Teknisk dokumentation

Hampus Westerberg
Joel Wiklund
Tomasz Mazurek
Carl Liljeberg
Mohammad Rajabi
Anton Lund
Simon Svahn

12 december 2024

Version 0.1

Autonom Truck

Status

Granskad	ALLA	2024-12-11
Godkänd	Gustav Zetterqvist	2024-xx-xx

Projektidentitet

Grupp E-post: hamwe392@student.liu.se

Beställare: Gustav Zetterqvist, Linköpings universitet
Tfn: 013-28 19 94
E-post: gustav.zetterqvist@liu.se

Kund: Johan Lindell, Toyota Material Handling
E-post: johan.lindell@toyota-industries.eu

Handledare 1: Sebastian Karlsson (ISY)
E-post: sebastian.karlsson@liu.se

Handledare 2: Oskar Bergkvist (Toyota Material Handling)
E-post: oskar.bergkvist@toyota-industries.eu

Handledare 3: Andreas Bergström (Toyota Material Handling)
E-post: Andreas.Bergstrom.ext@toyota-industries.eu

Kursansvarig: Daniel Axehill, Linköpings universitet
Tfn: +4613284042
E-post: daniel.axehill@liu.se

Projektdeltagare

Namn	**Ansvar**	**E-post**
Hampus Westerberg	Projektledare (PL)	hamwe392@student.liu.se
Joel Wiklund	Testansvarig (TA)	joewi329@student.liu.se
Carl Liljeberg	Informationsansvarig (IA)	carli426@student.liu.se
Mohammad Rajabi	Dokumentansvarig (DOK)	mohra735@student.liu.se
Anton Lund	Mjukvaruansvarig (MA)	antlu106@student.liu.se
Simon Svahn	Designansvarig (DES)	simsv926@student.liu.se
Tomasz Mazurek	Sekreterare (SEK)	tomma870@student.liu.se

INNEHÅLL

1 Inledning 1
2 Systembeskrivning 1
3 Truckens specifikationer 2
4 Kartläggning 3
4.1 LIDAR 3
4.2 Kartläggning 4
4.3 Kostnadkartan 5
5 Hinderdetektion och målnodssättning 6
5.1 Systeminteraktioner 6
5.2 För- och nackdelar 6
6 Ruttplanering 7
6.1 Tillståndsmiljön 7
6.2 Ruttplaneringsalgoritm 7
6.3 Systeminteraktioner 9
6.4 För- och nackdelar 9
7 Ruttbyte 10
7.1 Systeminteraktioner 10
7.2 För- och nackdelar 10
8 Ruttföljare 11
8.1 Kinematisk modell 11
8.2 Funktionalitet 12
8.3 Nyckelprinciper 12
8.4 Systeminteraktioner 13
8.5 Fördelar och Begränsningar 13
8.6 Användning i Projektet 13
9 Simuleringsmiljö 14
9.1 Simuleringsverktyg 14
9.2 Miljö och hinder 14
9.3 Koordinatsystem och Transformationer 14
9.4 Sensorsimulering 14
9.5 Truckmodell 15
10 Gränssnitt 16
10.1 ROS2 16
10.2 Noder och *topics* 16
10.3 Användargränssnitt 18
11 Referenser 21

DOKUMENTHISTORIK

Version	Datum	Utförda förändringar	Utförda av	Granskad
0.1	2024-12-11	Första utkastet	Alla	Alla

1 INLEDNING

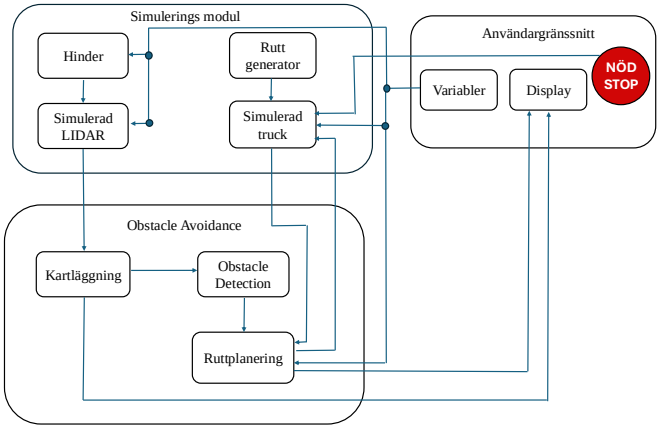
Detta dokument är en teknisk dokumentation för ett projekt i kursen TSRT10. Projektet är framtaget i samarbete med Toyota Material Handling, härefter benämnt Toyota, och syftar till att undersöka möjligheterna för hinderundvikande eller *obstacle avoidance*.

Toyota utvecklar och säljer i dagsläget självkörande gaffeltruckar för arbete i lager och liknande miljöer. I nuläget kan dessa endast köra längs förutbestämda banor i lagret och stannar upp när de möter ett hinder och kräver då hjälp från en anställd. Projektets syfte är att utveckla ett system som låter truckarna själva hitta alternativa rutter runt hindret. Systemet ska utvecklas och testas i en simuleringsmiljö.

