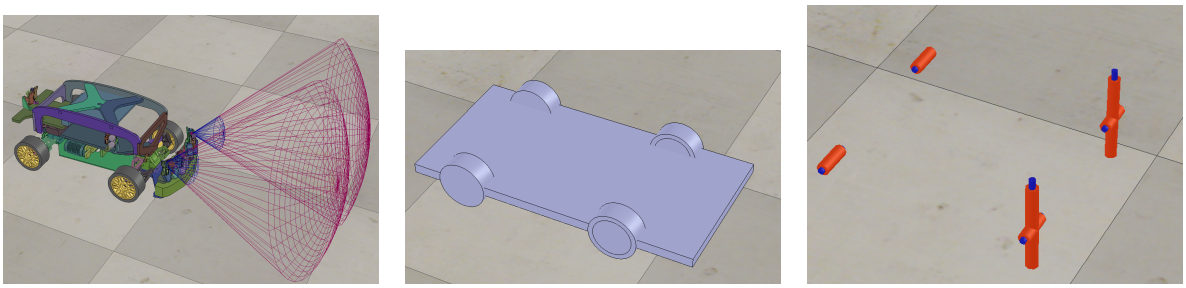
4.6 Virtuella modellen

Det program som används för att modellera Gulliver-bilen och miljöerna är V-REP, som beskrivs i avsnitt 3.4, och den aktuella versionen som används är 3.2.1. Kapitlet tar även upp hur Gulliver-bilens olika komponenter integreras samt hur script används.

4.6.1 Modellering av Gulliver-bilen

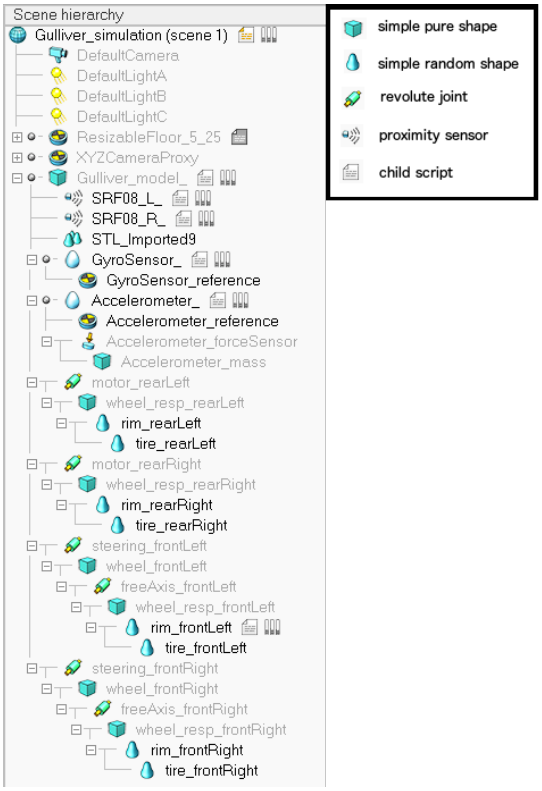
Gulliver-bilens skal implementerades genom att ta en existerande CAD-fil av Gulliver-bilen, som skapats av Chalmers studenter, och konvertera den till en läsbar fil för V-REP. Ur den konverterade CAD-filen importerades varje del av skalet, 95 stycken, direkt till V-REP. Delarna är av typen *simple random shape* vilka har ett högt polygonantal som gör dem beräkningstunga [27]. Dessa delar är bara statiska objekt i V-REP och finns

främst som referens till placering av sensorer och av estetiska skäl. V-REP använder sig av 16 lager där de åtta första är synliga och de åtta sista är osynliga som standard. Lager används för att gruppera objekt för enklare hantering av större modeller då man enkelt kan välja vilka grupper av objekt som ska visas. De importerade statiska objekten ligger i det enda synliga lagret och tillsammans utgör de den synliga delen av bilen. I två osynliga lager ligger de objekt som utgör den faktiska bilen och ger bilen de egenskaperna den fysiska Gulliver-bilen har. De tre lagrerna visas i figur 6



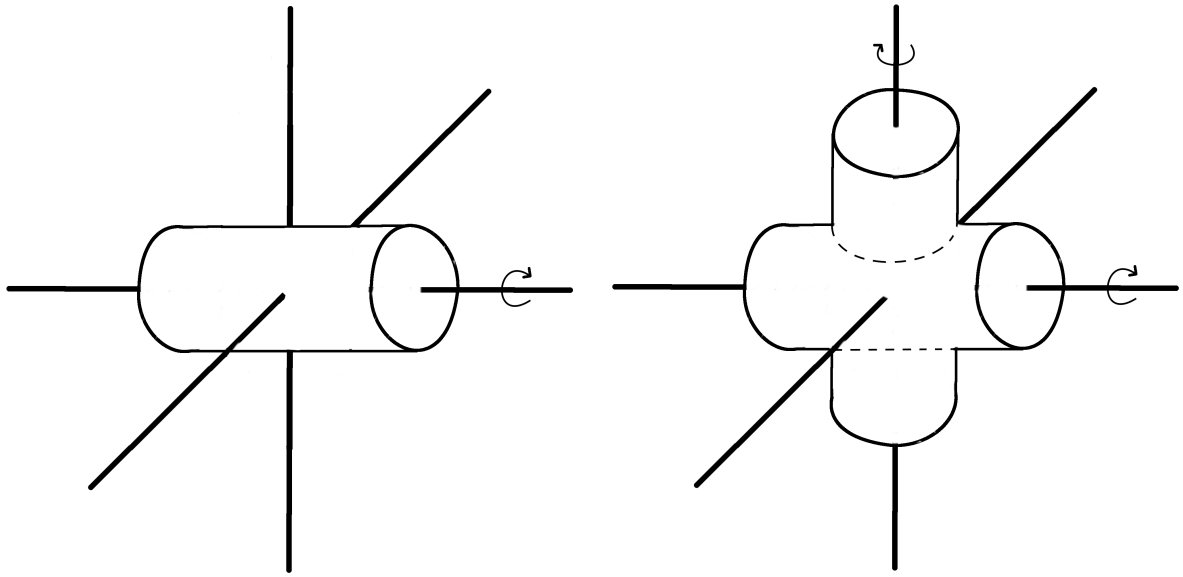
Figur 6: Bilden till vänster visar det synliga lagret, bilden i mitten visar det första osynliga lagret och bilden till höger visar det andra osynliga lagret.

Det första osynliga lagret består av sju dynamiska objekt; ett rätblock som är basobjektet, vilket innebär att alla andra objekt är indirekt kopplade till objektet, och sex cylindrar. Fyra av cylindrarna används för att driva bilen framåt eller bakåt, dvs hjulen, och två cylindrar används som styraxlar för att styra bilen. Alla dessa objekt är av typen *pure simple shape* vilket rekommenderas eftersom de har ett lågt polygonantal som ger bättre prestanda [27]. Modellen använder en så kallad *child/parent-struktur* där basobjektet, eller *parent*-objektet, har ett eller flera *child*-objekt som i sin tur kan ha ytterligare flera child-objekt. Ett child kommer alltid behålla sin relativa position till sin parent och då parent-objektet förflyttas vid simulation så kommer positionen av samtliga child-objekt under parent-objektet att uppdateras så att de alltid befinner sig på samma punkt relativt sitt parent-objekt. Detta innebär att varje objekt har potential att vara både child och parent samtidigt och är alltså inte bundna till det ena eller andra. Child/parent strukturen kan ses i figur 7



Figur 7: Gulliver-modellens struktur i V-REP. Varje indentering är en subnivå i child/parent strukturen.

För att dynamiskt sammankoppla två objekt används *joints*. Modellen använder så kallade *revolute joints* där rotation bara sker kring en axel. En joint har ett parent-objekt som inte påverkas av rotationen och ett child-objekt som rör sig med rotationen, enligt den tidigare nämnda child/parent-strukturen. På bilens bakhjul finns två joints på varje sida som används som motorer. Joint-objekten har basobjektet som parent och en cylinder som child-objekt. När motorn roterar följer cylindern rotationen och bilen rullar då framåt eller bakåt. På framhjulen finns två liknande joints, med tillhörande cylindrar, på varje sida där skillnaden är att motorn är passiv för att efterlikna en fri axel mellan cylindrarna. På framsidan finns ytterligare två joints som istället roterar tillhörande cylinder kring z-axeln relativt xy-planet bilen kör på. Detta innebär rent praktiskt att dessa två joints ansvarar för att svänga bilen. Eftersom cylindern för hastighet är child till cylindern för styrning kommer bilen rulla i styrvinkeln. Alla joints ligger i det andra osynliga lagret. En grafisk representation av hur dessa revolute joints ser ut kan ses i figur 8. I dessa bilder kan man tänka sig bakaxeln som roterar bilens bakhjul i färdriktningen, samt framaxeln som inte bara roterar framhjulet i färdriktningen, utan även vrider framhjulet för att svänga och därmed justera färdriktningen.



Figur 8: Grafisk representation av *revolute joints* för bak- respektive framaxlar på bilen.

För att öka prestandan i en simulation kan man välja att göra det synliga lagret osynligt och istället göra det första osynliga lagret synligt. Detta resulterar i en visualisering som inte är lika estetiskt lockande men i övrigt uppfyller samma funktion.

Eftersom tekniska specifikationer på den fysiska Gulliver-bilen inte kunde tillhandahållas så uppskattades maximal svängradie och hastighet för modellen.

4.6.2 Miljön

Den virtuella miljön skapades på enklast möjliga sätt för att hålla nere prestandakravet så mycket som möjligt. Grunden på miljön är ett standard golv och ovanpå detta har det dragits en karta för att representera den väg som bilen bör följa. En karta i V-REP är en platt yta som beskrivs av ett antal vektorer, som sedan kan exporteras med hjälp av API:t. De sista elementen som utgör miljön är de ankare som används för trilateration. Dessa placeras med ungefär 120 graders vinkel mellan varandra, och är vid samma position i alla de olika miljöerna.

4.6.3 Child scripts

Ett child script är ett litet program, skrivet i programmeringspråket Lua, som hanterar en eller flera önskade funktioner i simulationen. Ett child script kontrollerar oftast en modell, exempelvis en ultraljudssensor, och är associerat med ett objekt i modellen. Ett child script kan även kontrollera flera modeller då en mer central hantering önskas. Child script tillsammans med V-REP:s mångsidiga API gör det möjligt att få alla de funktioner som önskas i den virtuella modellen. Totalt har Gulliver-modellen fem child script varav tre har skapats i projektet och kommer förklaras. De resterande två är child script som är associerade med accelerometern och gyroskopet. De är standardscript som medföljer

modellerna.

Child scripten för ultraljudssensorerna heter `SRF08_L` och `SRF08_R` och är identiska så när som på namnen som syftar på om det är högra eller vänstra ultraljudssensorn. Om ROS är aktiverat startas två publishers, `SRF08_L` och `SRF08_R`, en för respektive sensor, och avstånd i meter till närmsta objekt skickas om det finns något objekt inom räckhåll. Om inget objekt är inom räckhåll skickas värdet 0.

I `Gulliver_model` hanteras resten av modellerna. Om ROS är aktiverat startas först en subscriber, `motor`, som lyssnar på indata till motorregulatorn. Sedan startas fyra subscribers; `pathType` som skickar karttypen med vägpunkter för den autonoma styrningen att följa, `gyroAcc` som modellerar IMU:n och kontinuerligt skickar x och y-värden från accelerometern och z-värdet från gyroskopet, `position` som kontinuerligt skickar bilens absoluta position och `ank1/ank2/ank3_distance` som modellerar RCM:er och kontinuerligt skickar avståndet från ankare ett, två eller tre till bilens egna RCM-modell. Scriptet hanterar ROS-styrning av bilen via både handkontroll och autonom styrning samt tangentbordstyrning då ROS inte är aktiverat, båda styrsätten använder ackermannstyrning. För fullständig källkod för child scripts se Appendix B.

4.6.4 Motorregulator

Bilen använder en väldigt enkel motorregulator. Inparametrarna till motorregulatorn utgörs bara av hastighet och styrvinkel. Motorregulatorn tar den önskade styrvinkeln och sätter olika vinklar på respektive framdäck. Vinklarna på hjulen följer den ackermannmodell som beskrivs i avsnitt 2.3. Hastigheten på de båda bakmotorerna sätts direkt till inparametervärdet. Motorregulatorn har bara en utparameter som är total sträcka bilen har kört. Motorregulatorn implementeras med child script och syns därför inte i modellstrukturen som visas i figur 7.

4.6.5 IMU

För att modellera IMU:n i V-REP används bara en accelerometer och ett gyroskop vilka finns som standard modeller i V-REP. Gulliver-bilens IMU har flera funktioner men eftersom den virtuella modellen bara behöver mäta data från accelerometern och gyroskopet implementerades en enklare IMU. Detta sparar både tid och prestanda. IMU:n läser konstant in x, y och z-värden från modellerna men skickar endast vidare x och y-värden från accelerometern och z-värdet från gyroskopet. Detta eftersom den virtuella kartan är platt, dvs tvådimensionell.

4.6.6 RCM

Gulliver-bilen är utrustad med ett RCM-system för positionsbestämning. Systemet mäter avstånden till tre stycken ankare, och dessa värden används sedan för att bestämma bilens position med hjälp av trilateration. För att simulera systemet i V-REP har tre stycken ankare placerats ut på fasta positioner på de testbanor som har modellerats. V-REP har en inbyggd funktion, *distance measurement*, som mäter avståndet från bilen till varje ankare.

Funktionen utförs i ett child script och det implementerade RCM-systemet syns därför inte i figur 7.

4.6.7 Ultraljudssensorer

Den fysiska Gulliver-bilen använder sig av två ultraljudssensorer på fronten som mäter avstånd till föremål framför bilen. För att modellera dessa sensorer i V-REP används standard modeller som ingår i V-REP. Dessa modeller sätts på varje sida av fronten på den modellerade bilen. Sensorerna syns i bilden till vänster i figur 6 som de röda konerna på framsidan av bilen.

4.6.8 Kommunikation med ROS

För att kommunicera med ROS kan V-REP starta både publisher och subscriber och ange vilken topic de ska skicka till alternativt mottaga från, även meddelandetyp anges. V-REP använder sig av så kallade signaler som innehåller den information som skickas eller mottags. En signal har en primitiv typ men kan inte innehålla ett fält av den typen. Då flera värden av samma typ önskas skickas eller mottags samtidigt packas istället värdena i en sträng och en signal av typen sträng används. Detta kräver att mottagaren av meddelandet packar upp strängen korrekt innan värden kan användas.

4.7 Dataloggning

För att kunna utvärdera och förstå systemet behöver data loggas så att den kan tolkas i efterhand. I V-REP hanteras loggning i child scriptet `Gulliver_model` där önskad data hämtas och skrivs direkt till separata textfiler. ROS har en loggningsfunktion där man anger önskad data som ska loggas och hanteringen sköts av ROS. Data som loggas kommer även användas för att motivera resultat.

4.8 Joystick

För att tidigt kunna testa den virtuella modellen och senare simulera störningar i autopiloten implementerades styrning med joystick, detta fanns inte stöd för i V-REP för Ubuntu så detta implementerades i ROS.

4.9 Tester

Ett antal tester genomfördes under projektets gång. Detta för att utvärdera de system som användes och för framställning av resultat.

4.9.1 Testning av mjukvaruarkitekturen

Testning av mjukvaruarkitekturen genomfördes successivt genom hela projektet. Testningen gick ut på att undersöka varje paket för att säkerställa att kommunikation mellan varje paket i ROS och topic i V-REP fungerar och att korrekta värden skickas. Detta gjordes genom att undersöka den data som ROS rapporterade över de topics relevanta för det testade paketet via ROS egna funktion `rostopic echo [topic-namn]`. Med hjälp

av denna funktion kunde man enkelt observera en dataström från V-REP i ett terminalfönster och jämföra denna mot data som rapporterades lokalt i simulatorn, dock endast då ett värde skickades. Vid kommunikation av strängdata fungerade inte längre denna metod lika dynamiskt utan antaganden var tvungna att göras av hur strängdatan skulle bearbetas i gränssnittet varpå kommandot sedan användas för att istället visa datan efter gränssnittet.

Efter att varje paket och nod testats individuellt testades hela systemet med alla paket och noder samtidigt via observation av hur bilen körde i simulatorn. Det slutgiltiga testet är det mest intressanta, eftersom hela mjukvaruarkitekturen körs. Detta innebär att alla paketen i ROS som skapar vägplaneringen, positionsestimeringen, kollisionsundvikningen nu körs, samt att all data från V-REP, som innefattar sensordata och motorregulatorn, nu skickas och tas emot samtidigt. Denna data loggas sedan via två metoder för att att utvärdera problem som kan tänkas uppstå vid simulationen, samt för att möjliggöra kalibrering av filter vid introduktion av artificiella störningar i sensorerna då ett mer verkligt scenario vill uppnås. Den första metoden är ROS egna, inbyggda, loggfunktion som rapporterar syntax-/kommunikationsfel i koden eller ramverket. Den andra metoden är egenhändigt skrivna loggar som endast skriver data på de mätvärdet som rapporteras för att kunna rita grafer och utvärdera filter och regulator. Då mjukvaruarkitekturen i sig testas används kommandot `rqt_graph` i ROS som ritar en figur över alla topics och noder som körs och länkar samman dessa efter kommunikationsvägar. Om denna figuren visar alla tänkta topics och noder, samt korrekta kommunikationsvägar kan mjukvaruarkitekturen konstateras vara den önskade då ROS egna verktyg anses vara tillförlitliga.

4.9.2 Simulation

Att testa varje sensor som modellerats i V-REP samt motorregulatorn är svårt eftersom projektet inte hade tillgång till någon data från den fysiska modellen. Däremot kan man fortfarande testa att data skickas från de olika sensorerna, samt att dessa ger korrekt utslag på positionsestimeringen samt autopiloten som båda antas fungera sedan tidigare tester utförda av H. Fransson och V. Nilsson på den fysiska Gulliver-bilen.

Bilens motorkontroller testas genom att den använder den data som skickas från ROS till V-REP via joystick och gränssnitt. Detta görs genom att observera hur bilen beter sig då motorkontrollern får styrmeddelanden från de signaler som skickas från joysticken. Eftersom gränssnittet för motorkontrollern på ROS-sidan arbetar med samma meddelandetyp för både joystick och autonom styrning kan motorkontrollern konstateras fungera även för den autonoma styrningen.

4.9.3 Vägplanering

Denna funktion anses vara en av de viktigaste funktionerna att testa. Den testas dels genom observation av simulationen för att se hur bra bilen följer sina vägpunkter, men även genom avläsningar av loggdata för att se hur väl uppmätt positionsdata överensstämmer med vägpunkternas position. Denna loggdata går sedan att använda för att trimma bilens tolerans för när den är nära nog en vägpunkt för att välja att åka vidare till nästa.

4.9.4 Positionsestimation

Testning av positionsestimeringen görs genom att jämföra loggdata från den implementerade positionsestimeringen med loggdata från den faktiska position som simulationen i V-REP säger att Gulliver-modellen har. Ju närmare positionsestimeringens uppskattade position är den faktiska positionen från V-REP desto bättre är positionsestimeringen. För att förbättra positionsestimeringen kan parametrarna för paketet `estimate_position`:s kalmanfilter justeras. Dessutom kan störningar läggas på de virtuella sensorerna för att närmare efterlikna ett verkligt scenario.

4.9.5 Systembelastning

Systembelastningen testas genom att köra simulationen på ett flertal olika system med olika hårdvara för att ge en bild av ungefär den datorprestanda som krävs för att köra simulationen stabilt. Systemspecifikationer finns i tabell 1.

Tabell 1: Visar de systemspecifikationer som använts för att stresstesta simulationen

1	Intel Core i7-5500U 2.40GHz Integrated Intel HD Graphics 8 GB DDR3-1600 Mhz Lubuntu Desktop 14.10
2	Intel Core i7-3630QM 2.40GHz GeForce GTX 670MX 24 GB DDR3-1600 Mhz Lubuntu Desktop 14.04
3	Intel Core i7-4790K 4.00GHz GeForce GTX 770 32 GB DDR3-2133 Mhz Lubuntu Desktop 14.04

Det viktigaste för detta test är att simulatorn fortfarande kan köra i ett realtidsläge för att inte få fel värden i uppdateringen av färdriktningen utifrån gyroskopdatan. Tröskelvärdet för när detta inte längre är möjligt ligger omkring 20 bildrutor per sekund (fps). Test genomförs både med den mindre krävande grafiska modellen samt med den mer krävande. Metodiken för testet var att för varje hårdvarusystem kopiera upp så många kopior som möjligt av bilen i V-REP och observera om simulatorn rapporterade att den inte längre kunde hålla igång realtidsläget. Då simulatorn inte längre klarade av att driva en realtidssimulation så togs den sist tillagda bilen bort och ungefärlig fps observerades och antecknades. ROS var inte aktiverat under detta test men konstaterades via observation av systemets aktivitetshanterare ha en försumbar inverkan på den totala systembelastningen. Orsaken till detta var svårigheter med kommunikation mellan V-REP och ROS vid implementation av flera bilar. Även den totala tiden för arkitekturen att bearbeta data mäts via ROS egna verktyg `rqt_bag` som bland annat visar tidsstämpel för varje nod då någon form av kommunikation sker i ROS ramverket.

5 Resultat

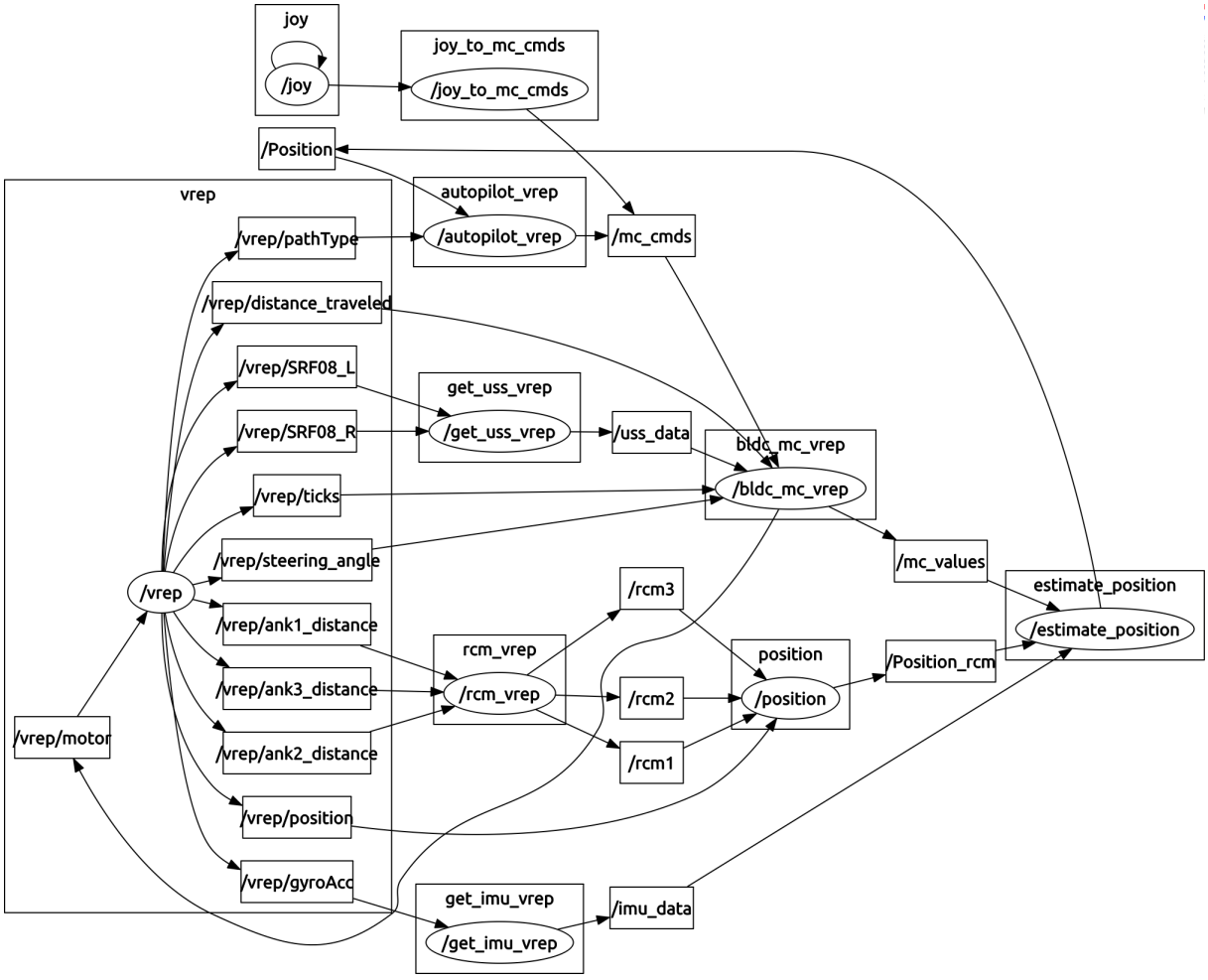
Detta kapitel presenterar resultatet som uppnåtts i projektet. Resultatet redovisas delvis i form av grafer där loggad data jämförs och resultaten av testerna beskrivna i kapitel 4.9 presenteras. Önskad mjukvaruarkitektur jämförs med den faktiska och bilder på den virtuella modellen visas och diskuteras.

5.1 Mjukvaruarkitekturen

För att bygga mjukvaruarkitekturen skapades först en modell över önskad mjukvaruarkitektur. I ROS finns en funktion som visar hur den faktiska arkitekturen, med noder och topics, ser ut. Denna funktion används för att rita upp byggd arkitektur.

5.1.1 Uppnådd mjukvaruarkitektur efter test

För att få förståelse över hur bra den önskade mjukvaruarkitektur sammanfaller med den faktiska mjukvaruarkitekturen används ROS egna verktyg `rqt_graph`. Denna funktion ritar en graf med kommunikationen i ROS genom att rita upp topics och noder samt pilar mellan dessa. Denna graf syns i figur 9. I figuren kan man se rutan *vrep* som motsvaras av den vänstra sidan i figur 5 och resten motsvarar delarna i ROS. Vissa namn är olika men man kan konstatera att det är samma mjukvaruarkitektur.



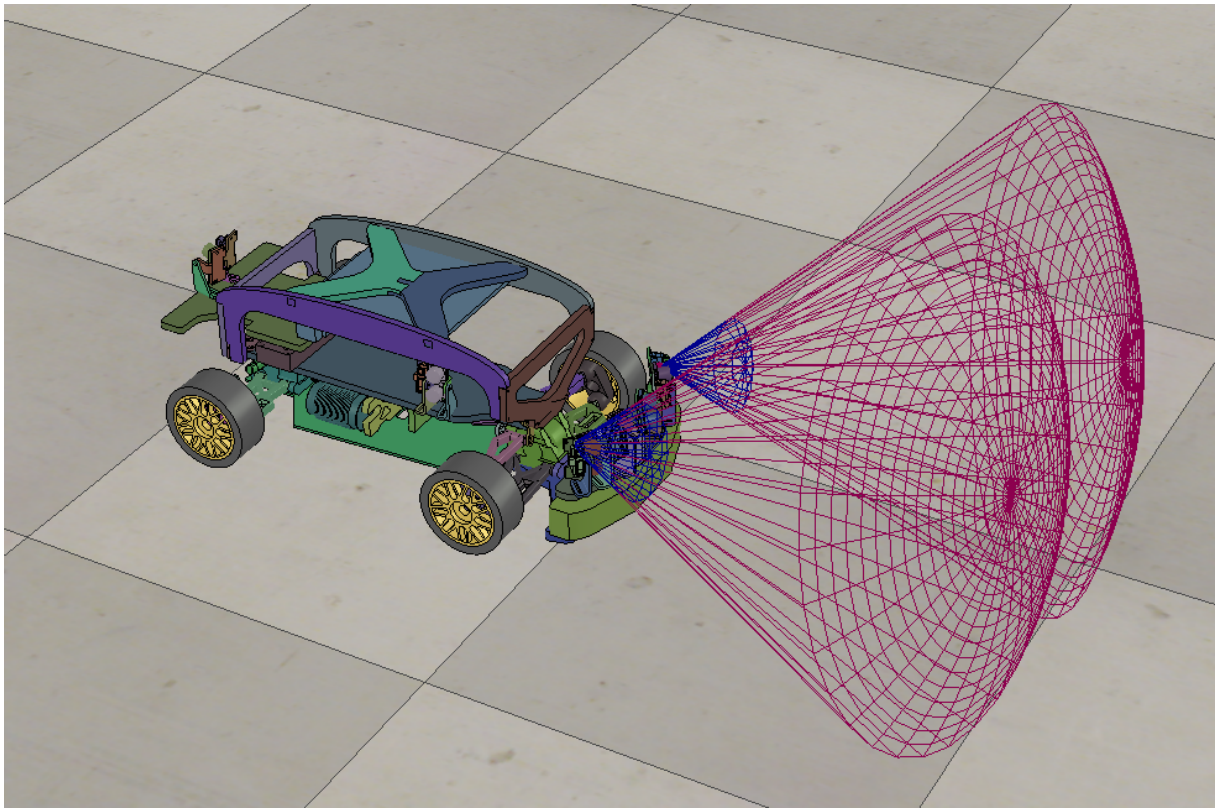
Figur 9: Resultatet från `rqt_graph`. Rektanglarna motsvarar topics och cirklarna motsvarar noder.

5.2 Modellering

Här visas resultatet av den virtuella modellen innehållandes Gulliver-modellen och miljön.

5.2.1 Bilen

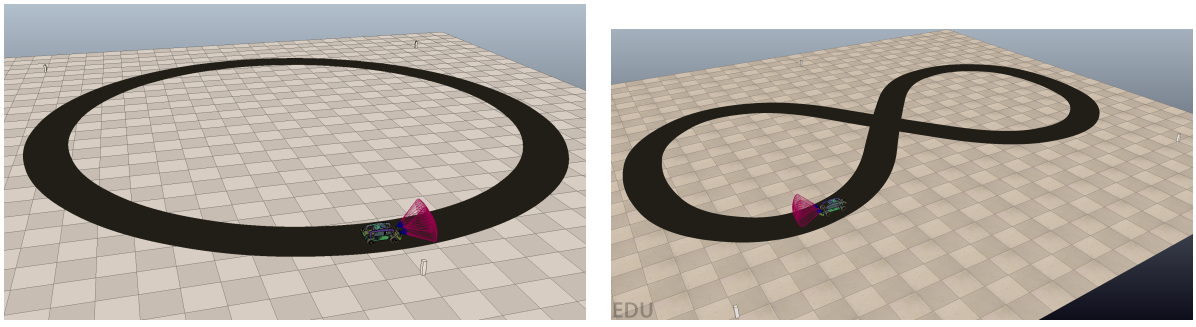
Den slutgiltiga Gulliver-modellen visas i figur 10. Bildelarna som syns, presenterat i olika färger, är de 95 delarna som Gulliver-bilen består av och konerna är ultraljudssensorerna. Resterande sensorer syns inte då de implementeras i form av mjukvara. Önskas bättre prestanda i simulationen kan en visuellt enklare bilmodell användas, vilket beskrivs i kapitel 4.5.1.



Figur 10: Gulliver-bilen modellerad i V-REP med ultraljudssensorer

5.2.2 Miljön

De två miljöerna som modellerats i V-REP kan ses i figur 11. Gulliver-modellen skall köra på de svarta vägarna och varje miljö har tre vita pelare som symboliserar de tre ankarna som behövs för trilateration när man estimerar positionen för Gulliver-modellen. Bilderna är på cirkeln och åttan som skulle skapas i miljön.



Figur 11: Miljö med cirkel som karta med Gulliver-modellen samt de tre ankarna.

5.3 Loggar

Här beskrivs de loggar som egenhändigt skapats för beräkning av sensordata samt vad de loggar. De loggar som skapas av ROS kommer ej att behandlas.

5.3.1 V-REP

Följande loggar visar data från V-REP:

Filnamn	**Förklaring**
`distance.log`	Total sträcka bilen har kört samt tachometervärde.
`imu.log`	x- och y-värden från accelerometern och z-värden från gyroskopet.
`rcm1.log`	Avstånd mellan bilen och *Ankare*.
`rcm2.log`	Avstånd mellan bilen och *Ankare0*.
`rcm3.log`	Avstånd mellan bilen och *Ankare1*.
`position.log`	Bilens absoluta position.
`motor.log`	Bilens hastighet och styrvinkel.
`uss_L.log`	Avstånd till närmsta objekt, inom räckvidd, framför bilen.
`uss_R.log`	Avstånd till närmsta objekt, inom räckvidd, framför bilen.

5.3.2 ROS

Nedanstående ROS-paket innehåller loggar som visar följande data:

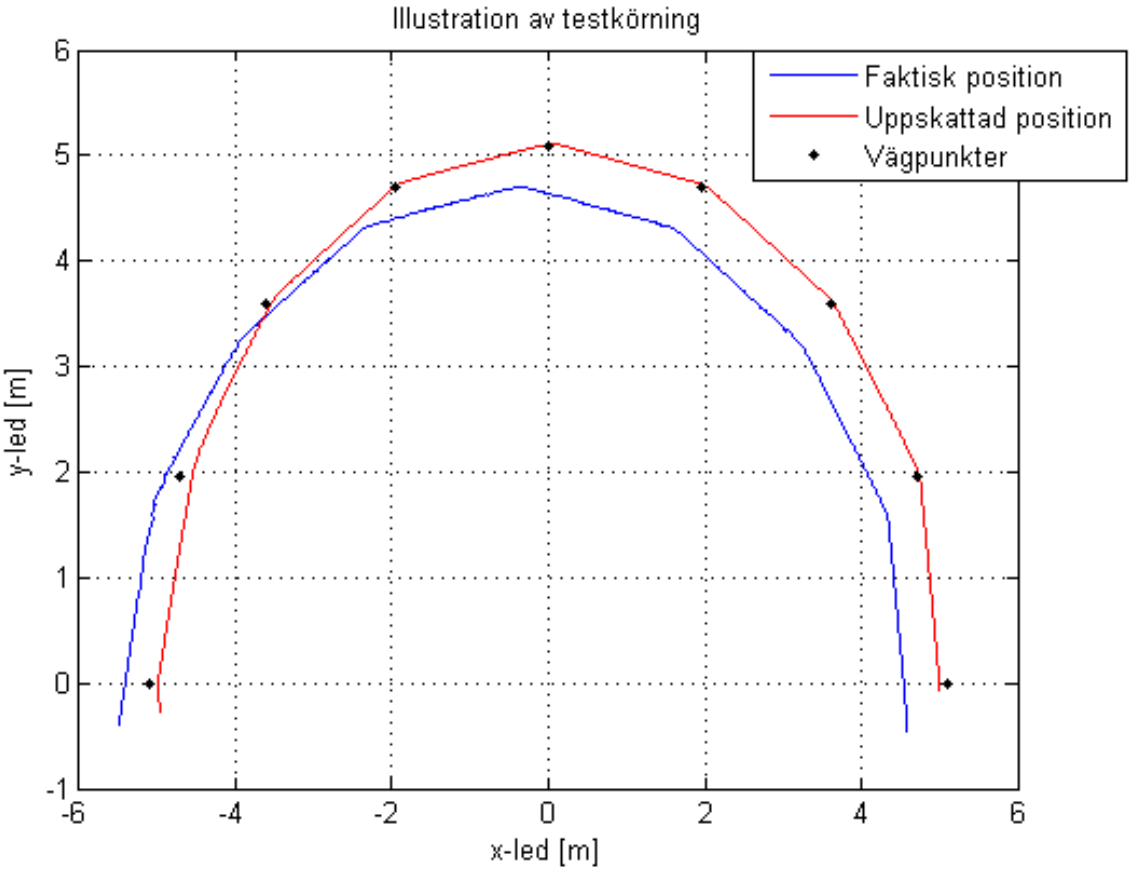
Paket	Förklaring
`autopilot_vrep`	Vilken statisk karta, indikerar om ingen statisk karta lästs in. Vilken position och färdriktning som mottags från `estimate_position`. Uppnådda vägpunkter, när sista vägpunkt har nåtts samt omstart. Indikerar om en vägpunkt är utanför det rimliga området.
`estimate_position`	Position som mottags från trilaterationen. Uppskattad total sträcka körd. Mottagen styrvinkel. Uppskattad färdriktning. Position och färdriktning som publiceras efter kalmanfiltrering.
`rcm_vrep`	Mottagna avstånd, från V-REP, mellan bilen och alla tre ankare.
`position`	Mottagna avstånd, från `rcm_vrep`, mellan bilen och alla tre ankare. Uppskattad position samt om uppskattat värde ligger utanför satt tolerans. Absolut position mottaget från V-REP. Felmarginal mellan uppskattad och absolut position.
`get_imu_vrep`	Mottagna IMU-värden från V-REP.
`get_uss_vrep`	Mottagna värden från ultraljudssensorerna från V-REP.
`joy_to_mc_cmds`	Avlästa joystick värden och resulterande hastighet och styrvinkel.

5.4 Tester

Resultatet av de olika tester som genomfördes under projektet redovisas i detta underkapitel. Data fås från loggarna och några tester är bara observationer.

5.4.1 Vägplaneraren

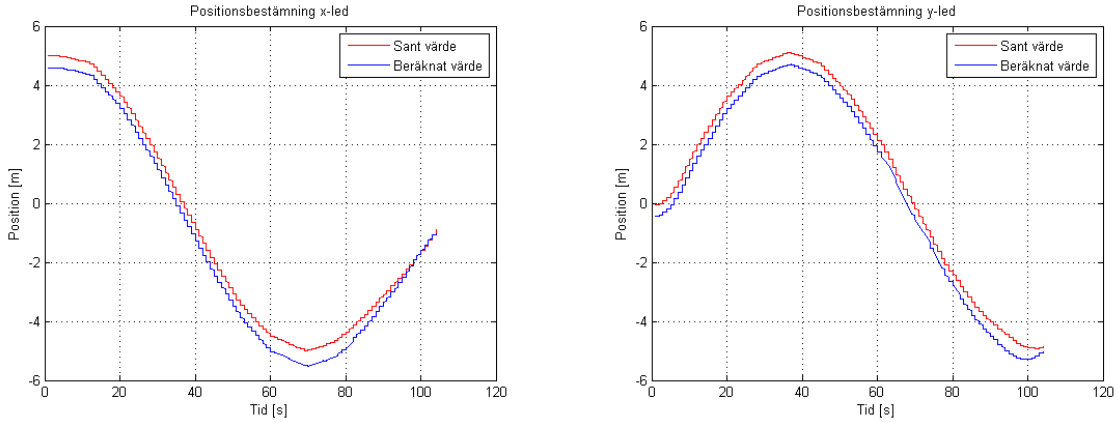
Vägplaneringen testades i miljön med en cirkel som karta genom att låta bilen köra ett varv. Uppskattad position, faktisk position och vägpunkterna som beräknad utifrån loggdata visas i figur 12. För att lättare kunna se resultatet visas bara halva cirkeln. En iakttagelse kan göras från figur 12 som konstaterar att positionsestimeringen inte är perfekt utan har ett fel som varierar mellan försumbart och ungefär en halvmeter från vägpunkterna. Detta observeras genom att den sanna position inte ligger i linje med den estimerade positionen. Däremot så kan man se att autopiloten beter sig helt som önskat då den position bilen tror sig ha stämmer precis överens med vägpunkternas position. Datan i denna figur har hämtats från loggarna i `autopilot_vrep` och `position`.



Figur 12: Resultat av ett halvt varvs körande med vägplaneraren

5.4.2 Positionsuppskattning

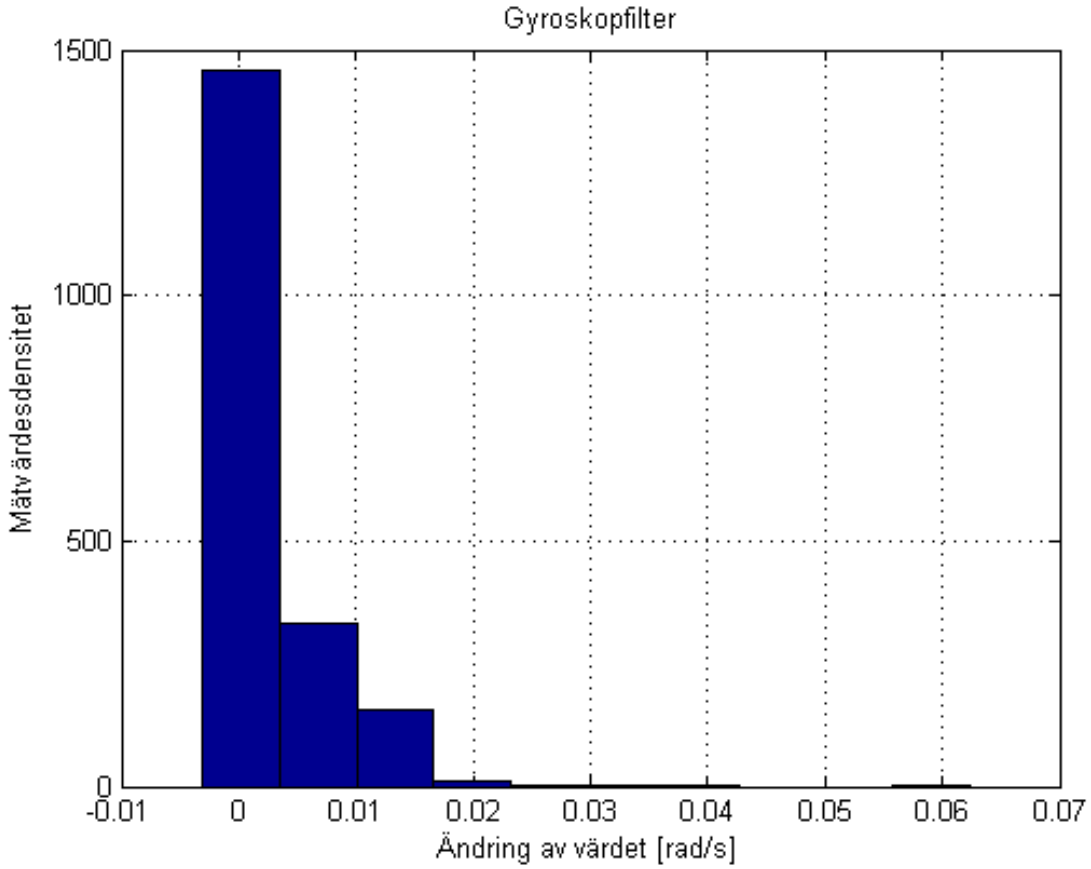
I figur 13 jämförs bilens faktiska position med den uppskattade positionen för att visa skillnaden i x-led respektive y-led. Datan i denna figur har hämtats från loggarna i `position`.



Figur 13: Hur väl positionsuppskattningens beräknade värden stämmer med de sanna värdena.

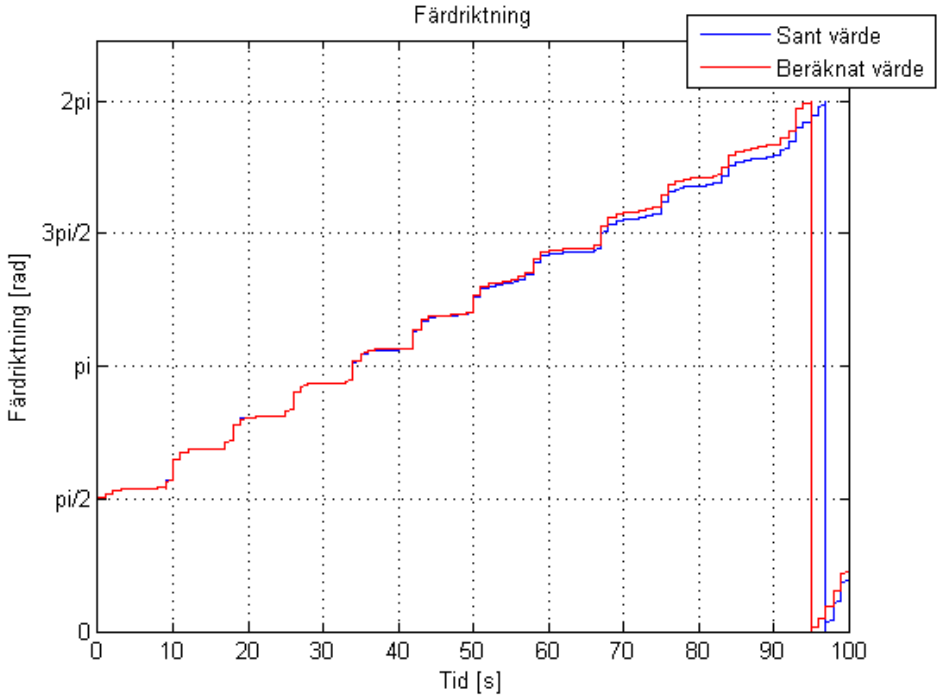
5.4.3 Sensorer och motorkontrollern

Eftersom det antas att kalmanfiltret fungerar väl kan data jämföras innan och efter filtrering för att bekräfta att sensorerna är korrekt modellerade, figur 14 beskriver detta. Nästan ingen skillnad syns före och efter filtrering, detta tyder på att sensorerna skickar data utan brus. Detta skall också vara fallet då inget brus är implementerat i simulationen. Datan i denna figur har hämtats från loggarna i `estimate_position`.



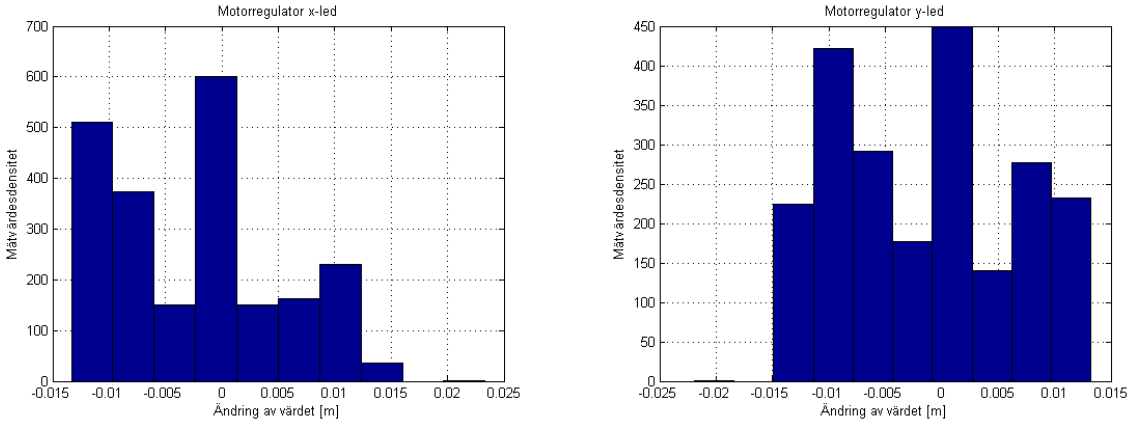
Figur 14: Ett histogram som visar skillnaden i sensorvärde för gyroskopet före och efter filtrering

Sensordatan används sedan för att ändra färdriktningen, det vill säga den riktning som bilens front har. Figur 15 visar vilken beräknad färdriktning som bilen har efter filtrering och jämför det med det faktiska värdet från simulationen. Från figuren kan det utläsas att felet växer med tiden vilket tyder på ett fel i positionsuppskattningen. Om det dessutom skulle komma brusiga signaler från sensorerna hade felet med största sannolikhet varit ännu större. Datan i denna figur har hämtats från loggarna i `get_imu_vrep` och `estimate_position`.



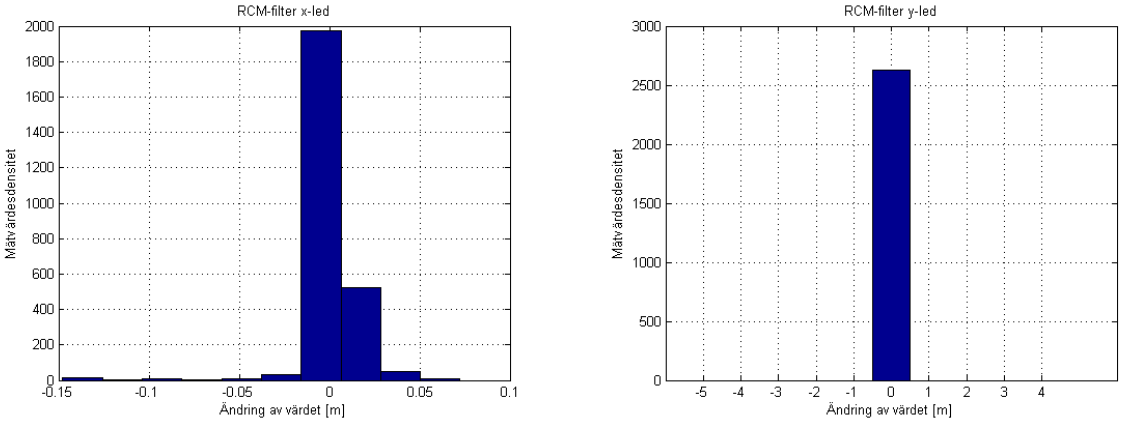
Figur 15: Beskriver skillnaden mellan beräknad färdriktning och faktisk färdriktning. Den plötsliga förändringen sker när bilen kört ett varv runt sin egen axel.

Den önskade färdriktningen skickas sedan till motorregulatorn. Fördelningen av skillnaden på önskad och faktisk färdriktning illustreras i figur 16. Detta bevisar att motorregulatorn tar emot, reglerar och skickar data vilket tyder på att regulatorn fungerar. Datan i denna figur har hämtats från loggarna i `estimate_position`.



Figur 16: Histogram som visar fördelningen av skillnader mellan in- och utdata i x- respektive y-led för motorregulatorn

Datan som skickas från RCM-sensorerna filtreras också i Kalmanfiltret. Skillnaden före och efter filtreringen beskrivs i figur 17. Datan i denna figur har hämtats från loggarna i `estimate_position`.



Figur 17: Histogram som visar fördelningen av positionskoordinater i x- respektive y-led för RCM-systemet

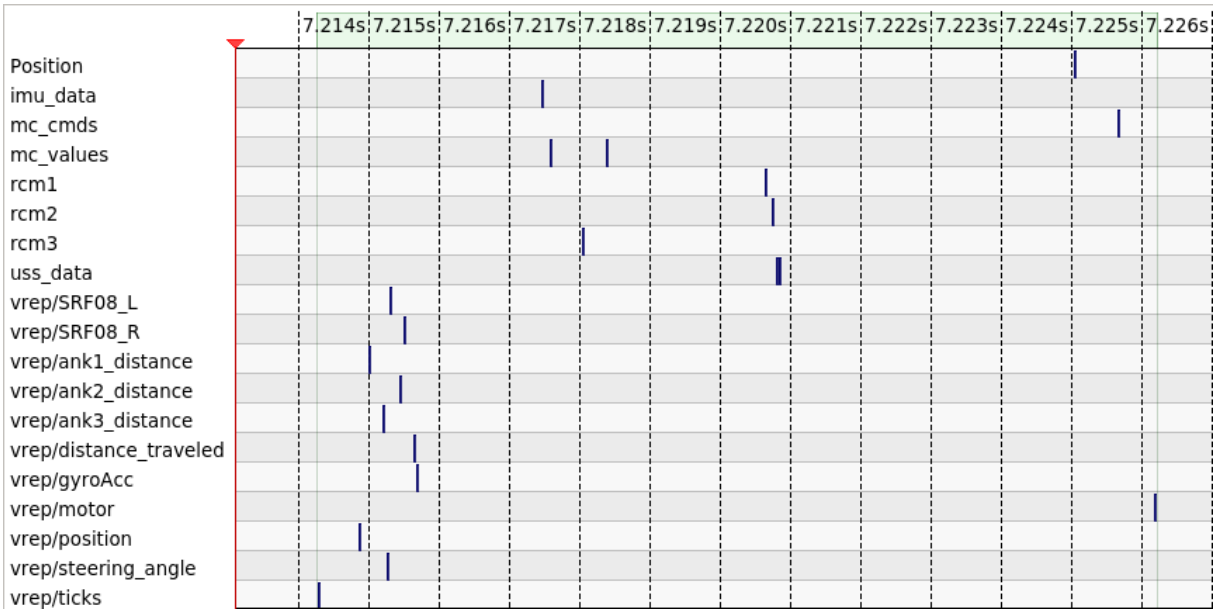
5.4.4 Systembelastning

Tabell 2 beskriver de systemkrav som behövs för att simulationen skall kunna köra i realtid. Bilarna kör inte i konvoj när detta test genomförs.

Tabell 2: Visar de systemspecifikationer som använts för att stresstesta den lågupplösta samt högupplösta Gulliver-modellen. tröskelvärdet för att inte förlora realtid för simulationen ligger strax under 20 fps.

1	Intel Core i7-5500U 2.40GHz Integrated Intel HD Graphics 8 GB DDR3-1600 Mhz Lubuntu Desktop 14.10	Högupplöst modell: 0 bilar
		Lågupplöst modell: 2 bilar, 22 fps
2	Intel Core i7-3630QM 2.40GHz GeForce GTX 670MX 24 GB DDR3-1600 Mhz Lubuntu Desktop 14.04	Högupplöst modell: 1 bil, 25 fps
		Lågupplöst modell: 7 bilar, 20 fps
3	Intel Core i7-4790K 4.00GHz GeForce GTX 770 32 GB DDR3-2133 Mhz Lubuntu Desktop 14.04	Högupplöst modell: 3 bilar, 20 fps
		Lågupplöst modell: 13 bilar, 20 fps

Det finns en tidsfördröjning i mjukvaruarkitekturen vilket är den tid det tar för den kritiska processen att börja om. När mjukvaruarkitekturen kördes på en dator enligt specifikationerna i rad ett i tabell 2 hade den en tidsfördröjning på ungefär 12 millisekunder, resultatet visas i figur 18 där vrep/motor var den kritiska processen.



Figur 18: Beskriver den tidsfördröjning varje paket har. Mjukvaran startar vid markeringen för *vrep/ticks*. Markeringen i varje rad visar när noden har kört en iteration

6 Diskussion

I följande kapitel diskuteras de resultat som har fåtts från projektet, fördelar och nackdelar med systemet samt framtida arbeten som kan vidareutveckla detta.

6.1 Mjukvaruarkitekturen

Den skapade mjukvaruarkitekturen kan optimeras då flera av paketen är skrivna i Python för tidsbesparing. Om all mjukvara istället skrevs i C++ förmodas mjukvaruarkitekturen ha mindre tidsfördröjning, då detta språk har en generellt högre prestanda. Samtidigt antas att kompatibiliteten till den fysiska modellen finns kvar eftersom topics har samma namn och samma meddelandetyper används. Detta testas dock aldrig, vilket inte heller var planerat, men förhoppningsvis underlättar detta för framtida arbeten.

6.2 Positionsuppskattning samt vägplaneraren

Som ses i figur 12 innehåller resultaten från vägplaneraren en felmarginal som ligger mellan 0 till 0.5 meter från den önskade vägen. Detta tror vi kan spåras tillbaka till positionsuppskattningen eftersom vägplaneraren följer den önskade vägen för den uppskattade positionen men inte för den faktiska positionen. Troligtvis är felet i positionsuppskattningen ett fel som beror på flera parametrar. En av parametrarna syns i figur 13 där positionsuppskattningen har ett förhållandevis konstant fel förutom när Gulliver-modellen svänger skarpt, då ökar felet. Vi uppmärksammar också en annan felparameter vilket är att felet på färdriktningen byggs upp över tiden som syns i figur 15. Vi tror att dessa två anledningar bidrar till en sämre positionsuppskattning.

En annan anledning till felmarginalen i positionsuppskattningen kan vara att kalmanfiltret har ett för lågt kovariansvärde. Detta ackumulerar upp ett fel ju längre bilen kör eftersom även här används gamla värden när det nya skall beräknas, och resulterar i en större felmarginal ju längre bilen har kört.

Ur histogrammen kan det avläsas att Kalmanfiltret i `estimate_position` gör ytterst små regleringar i det avlästa mätvärdet. Detta tros betyda att orsaker till mätfel inte ligger i parametervärden för filter, utan snarast någonstans i arkitektur eller simulator. Mest troligt skulle vara avrundningsfel mellan simulator och ramverk.

Positionsuppskattningen använder sig också av trilaterering för att bestämma var bilen befinner sig, men detta ger upphov till problem när bilen är i rörelse. Under den tiden det tar för systemet att mäta avståndet till ankarna, skicka det till ROS-noden för positionsuppskattning, samt använda datan för att beräkna den faktiska positionen, så befinner sig bilen redan vid en ny position. Detta gör att när positionen jämförs med den önskade positionen för att ställa in styrvinkeln som krävs för att nå dit, så är den inte helt korrekt. Därför gör bilen korrigeringar av färdriktningen senare än vad den borde, som i sin tur ger upphov till en felmarginal mellan den önskade och den faktiska vägen. Den effekten blir mer påtaglig ju högre hastighet bilen har, men kan motverkas genom att ha en låg hastighet och en hög frekvens på RCM:erna.

6.3 Prestanda

Simulationen som genomförs i V-REP kräver ett kraftfullt datorsystem för att fungera, vilket ses i tabell 2, där system ett är en standardlaptop medan system tre är en stationär speldator. Detta är ett problem speciellt för konvojkörning om man har ett sämre datorsystem. Även då simulationen är optimerad med lågupplöst modell och en simpel miljö kräver V-REP mycket beräkningskraft, men vi anser att systemet fortfarande är applicerbart trots de höga systemkraven. Vi ser också att det finns möjlighet att implementera den konvojkörning och kommunikation vi önskar om man har datorsystem tre i tabell 2 samt använder den lågupplösta modellen.

Datorkraften som finns tillgänglig för simuleringen påverkar tidsfördröjningen i mjukvaruarkitekturen. Denna kunskap är värdefull när man ska bestämma hur ofta simulationen ska uppdateras, denna tid kan variera i tidssteg mellan 10-200 millisekunder. I det fall som är testat på datorsystem ett enligt tabell 2, det vill säga det sämst presterande systemet, skulle tidssteg i simulatorn under tolv millisekunder kunna resultera i att ROS inte hinner genomföra alla beräkningar innan simulationen uppdateras vilket gör att beräknade värden inte längre är aktuella. För att få bästa resultat bör därmed tidsfördröjningen för det aktuella systemet mätas för att undvika denna typ av fel, så att uppdateringstiden i V-REP kan justeras därefter.

6.4 Framtida utmaningar

Framtida utmaningar för andra projekt är att färdigställa ett fungerande kommunikationssystem i mjukvaruarkitekturen för förbättrad konvojkörning. En enkel förberedelse för nätverkskommunikation har implementerats. Framtida arbeten kan också säkerställa, utveckla nya eller förbättra olika filter och regulatorer för att förbättra vägplaneringen och positionsuppskattningen samt få systemet att köra i den fysiska modellen.

Något som även behöver tänkas på när simulerade system skall testas i en fysiska miljö är att de sensorer som finns i V-REP inte innehåller något brus, samt att de alla beter sig exakt som specificerat. I verkligheten är det inte så enkelt då brus i signalerna samt små individuella skillnader mellan sensorer av samma typ kan ge olika resultat. Därför behövs det diverse filter för att få samma resultat på den fysiska bilen som på den simulerade. Ett sätt att lösa det problemet är att manuellt införa någon form av slumpmässig data i de signaler som mäts upp av V-REP som då får simulera de störningar som alltid återfinns i fysiska system.

Den slutgiltiga utmaningen blir att implementera ett helt fungerande system på den fysiska Gulliver-bilen efter att ha testat det i den virtuella modellen. Vi anser att en modell i V-REP kan efterlikna verkligheten till den grad att det blir jämförbart med en fysisk modell. Vår modell behöver dock vidareutvecklas, som diskuterat, för att uppnå denna noggrannhet.

7 Slutsats

En av utmaningarna med att testa autonoma styrsystem menade för fullskaliga fordon är testkörningar av dessa innan de är säkra nog att användas i verkliga situationer. Ofta används nedskalade modeller, ett exempel är plattformen Gulliver som är en av grunderna till projektet, men även dessa är både dyra och tidskrävande att bygga, samt tar tid att ändra om man vill testa med olika värden på parametrar och olika typer av sensorer. Den virtuella modell som har utvecklats i detta projekt är därför ett utmärkt verktyg att använda som en första plattform när autonoma system skall utvecklas. Efter utvecklingen av grundsystemet är det ett mycket billigt och tidseffektivt verktyg. Flera bilar kan även simuleras samtidigt, något som är både dyrt och riskabelt att göra med fysiska bilar.

En av de viktigaste slutsatserna av detta projekt är att verktygen vi använder och den mjukvaruarkitektur som byggts för vår virtuella modell fungerar bra och kan vidareutvecklas. Olika filter och regulatorer för bilar kan lätt utvecklas och testas eftersom miljön är modulär. Detta projekt går att utveckla till att omfatta väldigt mycket. Det går att utveckla och testa nya system utan att behöva slösa tid och pengar på hårdvara.

Referenser

[1] M. Peden, R. Scurfield och D. Sleet. *World Report on Road Traffic Injury Prevention. Albany, NY, USA*. 2004. URL: http://site.ebrary.com.proxy.lib.chalmers.se/lib/chalmers/reader.action?docID=10053723&ppg=22 (hämtad 2015-01-27).

[2] M. Arora, P. Kochar och T. Gandhi. *Autonomous Driven Car, i 2nd International Conference on Emerging Trends in Engineering and Managemen*. 2013. URL: http://www.journals.rgsociety.org/index.php/ijse/article/view/422/206 (hämtad 2015-01-27).

[3] E. Frazzoli. *Can We Put a Price on Autonomous Driving?* 2014. URL: http://www.technologyreview.com/view/525591/can-we-put-a-price-on-autonomous-driving/ (hämtad 2015-05-14).

[4] Papatriantalou-M. Pahlavan M. och E. M. Schiller. *Gulliver: a test-bed for developing, demonstrating and prototyping vehicular systems.* 2011.

[5] H. Fransson och V. Nilsson. Privat kommunikation. 2015.

[6] E Eriksson, F. Hagslätt, R. Nard m. fl. "Gulliver Simulation: En systemarkitektur for natverkssimulering av kooperativa robotar". I: (2015).

[7] Christian Berger, Erik Dahlgren, Johan Grundén m. fl. "Bridging Physical and Digital Traffic System Simulations with the Gulliver Test-Bed". English. I: (2013).

[8] Kajko-Mattsson M. Nikitina N. och M. Stråle. "From scrum to scrumban: A case study of a process transition C3 - 2012 International Conference on Software and System Process, ICSSP 2012 - Proceedings". I: *2012 International Conference on Software and System Process, ICSSP 2012* (2012), s. 140 –149.

[9] T. Glad och L. Ljung. *Reglerteknik: grundläggande teori*. 4th. Studentlitteratur, 2006.

[10] Cruceru-C. Suliman C. och F. Moldoveanu. "Mobile robot position estimation using the Kalman filter". I: *Department of Automation* (2009), s. 1–3.

[11] Burgard W. Thrun S. och D. Fox. *Probabilistic Robotics*. MIT Press, 2005, s. 34 –39.

[12] J. M. Snider. *Automatic steering methods for autonomous automobile path tracking*. Pittsburgh, PA, Tech. Rep. CMU-RITR-09-08, 2009.

[13] F. Thomas och L Ros. "Revisiting trilateration for robot localization". I: *IEEE Transactions on Robotics* 21.1 (2005), s. 93–101.

[14] *About ROS*. URL: http://www.ros.org/about-ros/ (hämtad 2015-05-14).

[15] J. Bohren. *Packages*. 14 mars 2014. URL: http://wiki.ros.org/Packages (hämtad 2015-05-14).

[16] K. Conley. *Nodes*. 3 febr. 2012. URL: http://wiki.ros.org/Nodes (hämtad 2015-05-14).

[17] N. Lamprianidis. *Messages*. 18 nov. 2012. URL: http://wiki.ros.org/Messages (hämtad 2015-05-14).

[18] D. Forouher. *Topics*. 1 juni 2014. URL: http://wiki.ros.org/Topics (hämtad 2015-05-14).

[19] *PulsON 410 Ranging and Communication Module Data Sheet*. Time Domain. Nov. 2013. URL: http://www.timedomain.com/datasheets/320-0289E\%20P410\%20Data\%20Sheet.pdf (hämtad 2015-05-14).

[20] *MEMS motion sensor: three-axis digital output gyroscope L3GD20 Data Sheet*. STMicroelectronics. Febr. 2013. URL: http://www.st.com/web/en/resource/technical/document/datasheet/DM00036465.pdf (hämtad 2015-05-14).

[21] *Ultra-compact high-performance eCompass module: 3D accelerometer and 3D magnetometer LSM303DLHC Data Sheet*. STMicroelectronics. Nov. 2013. URL: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00027543.pdf (hämtad 2015-05-14).

[22] *SRF08 Ultra sonic range finder Data Sheet*. Devantech. URL: http://www.robot-electronics.co.uk/htm/srf08tech.shtml (hämtad 2015-05-14).

[23] *Features - V-REP*. URL: http://www.coppeliarobotics.com/features.html (hämtad 2015-05-14).

[24] W. Woodall, M. Liebhardt, D. Stonier m. fl. "ROS Topics: Capabilities [ROS Topics]". I: *Robotics Automation Magazine, IEEE* 21.4 (2014), s. 14–15.

[25] Oscar Morales-Ponce, Elad M. Schiller och Paolo Falcone. "Cooperation with Disagreement Correction in the Presence of Communication Failures". English. I: (2014).

[26] A. Casimiro, O. Morales Ponce, T. Petig m. fl. "Vehicular Coordination via a Safety Kernel in the Gulliver Test-Bed". I: (2014), s. 167–176. ISSN: 1545-0678. DOI: 10.1109/ICDCSW.2014.25.

[27] *Manual V-REP*. URL: http://www.coppeliarobotics.com/helpFiles/index.html (hämtad 2015-05-14).

Appendix A

ackermann_msgs

AckermannDrive.msg

```
## Driving command for a car-like vehicle using Ackermann steering.
#  $Id$

# Assumes Ackermann front-wheel steering. The left and right front
# wheels are generally at different angles. To simplify, the commanded
# angle corresponds to the yaw of a virtual wheel located at the
# center of the front axle, like on a tricycle.  Positive yaw is to
# the left. (This is *not* the angle of the steering wheel inside the
# passenger compartment.)
#
# Zero steering angle velocity means change the steering angle as
# quickly as possible. Positive velocity indicates a desired absolute
# rate of change either left or right. The controller tries not to
# exceed this limit in either direction, but sometimes it might.
#
float32 steering_angle          # desired virtual angle (radians)
float32 steering_angle_velocity # desired rate of change (radians/s)

# Drive at requested speed, acceleration and jerk (the 1st, 2nd and
# 3rd derivatives of position). All are measured at the vehicle's
# center of rotation, typically the center of the rear axle. The
# controller tries not to exceed these limits in either direction, but
# sometimes it might.
#
# Speed is the desired scalar magnitude of the velocity vector.
# Direction is forward unless the sign is negative, indicating reverse.
#
# Zero acceleration means change speed as quickly as
# possible. Positive acceleration indicates a desired absolute
# magnitude; that includes deceleration.
#
# Zero jerk means change acceleration as quickly as possible. Positive
# jerk indicates a desired absolute rate of acceleration change in
# either direction (increasing or decreasing).
#
float32 speed                   # desired forward speed (m/s)
float32 acceleration            # desired acceleration (m/s^2)
float32 jerk                    # desired jerk (m/s^3)
```

AckermannDriveStamped.msg

```
## Time stamped drive command for robots with Ackermann steering.
#  $Id$

Header           header
AckermannDrive   drive
```

autopilot_vrep
ap_utils.py

```
#!/usr/bin/env python
# Software License Agreement (BSD License)
#
# Copyright (c) 2014, Viktor Nilsson and Herman Fransson.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
#  * Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
#  * Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
#    with the distribution.
#  * Neither the name of Willow Garage, Inc. nor the names of its
#    contributors may be used to endorse or promote products derived
#    from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
# Revision $Id$

import math
import rospy # Only for logging/debugging
import config

prev_error = 0
i_term = 0

# Returns the difference between angle to next waypoint and heading of the
    car.
def get_angle_error(px,py,cx,cy,heading):
  length = math.sqrt(pow((px-cx),2) + pow((py-cy),2))
  cxx = length * math.cos(heading)
  cyy = length * math.sin(heading)
  pxx = px-cx
  pyy = py-cy
  angle = math.atan2(pyy, pxx) - math.atan2(cyy, cxx)

```

```
52    angle = angle % (2*math.pi)
53    if angle > math.pi:
54      angle -= 2*math.pi
55
56    #rospy.loginfo("heading: %f, (%f, %f) -> (%f, %f), angle error is %f" %
57    #               (heading, cx, cy, px, py, angle))
58    return angle
59
60
61 # Given input, returns the distance between the circle arc (that the car is
62 # believed to travel) and the checkpoint p.
63 def is_on_track(waypoint,cx,cy,heading,steering_angle):
64    px,py = waypoint
65    R = config.L/(math.tan(steering_angle))
66    angle_to_origo = math.pi/2+heading
67    o_x = R*math.cos(math.pi-angle_to_origo)
68    o_y = R*math.sin(math.pi-angle_to_origo)
69    distance_o_to_p = math.sqrt(pow((px-o_x),2)+pow((py-o_y),2))
70
71    return distance_o_to_p-R
72
73
74 # Every time we get a new position,
75 # we adjust the steering angle
76 # input: wp = (x,y) of next waypoint
77 #        (cx,cy) = current position of car
78 #        heading = current heading of car (not steering angle)
79 # output: a new heading in degrees
80 def update_steering_angle(wp,cx,cy,heading,steering_angle):
81    global i_term, prev_error
82    px,py = wp
83    angle_error = get_angle_error(px,py,cx,cy,heading)
84
85    p_term = angle_error * config.K_p
86    i_term += (angle_error / config.POS_UPDATE_FREQ) * config.K_i
87    d_term = ((angle_error - prev_error) * config.POS_UPDATE_FREQ) * config.
       K_d
88
89    steering_angle = p_term + i_term + d_term
90    prev_error = angle_error
91
92    return max(config.MIN_STEERING_ANGLE,
93               min(steering_angle, config.MAX_STEERING_ANGLE))
94
95
96 def is_point_behind(point,car_x,car_y,heading):
97    x,y = point
98    # This is the distance between the rear axis and the imaginary line which
       the
99    # given point is or is not behind. E.g., if equal to 0.3 [m], every point
100   # behind the front axis is said to be behind.
101   d = 0.1
102
103   # Get point of car front
104   (front_x,front_y) = get_new_point(car_x,car_y,d,heading)
```

```
105   # Calculate a point p1 that is to the left of front
106   (p1x,p1y) = get_new_point(front_x,front_y,10,heading+math.pi/2)
107   # Calculate a point p2 that is to the right of front
108   (p2x,p2y) = get_new_point(front_x,front_y,10,heading-math.pi/2)
109
110   # This catches some corner-cases due to rounding errors
111   if heading == math.pi:
112     return car_x-d < x
113   elif heading == 0:
114     return car_x+d > x
115
116   if (heading < math.pi):
117     return is_point_behind_line(x,y,p1x,p1y,p2x,p2y)
118   else:
119     return is_point_behind_line(x,y,p2x,p2y,p1x,p1y)
120
121
122 # Returns a new point from angle and length of vector
123 def get_new_point(x,y, length, heading):
124   new_x = x + length * math.cos(heading)
125   new_y = y + length * math.sin(heading)
126   return (new_x,new_y)
127
128
129 # Returns true if (x,y) is under the line that is between the points
130 # (px1,px1) and (px2,py2).
131 def is_point_behind_line(x,y,px1,py1,px2,py2):
132   v1 = (px2-px1, py2-py1)
133   v2 = (px2-x, py2-y)
134   xp = v1[0]*v2[1] - v1[1]*v2[0]
135
136   if xp > 0:
137     return True
138   elif xp < 0:
139     return False
140   else:
141     return False
142
143 def is_close_enough(wp,x,y,margin):
144   wp_x,wp_y = wp
145   if abs(wp_x-x) < margin and abs(wp_y-y) < margin:
146     return True
147   else:
148     return False
```

config.py

```
from math import radians

L = 0.3
SPEED = 1.0

K_p = 1.2
K_i = 0.0
K_d = 0.000005

MIN_STEERING_ANGLE = radians(-22.0)
MAX_STEERING_ANGLE = radians(22.0)
POS_UPDATE_FREQ = 200
WP_MARGIN = 0.1

MIN_X = -6
MAX_X = 6
MIN_Y = -6
MAX_Y = 6
```

path_planning_vrep.py

```
#!/usr/bin/env python

import rospy, roslib, rospkg
import ap_utils, config
import math, time
from std_msgs.msg import String
from ackermann_msgs.msg import AckermannDrive
from estimate_position.msg import Position
import pandas as pd

last_drive_msg = AckermannDrive()
current_steering_angle = 0
current_wp = 1
finished = False
rospack=rospkg.RosPack()


def scene_callback(data):
  global wps
  if data.data == 'cirkel':
    # Path for Track_cirkel.
    cirkel = pd.read_csv(rospack.get_path('autopilot_vrep') + '/csv/cirkel.csv', delimiter=',', header=None, usecols=[0, 1])
    cirkel_x = cirkel[0].tolist()
    cirkel_y = cirkel[1].tolist()
    wps = [(cirkel_x[i], cirkel_y[i]) for i in range(len(cirkel_x))]
  elif data.data == 'cirkel2':
    # Path for Track_cirkel2.
    cirkel2 = pd.read_csv(rospack.get_path('autopilot_vrep') + '/csv/cirkel21.csv', delimiter=',', header=None, usecols=[0, 1])
    cirkel2_x = cirkel2[0].tolist()
```

```
30      cirkel2_y = cirkel2[1].tolist()
31      wps = [(cirkel2_x[i], cirkel2_y[i]) for i in range(len(cirkel2_x))]
32    elif data.data == 'eight':
33      # Path for Track_eight.
34      eight = pd.read_csv(rospack.get_path('autopilot_vrep') + '/csv/eight.
        csv', delimiter=',', header=None, usecols=[0, 1])
35      eight_x = eight[0].tolist()
36      eight_y = eight[1].tolist()
37      wps = [(eight_x[i], eight_y[i]) for i in range(len(eight_x))]
38    else:
39      print('No scene found.')
40
41
42  # This is run every time a new Position is received.
43  def callback_position(pos, pub):
44    global last_drive_msg, current_steering_angle, current_wp, finished
45
46    rospy.loginfo("\n[AUTOPILOT] Got a new position: x=%f, y=%f, heading=%f"
       % (pos.x, pos.y, pos.heading))
47
48  #  rospy.loginfo("\n[AUTOPILOT] Path is: %s", wps)
49
50    pos.heading = pos.heading % (2*math.pi)
51    drive = AckermannDrive()
52
53    # If we are close enough to current waypoint, aim for next.
54    if ap_utils.is_close_enough(wps[current_wp], pos.x, pos.y, config.
       WP_MARGIN):
55      if current_wp < len(wps)-1:
56        rospy.loginfo("\n[AUTOPILOT] I reached waypoint %i!\n[AUTOPILOT]
       Heading for (%s)" % (current_wp, wps[current_wp+1]))
57        current_wp += 1
58      elif current_wp == len(wps)-1:
59        rospy.loginfo("\n[AUTOPILOT] I reached my destination!")
60        finished = True
61  #    elif ap_utils.is_point_behind(wps[current_wp],
62  #      pos.x, pos.y, pos.heading) and not finished:
63  #      rospy.loginfo("I missed waypoint %i!" % (current_wp))
64  #      current_wp += 1
65
66    if not finished:
67      drive.speed = config.SPEED
68    else:
69      drive.speed = 0.00
70
71    # Set speed to 0 if you are outside your "comfort zone"
72    if not (config.MIN_X < pos.x < config.MAX_X and \
73      config.MIN_Y < pos.y < config.MAX_Y):
74      drive.speed = 0.0
75      rospy.loginfo("\n[AUTOPILOT] I'm not comfortable with this... (Area out
       of bounds.)\n[AUTOPILOT] My position is: (%s, %s)", pos.x, pos.y)
76
77    # Set steering angle based on current position and next waypoint
78    current_steering_angle = ap_utils.update_steering_angle(
79      wps[current_wp], pos.x, pos.y, pos.heading, current_steering_angle)
```

```
80    drive.steering_angle = math.degrees(current_steering_angle)
81 #  rospy.loginfo("\n[AUTOPILOT] Setting steering angle: heading=%s", math.
       degrees(current_steering_angle))
82
83    # Only publish messages if it differs from the last one
84    if drive != last_drive_msg:
85      last_drive_msg = drive
86      pub.publish(drive)
87
88 def autopilot():
89    # Sleep before starting. If parameter not set, default to 5 seconds.
90 #  rospy.Subscriber("/vrep/pathType", String, scene_callback)
91    time.sleep(int(rospy.get_param('Autopilot/wait_time', 1)))
92
93    rospy.init_node("autopilot", anonymous=True)
94    pub = rospy.Publisher('mc_cmds', AckermannDrive, queue_size=1)
95    rospy.Subscriber("Position", Position, callback_position, pub)
96    rospy.spin()
97
98 if __name__ == '__main__':
99    try:
100     autopilot()
101   except rospy.ROSInterruptException: pass
```

bldc_mc_vrep

MCValues.msg

```
Header header
int16 TEMP_MOS1
int16 TEMP_MOS2
int16 TEMP_MOS3
int16 TEMP_MOS4
int16 TEMP_MOS5
int16 TEMP_MOS6
int16 TEMP_PCB
int32 current_motor
int32 current_in
int16 duty_now
int32 rpm
int32 v_in
int32 amp_hours
int32 amp_hours_charged
int32 watt_hours
int32 watt_hours_charged
int32 tachometer
int32 tachometer_abs
float32 current_steering_angle
float32 current_speed
```

mc_vrep_listalk.py

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String, Float32, Int32
from ackermann_msgs.msg import AckermannDrive
from get_uss.msg import srf08_msg
from bldc_mc.msg import MCValues
from struct import pack

MC = MCValues()
sensor_R = 0
sensor_L = 0

def ohshit(data):
  global sensor_R
  global sensor_L
  sensor_R = data.sensor_0
  sensor_L = data.sensor_1

def callback_mc(data, pub):
  rospy.loginfo("\n[MC] Steering Angle=%s, Speed=%s", data.steering_angle,
    data.speed)
  rospy.loginfo("\n[SRF08] Right sensor=%s, Left sensor=%s", sensor_R,
    sensor_L)
  speed = data.speed
  angle = data.steering_angle
```

```
25
26    emergency = False
27
28    if speed > 0 and emergency is False and sensor_R or sensor_L > 0.5:
29      rospy.loginfo("EMERGENCY!\n")
30      emergency = True
31      prev_speed = speed
32      speed = 0
33    elif emergency is True and sensor_R and sensor_L <= 0.5:
34      emergency = False
35      speed = prev_speed
36
37    floatlist = [speed, angle]
38    buf = pack('%sf' % len(floatlist), *floatlist)
39
40    pub.publish(buf)
41
42  def callback_tacos(data, pub):
43    MC.tachometer = data.data
44    pub.publish(MC)
45    rospy.loginfo("\n[MC] Tachometer=%s", data.data)
46
47  def callback_true_distance(data):
48    rospy.loginfo("\n[MC] True distance=%s", data.data)
49
50  def callback_steering_angle(data,pub):
51    MC.current_steering_angle=data.data
52    pub.publish(MC)
53    rospy.loginfo("\n[MC] Steering angle=%s", data.data)
54
55  def mc_vrep():
56    # Sleep before starting.
57    # time.sleep(5)
58
59    rospy.init_node("motor", anonymous=True)
60    pub = rospy.Publisher('/vrep/motor', String, queue_size=1)
61    rospy.Subscriber("mc_cmds", AckermannDrive, callback_mc, pub)
62    rospy.Subscriber("uss_data", srf08_msg, ohshit)
63    pub_mc = rospy.Publisher('mc_values', MCValues, queue_size=1)
64    rospy.Subscriber("/vrep/ticks", Int32, callback_tacos, pub_mc)
65    rospy.Subscriber("/vrep/distance_traveled", Float32,
        callback_true_distance)
66    rospy.Subscriber("/vrep/steering_angle", Float32, callback_steering_angle
        , pub_mc)
67    rospy.spin()
68
69  if __name__ == '__main__':
70    try:
71      mc_vrep()
72    except rospy.ROSInterruptException: pass
```

estimate_position_vrep

Range.msg

```
uint8 anchorId
uint32 rangeMm
uint16 rangeErrEst
uint32 timestamp
```

Position.py

```
#!/usr/bin/env python

import rospy
#import numpy
import math
from estimate_position.msg import Position
from estimate_position_vrep.msg import Range
from std_msgs.msg import String
from struct import unpack

d1=rospy.get_param('d1', 6000)/1000
d2=rospy.get_param('d2', 6000)/1000
d3=rospy.get_param('d3', 6000)/1000

tol = 0.1
mpt = 1/450

pos = Position()

def anch1(data):
  global d1
  d1_temp=data.rangeMm+0.0
  d1=d1_temp/1000
  rospy.loginfo("\n[POS_RCM_1] I heard: %f", d1)

def anch2(data):
  global d2
  d2_temp=data.rangeMm+0.0
  d2=d2_temp/1000
  rospy.loginfo("\n[POS_RCM_2] I heard: %f", d2)

def anch3(data):
  global d3
  d3_temp=data.rangeMm+0.0
  d3=d3_temp/1000
  rospy.loginfo("\n[POS_RCM_3] I heard: %f", d3)

def possessed(pub):
  global d1, d2, d3
  # Position estimation
  x1 = 1.0/22*(-math.sqrt(abs(-d2**4+2*d2**2*d3**2+242*d2**2-d3**4+242*d3**
    2-14641))-121)
  x2 = 1.0/22*(math.sqrt(abs(-d2**4+2*d2**2*d3**2+242*d2**2-d3**4+242*d3**
    2-14641))-121)
```

```
43    y = 1.0/22*(d3**2-d2**2)
44
45    try:
46      if(math.sqrt((x1-6)**2+(y)**2) > d1-tol and math.sqrt((x1-6)**2+(y)**2)
      < d1+tol):
47        rospy.loginfo("\n[POS_CALC_1] Calculated position is: x=%f, y=%f (mm)
     \nd1=%s\nd2=%s\nd3=%s", x1*1000, y*1000, d1, d2, d3)
48        pos.x = x1*1000
49        pos.y = y*1000
50#        pub.publish(pos)
51      else:
52        rospy.loginfo("\n[POS_CALC_2] Calculated position is: x=%f, y=%f (mm)
     \nd1=%s\nd2=%s\nd3=%s", x2*1000, y*1000, d1, d2, d3)
53        pos.x = x2*1000
54        pos.y = y*1000
55#        pub.publish(pos)
56        rospy.Rate(100).sleep()
57    finally:
58      pass
59
60  def true_pos(data):
61    global posx, posy
62    truepos=unpack('fff',data.data)
63    pos_err_x=(truepos[0]*1000)-pos.x
64    pos_err_y=(truepos[1]*1000)-pos.y
65    rospy.loginfo("\n[POS_EST] Real position is: (x=%s, y=%s) (mm)",truepos
      [0] ,truepos[1])
66    rospy.loginfo("\n[POS_ERR] Position estimation error is: (x=%s, y=%s) (mm
      )", pos_err_x, pos_err_y)
67#   pos.x=truepos[0]*1000
68#   pos.y=truepos[1]*1000
69#   rospy.loginfo("\n[POS_EST] Real position is: (x=%s, y=%s) (mm)",truepos
      [0] ,truepos[1])
70#   pub.publish(pos)
71    rospy.Rate(100).sleep()
72
73  def pos_est_main():
74#   time.sleep(5)
75
76    rospy.init_node('est_pos', anonymous=True)
77    rospy.Subscriber("rcm1", Range, anch1)
78    rospy.Subscriber("rcm2", Range, anch2)
79    rospy.Subscriber("rcm3", Range, anch3)
80    pub_pos = rospy.Publisher('Position_rcm', Position, queue_size=1)
81#   rospy.Subscriber("vrep/position", String, true_pos, pub_pos)
82    rospy.Subscriber("vrep/position", String, true_pos)
83    while not rospy.is_shutdown():
84      possessed(pub_pos)
85    rospy.Rate(100).sleep()
86
87    rospy.spin()
88
89  if __name__ == '__main__':
90    pos_est_main()
```

rcm.py

```
#!/usr/bin/env python

"This script listens to the streaming of distance to both cars on every
    anchor in V-REP."

import rospy
from struct import unpack
from std_msgs.msg import String
from estimate_position.msg import Position
from estimate_position_vrep.msg import Range

msg = Range()

def callback1(data, pub):
  rcm=unpack('fffffff', data.data)
  rospy.loginfo("\n[RCM_POS_1] Anchor 1 position is: x=%f, y=%f", rcm[0],
    rcm[1])
  msg.anchorId=1
  msg.rangeMm = int(round(rcm[6]*1000))
  rospy.loginfo("\n[RCM_DIST_1] Distance from anchor 1 to car is %f", msg.
    rangeMm)
  pub.publish(msg)
  rospy.Rate(100).sleep()

def callback2(data, pub):
  rcm=unpack('fffffff', data.data)
  rospy.loginfo("\n[RCM_POS_2] Anchor 2 position is: x=%f, y=%f", rcm[0],
    rcm[1])
  msg.anchorId=2
  msg.rangeMm = int(round(rcm[6]*1000))
  rospy.loginfo("\n[RCM_DIST_2] Distance from anchor 2 to car is %f", msg.
    rangeMm)
  pub.publish(msg)
  rospy.Rate(100).sleep()

def callback3(data, pub):
  rcm=unpack('fffffff', data.data)
  rospy.loginfo("\n[RCM_POS_3] Anchor 3 position is: x=%f, y=%f", rcm[0],
    rcm[1])
  msg.anchorId=3
  msg.rangeMm = int(round(rcm[6]*1000))
  rospy.loginfo("\n[RCM_DIST_3] Distance from anchor 3 to car is %f", msg.
    rangeMm)
  pub.publish(msg)
  rospy.Rate(100).sleep()

def rcm_vrep():
  rospy.init_node('rcm_data', anonymous=True)

  pub1 = rospy.Publisher('rcm1', Range, queue_size=1)
```

```
44    pub2 = rospy.Publisher('rcm2', Range, queue_size=1)
45    pub3 = rospy.Publisher('rcm3', Range, queue_size=1)
46
47    rospy.Subscriber("/vrep/ank1_distance", String, callback1, pub1)
48    rospy.Subscriber("/vrep/ank2_distance", String, callback2, pub2)
49    rospy.Subscriber("/vrep/ank3_distance", String, callback3, pub3)
50
51    rospy.spin()
52
53  if __name__ == '__main__':
54    rcm_vrep()
```

get_imu_vrep

IMUData.msg

```
Header header
float64 gyroX
float64 gyroY
float64 gyroZ
float64 accX
float64 accY
float64 accZ
float64 magX
float64 magY
float64 magZ
```

get_imu_vrep.py

```
#!/usr/bin/env python

import rospy
from struct import unpack
from std_msgs.msg import String
from get_imu.msg import IMUData
from math import pi

imu = IMUData()

def imu_parser(data,pub):
  floats = unpack('fff', data.data)
  rospy.loginfo(rospy.get_name() + "\n[IMU] accX=%s, accY=%s, gyroZ=%s",
    floats[0], floats[1], floats[2])

  imu.header.stamp.secs = rospy.get_rostime().secs
  imu.header.stamp.nsecs = rospy.get_rostime().nsecs
  imu.accX = floats[0]
  imu.accY = floats[1]
  imu.gyroZ = floats[2]*180/pi
  pub.publish(imu)
  rospy.Rate(100).sleep()

def imu_vrep():
  rospy.init_node("imu_node", anonymous=True)
  pub = rospy.Publisher('imu_data', IMUData, queue_size=1)
  rospy.Subscriber("/vrep/gyroAcc", String, imu_parser, pub)
  rospy.spin()

if __name__ == '__main__':
  imu_vrep()
```

get_uss_vrep

srf08_msg.msg

```
float32 sensor_0
float32 sensor_1
```

get_uss_vrep.py

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import *
from get_uss.msg import srf08_msg

# Listen from Topic SRF08_R.
def callbackR(data, pub):
  rospy.loginfo(rospy.get_caller_id() + "\n[SRF08_R] Distance=%s (m)", data
    .data)
  srf08_R = data.data
  msg = srf08_msg()
  msg.sensor_0 = srf08_R
#  msg.header.stamp = rospy.get_rostime()
  pub.publish(msg)

# Listen from Topic SRF08_L.
def callbackL(data, pub):
  rospy.loginfo(rospy.get_caller_id() + "\n[SRF08_L] Distance=%s (m)", data
    .data)
  srf08_L = data.data
  msg = srf08_msg()
  msg.sensor_1 = srf08_L
#  msg.header.stamp = rospy.get_rostime()
  pub.publish(msg)

def init():
  pub = rospy.Publisher('uss_data', srf08_msg, queue_size=1)
  rospy.init_node('get_uss_vrep', anonymous=True)
  rospy.Subscriber("/vrep/SRF08_R", Float32, callbackR, pub)
  rospy.Subscriber("/vrep/SRF08_L", Float32, callbackL, pub)
  rospy.spin()

if __name__ == '__main__':
  init()
```

joy_to_mc_cmds
joy_to_cmds.py

```
#!/usr/bin/env python
# Software License Agreement (BSD License)
#
# Copyright (c) 2014, Viktor Nilsson.
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
#  * Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
#  * Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
#    with the distribution.
#  * Neither the name of Willow Garage, Inc. nor the names of its
#    contributors may be used to endorse or promote products derived
#    from this software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
# Revision $Id$

import rospy
from std_msgs.msg import String
from sensor_msgs.msg import Joy
from ackermann_msgs.msg import AckermannDrive

## axes[4] right trigger
## axes[0] left stick left/right
## buttons[1] A
def callback(data, pub):
    rospy.loginfo(rospy.get_name() + "I heard this: \n buttons[1]==%f, axes
    [0]==%f, axes[4]==%f", data.buttons[1], data.axes[0], data.axes[5]);

    if data.buttons[0] == 0: direction = 1
    else: direction = -1
    degree = data.axes[0]*22
    speed = (1+speedToSpeed(data.axes[5]))/2*direction

```

```
52      rospy.loginfo("Setting degree to %f\n", degree)
53      rospy.loginfo("Setting speed to %f\n", speed)
54
55      # Create AckermannDrive msg
56      drive = AckermannDrive()
57      drive.steering_angle = degree
58      drive.speed = speed
59
60      # Publish under topic 'mc_cmds'
61      pub.publish(drive)
62
63
64  def init():
65      # Publish under topic 'mc_cmds'
66      pub = rospy.Publisher('mc_cmds', AckermannDrive, queue_size=1)
67
68      # Subscribs to 'joy'
69      rospy.Subscriber("joy", Joy, callback, pub)
70
71
72  # Converts a keypress (between 1.0 and -1.0) to between 0.0 and 1.0.
73  # If speed is exactly 50% (speed == 0.0), discard the value because
74  # that is probably a fault in joy.
75  def speedToSpeed(speed):
76    # Sometimes joy_node says that axis[4] is 0 (even though it should be 1.0
77    # when not pressed)
78    if speed == 0: return 0
79
80    # For convenience, we want a value between 0.0 and 1.0
81    speed = -((speed-1)/2)
82    if speed < 0.25:
83      return 0
84    return speed
85
86
87  # Converts from joystick angle to steering angle.
88  # joy_node outputs angle in -1.0 (full left) to 1.0 (full right)
89  # The angle are "not really" in degrees.
90  # +-70 (assuming middle is 114) is the last value that don't give that
        weird
91  # noice.
92  #def joyAngleToDegree(angle):
93  #    degree = 70*angle
94  #    return degree
95
96
97  if __name__ == '__main__':
98      rospy.init_node('joy_listener_talker', anonymous=False)
99      init()
100     rospy.spin()
```

Appendix B

gulliver_car_script

```
if (sim_call_type==sim_childscriptcall_initialization) then

  local pathType='cirkel' -- Alternativ: cirkel, cirkel2, eight
  local moduleName=0
  local moduleVersion=0
  local index=0
  local pluginNotFound=true
  while moduleName do
    moduleName,moduleVersion=simGetModuleName(index)
    if (moduleName=='Ros') then
      pluginNotFound=false
    end
    index=index+1
  end

  if (pluginNotFound) then
    controllerID=-1
  -- Display an error message if the plugin was not found:
    simDisplayDialog('Error','ROS plugin was not found.&&nSimulation will
     not run properly',sim_dlgstyle_ok,false,nil
     ,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
  else
  -- Motor controller input.
    controllerID=simExtROS_enableSubscriber('motor',1,
    simros_strmcmd_set_string_signal, -1, -1,'mcValuesSignal')
  -- Determines which static map to use.
    simSetStringSignal('pathTypeSignal', pathType)
    pathTypeID=simExtROS_enablePublisher('pathType',1,
    simros_strmcmd_get_string_signal, -1, -1,'pathTypeSignal',10)
  -- Exports IMU data to ROS.
    simSetStringSignal('gyroAccSignal', simPackFloats({0,0,0}))
    gyroAccID=simExtROS_enablePublisher('gyroAcc', 1,
    simros_strmcmd_get_string_signal, -1, -1,'gyroAccSignal')
  -- The cars true position in V-REP.
    simSetStringSignal('positionSignal', simPackFloats({0,0,0}))
    positionID=simExtROS_enablePublisher('position', 1,
    simros_strmcmd_get_string_signal, -1, -1,'positionSignal')
  -- Published distance traveled and tachometer ticks.
    distravID=simExtROS_enablePublisher('distance_traveled',1,
    simros_strmcmd_get_float_signal,-1,-1,'distance_traveled')
    ticksID=simExtROS_enablePublisher('ticks',1,
    simros_strmcmd_get_integer_signal,-1,-1,'tacho_ticks')
    angleID=simExtROS_enablePublisher('steering_angle',1,
    simros_strmcmd_get_float_signal,-1,-1,'steering_angle_signal')
    simAddStatusbarMessage("Distance traveled ID: "..distravID)
    simAddStatusbarMessage("Tachometer ID: "..ticksID)
    simAddStatusbarMessage("Steering angle ID: "..angleID)
  -- Publishes anchor distances to ROS.
    simSetStringSignal('rcm_signal1', simPackFloats({0,0,0,0,0,0,0}))
    simSetStringSignal('rcm_signal2', simPackFloats({0,0,0,0,0,0,0}))
    simSetStringSignal('rcm_signal3', simPackFloats({0,0,0,0,0,0,0}))
```

```
43      rcm1=simExtROS_enablePublisher('ank1_distance',1,
        simros_strmcmd_get_string_signal,-1,-1,'rcm_signal1')
44      rcm2=simExtROS_enablePublisher('ank2_distance',1,
        simros_strmcmd_get_string_signal,-1,-1,'rcm_signal2')
45      rcm3=simExtROS_enablePublisher('ank3_distance',1,
        simros_strmcmd_get_string_signal,-1,-1,'rcm_signal3')
46  end
47
48    gyroCommunicationTube=simTubeOpen(0,'gyroData'..simGetNameSuffix(nil),1)
49    accelCommunicationTube=simTubeOpen(0,'accelerometerData'..
        simGetNameSuffix(nil),1)
50
51    car=simGetObjectHandle('Gulliver_model')
52    steeringLeft=simGetObjectHandle('steering_frontLeft')
53    steeringRight=simGetObjectHandle('steering_frontRight')
54    motorLeft=simGetObjectHandle('motor_frontLeft')
55    motorRight=simGetObjectHandle('motor_frontRight')
56    sensorRight=simGetObjectHandle('SRF08_R')
57    sensorLeft=simGetObjectHandle('SRF08_L')
58    anchor1=simGetObjectHandle('Ankare')
59    anchor2=simGetObjectHandle('Ankare0')
60    anchor3=simGetObjectHandle('Ankare1')
61    rcm=simGetObjectHandle('Accelerometer_mass')
62    desiredSteeringAngle=0
63    desiredWheelRotSpeed=0
64    steeringAngleDx=2*math.pi/180
65    wheelRotSpeedDx=20*math.pi/180
66    d=0.13575 -- 2*d=distance between left and right wheels
67    l=0.33 -- l=distance between front and read wheels
68    distrav=0 -- distance traveled
69    lastpos=simGetObjectPosition(car,-1)
70
71    -- DATA LOGS
72    distance=io.open("logs/distance.log","w+")
73    distance:write("index\t\distance(m)\ttacho_ticks\n")
74    imu=io.open("logs/imu.log","w+")
75    imu:write("index\taccX\taccY\tgyroZ\n")
76    rcm1=io.open("logs/rcm1.log","w+")
77    rcm1:write("index\tx_anchor\ty_anchor\tz_anchor\tx_car\ty_car\tz_car\
        tdistance(m)\n")
78    rcm2=io.open("logs/rcm2.log","w+")
79    rcm2:write("index\tx_anchor\ty_anchor\tz_anchor\tx_car\ty_car\tz_car\
        tdistance(m)\n")
80    rcm3=io.open("logs/rcm3.log","w+")
81    rcm3:write("index\tx_anchor\ty_anchor\tz_anchor\tx_car\ty_car\tz_car\
        tdistance(m)\n")
82    poslog=io.open("logs/position.log","w+")
83    poslog:write("index\tx\ty\n")
84    motor=io.open("logs/motor.log","w+")
85    motor:write("index\tspeed\tangle\n")
86    logindex=1
87
88  end
89
90
```

```
91  if (sim_call_type==sim_childscriptcall_actuation) then
92
93    -- RCM Measurements
94    res,distance1=simCheckDistance(anchor1,rcm,0)
95    res,distance2=simCheckDistance(anchor2,rcm,0)
96    res,distance3=simCheckDistance(anchor3,rcm,0)
97    rcm_sig1=simSetStringSignal('rcm_signal1', simPackFloats(distance1))
98    rcm_sig2=simSetStringSignal('rcm_signal2', simPackFloats(distance2))
99    rcm_sig3=simSetStringSignal('rcm_signal3', simPackFloats(distance3))
100   rcm1:write(logindex)
101   rcm1:write("\t"..distance1[1].."\t"..distance1[2].."\t"..distance1[3].."\
        t"..distance1[4].."\t"..distance1[5].."\t"..distance1[6].."\t"..
        distance1[7])
102   rcm1:write('\n')
103   rcm2:write(logindex)
104   rcm2:write("\t"..distance2[1].."\t"..distance2[2].."\t"..distance2[3].."\
        t"..distance2[4].."\t"..distance2[5].."\t"..distance2[6].."\t"..
        distance2[7])
105   rcm2:write('\n')
106   rcm3:write(logindex)
107   rcm3:write("\t"..distance3[1].."\t"..distance3[2].."\t"..distance3[3].."\
        t"..distance3[4].."\t"..distance3[5].."\t"..distance3[6].."\t"..
        distance3[7])
108   rcm3:write('\n')
109
110   -- Distance traveled+tachometer
111   currpos=simGetObjectPosition(car,-1)
112   dt=math.sqrt(math.pow(currpos[1]-lastpos[1],2)+math.pow(currpos[2]-
        lastpos[2],2))
113   distrav=distrav+dt
114   lastpos=currpos
115   distsig=simSetFloatSignal('distance_traveled',distrav)
116   ticks=math.ceil(distrav*450)
117   ticksig=simSetIntegerSignal('tacho_ticks',ticks)
118   steersig=simSetFloatSignal('steering_angle_signal',desiredSteeringAngle)
119   distance:write(logindex)
120   distance:write("\t"..distrav.."\t"..ticks)
121   distance:write("\n")
122
123   -- Gyro and Acc values
124   data=simTubeRead(gyroCommunicationTube)
125   if (data) then
126     angularVariations=simUnpackFloats(data)
127   end
128   data=simTubeRead(accelCommunicationTube)
129   if (data) then
130       acceleration=simUnpackFloats(data)
131   end
132   if (angularVariations and acceleration) then
133     gyroAccFloats={acceleration[1], acceleration[2], angularVariations[3]}
134     imu:write(logindex-1)
135     imu:write("\t"..acceleration[1].."\t"..acceleration[2].."\t"..
        angularVariations[3])
136     imu:write("\n")
137     gyroAcc=simPackFloats(gyroAccFloats)
```

```
138     simSetStringSignal('gyroAccSignal', gyroAcc)
139   end
140
141   -- Absolute position
142   position=simGetObjectPosition(rcm, -1)
143   poslog:write(logindex)
144   poslog:write("\t"..position[1].."\t"..position[2])
145   poslog:write("\n")
146   positionString=simPackFloats(position)
147   simSetStringSignal('positionSignal', positionString)
148
149   -- EXTERNAL CONTROL OF THE CAR.
150   if (controllerID~=-1) then
151     mcValues=simGetStringSignal('mcValuesSignal')
152     if(mcValues) then
153       mcValuesFloats=simUnpackFloats(mcValues)
154       --simDisplayDialog('Floats ',mcValuesFloats[1],sim_dlgstyle_ok,NULL,
      NULL,NULL,NULL)
155     end
156     if (mcValuesFloats[1]>0) then
157       -- GO FORWARD
158       desiredWheelRotSpeed=desiredWheelRotSpeed+wheelRotSpeedDx*
      mcValuesFloats[1]
159       if (desiredWheelRotSpeed>=5) then
160         desiredWheelRotSpeed=5
161       end
162     end
163     if (mcValuesFloats[1]==0 and desiredWheelRotSpeed>0) then
164         desiredWheelRotSpeed=desiredWheelRotSpeed-wheelRotSpeedDx/2
165         if (desiredWheelRotSpeed<wheelRotSpeedDx/2) then
166           desiredWheelRotSpeed=0
167         end
168     end
169
170     if (mcValuesFloats[1]<0) then
171       -- GO BACK
172       desiredWheelRotSpeed=desiredWheelRotSpeed+wheelRotSpeedDx*
      mcValuesFloats[1]
173       if (desiredWheelRotSpeed<=-1) then
174         desiredWheelRotSpeed=-1
175       end
176     end
177     if (mcValuesFloats[1]==0 and desiredWheelRotSpeed<0) then
178       desiredWheelRotSpeed=desiredWheelRotSpeed+wheelRotSpeedDx/2
179       if (-desiredWheelRotSpeed<-wheelRotSpeedDx/2) then
180         desiredWheelRotSpeed=0
181       end
182     end
183
184     if (mcValuesFloats[2]>0) then
185       -- TURN LEFT
186       desiredSteeringAngle=desiredSteeringAngle+steeringAngleDx*
      mcValuesFloats[2]/22
187       if (desiredSteeringAngle>22*math.pi/180*mcValuesFloats[2]/22) then
188         desiredSteeringAngle=22*math.pi/180*mcValuesFloats[2]/22
```

```
189         end
190       end
191       if (mcValuesFloats[2]==0 and desiredSteeringAngle>0) then
192         desiredSteeringAngle=desiredSteeringAngle-steeringAngleDx
193 --        if (desiredSteeringAngle<steeringAngleDx/2) then
194 --          steeringAngleDx=0
195 --        end
196       end
197
198       if (mcValuesFloats[2]<0) then
199         -- TURN RIGHT
200         desiredSteeringAngle=desiredSteeringAngle+steeringAngleDx*
      mcValuesFloats[2]/22
201         if (desiredSteeringAngle<+22*math.pi/180*mcValuesFloats[2]/22) then
202           desiredSteeringAngle=+22*math.pi/180*mcValuesFloats[2]/22
203         end
204       end
205       if (mcValuesFloats[2]==0 and desiredSteeringAngle<0) then
206         desiredSteeringAngle=desiredSteeringAngle+steeringAngleDx
207 --        if (-desiredSteeringAngle<-steeringAngleDx/2) then
208 --          steeringAngleDx=0
209 --        end
210       end
211
212       simClearStringSignal('mcValuesTemp')
213     else
214
215       message,auxiliaryData=simGetSimulatorMessage()
216       while message~=-1 do
217         if (message==sim_message_keypress) then
218           if (auxiliaryData[1]==2007) then
219             -- up key
220             desiredWheelRotSpeed=desiredWheelRotSpeed+wheelRotSpeedDx
221           end
222           if (auxiliaryData[1]==2008) then
223             -- down key
224             desiredWheelRotSpeed=desiredWheelRotSpeed-wheelRotSpeedDx
225           end
226           if (auxiliaryData[1]==2009) then
227             -- left key
228             desiredSteeringAngle=desiredSteeringAngle+steeringAngleDx
229             if (desiredSteeringAngle>22*math.pi/180) then
230               desiredSteeringAngle=22*math.pi/180
231             end
232           end
233           if (auxiliaryData[1]==2010) then
234             -- right key
235             desiredSteeringAngle=desiredSteeringAngle-steeringAngleDx
236             if (desiredSteeringAngle<-22*math.pi/180) then
237               desiredSteeringAngle=-22*math.pi/180
238             end
239           end
240         end
241         message,auxiliaryData=simGetSimulatorMessage()
242       end
```

```
243     end
244
245     -- We handle the front left and right wheel steerings (Ackermann steering
          ):
246     steeringAngleLeft=math.atan(l/(-d+l/math.tan(desiredSteeringAngle)))
247     steeringAngleRight=math.atan(l/(d+l/math.tan(desiredSteeringAngle)))
248     simSetJointTargetPosition(steeringLeft,steeringAngleLeft)
249     simSetJointTargetPosition(steeringRight,steeringAngleRight)
250
251     -- We take care of setting the desired wheel rotation speed:
252     simSetJointTargetVelocity(motorLeft,desiredWheelRotSpeed)
253     simSetJointTargetVelocity(motorRight,desiredWheelRotSpeed)
254
255     motor:write(logindex)
256     motor:write("\t"..desiredWheelRotSpeed.."\t"..desiredSteeringAngle)
257     motor:write("\n")
258
259     logindex=logindex+1
260
261 end
262
263 if (sim_call_type==sim_childscriptcall_sensing) then
264
265
266 end
267
268
269 if (sim_call_type==sim_childscriptcall_cleanup) then
270
271     io.close(imu)
272     io.close(distance)
273     io.close(poslog)
274     io.close(motor)
275     io.close(rcm1)
276     io.close(rcm2)
277     io.close(rcm3)
278
279 end
```

srf08_script

```
if (sim_call_type==sim_childscriptcall_initialization) then

  noseSensor=simGetObjectHandle('SRF08_L')
  simSetFloatSignal('distanceSignal_L', 0)

  local moduleName=0
  local moduleVersion=0
  local index=0
  local pluginNotFound=true
  while moduleName do
    moduleName,moduleVersion=simGetModuleName(index)
    if (moduleName=='Ros') then
      pluginNotFound=false
    end
    index=index+1
  end

  if (pluginNotFound==false) then
    sensorID=simExtROS_enablePublisher('SRF08_L',1,
   simros_strmcmd_get_float_signal,-1,-1,'distanceSignal_L')
  end

  -- DATA LOG
  uss_L=io.open("logs/uss_L.log","w")
  uss_L=io.open("logs/uss_L.log","a")
  uss_L:write("index\tdistance(m)\n")
  logindex=1

end

if (sim_call_type==sim_childscriptcall_actuation) then

end


if (sim_call_type==sim_childscriptcall_sensing) then
  result,distance=simReadProximitySensor(noseSensor)
  if(result==1) then
    simSetFloatSignal('distanceSignal_L', distance)
  else
    distance=0
    simSetFloatSignal('distanceSignal_L', 0)
  end
  uss_L:write(tostring(logindex))
  uss_L:write("\t")
  uss_L:write(tostring(distance))
  uss_L:write("\n")
  logindex=logindex+1
end


if (sim_call_type==sim_childscriptcall_cleanup) then
  io.close(uss_L)
```

```
53 end
```