2 SYSTEMBESKRIVNING

Det simulerade systemet består av tre huvudmoduler, där varje modul innehåller flera undermoduler. För översikt se Figur 1. Kommunikationen mellan modulerna samt undermodulerna sker via *topics* i *Robotic Operating System* (ROS) som är ett flexibelt ramverk för att hantera meddelandeutbyte i robotiksystem. Se Avsnitt 9 för mer information om ROS. Nedan ges en kort beskrivning av huvudmodulerna och deras syften:

- **Simuleringsmodulen:** En virtuell simuleringsvärld där hinder, LIDAR-data, gaffeltrucken och dess ursprungliga rutt modelleras och simuleras.
- **Användargränssnitts:** En mjukvara som möjliggör konfigurering av truckens dimensioner, visar en karta över den kartlagda omgivningen samt konfigurerar miljö- och hinderparametrar.
- **Obstacle avoidance-modulen:** Denna modul utför kartläggning av truckens omgivning med hjälp av LIDAR-data som skickas från simuleringsmiljön. Kartan postprocessas och används sedan till detektion av hinder och planering av ny rutt runt hindret.

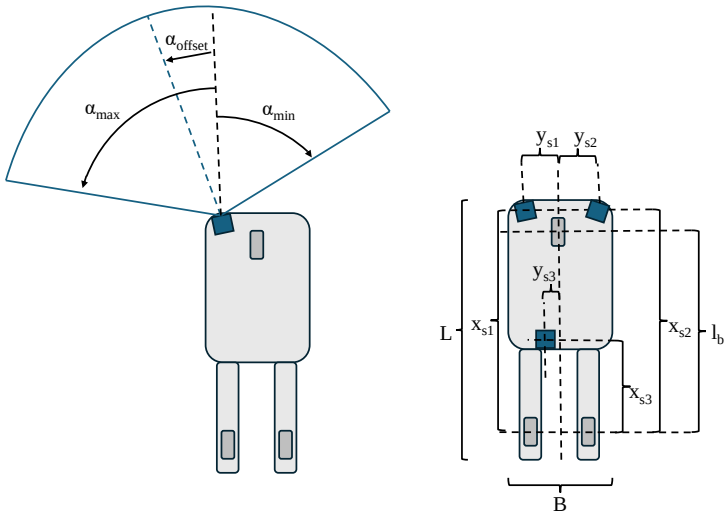


Figur 1: Generell översikt över det simulerade systemet och kommunikationen mellan modulerna (indikerat med blåa pilar).

3 TRUCKENS SPECIFIKATIONER

Den verkliga trucken tillverkas av Toyota och är försedd med LIDAR-enheter för att uppfatta omgivningen samt detektera hinder. Truckarna kommer i olika storlekar, vilket lägger krav på parametrisering i mjukvaran. Truckens övergripande mått går att se i Figur 2. Parametrarnas betydelse går att se i Tabell 1.

För simuleringarna i detta projekt valdes att skala ned modellen till en något förenklad variant. Detta bestod framför allt i att bara en lidar användes, som placerades längs truckens centrumlinje.



Figur 2: En bild över relevanta mått och storheter på trucken

Tabell 1: Visar truckens relevanta mått

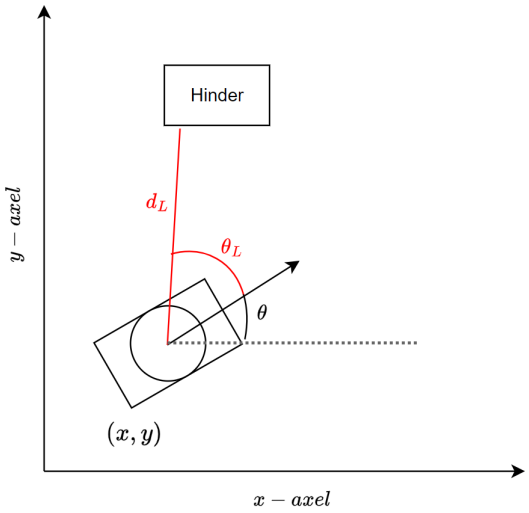
Parameter	Beskrivning
α_{max}	Sensorfältets maximala vinkel
α_{min}	Sensorfältets minimala vinkel
α_{offset}	Vinkel mellan fordonets framåtriktning och sensorns centrumlinje
y_s	Sensorns förskjutning från truckens centrumlinje
L	Truckens totala längd
B	Truckens totala bredd
l_b	Avståndet mellan truckens fram- och bak-hjul
$x_{s1,s2,s3}$	Alla tre sensorers förskjutning i x-led
$y_{s1,s2,s3}$	Alla tre sensorers förskjutning i y-led

4 KARTLÄGGNING

I det här avsnittet beskrivs hur sensordata från LIDAR används för att kartlägga gaffeltruckens omgivande miljö.

4.1 LIDAR

En LIDAR-enhet med 360 graders synvinkel används i *Gazebo*, en simuleringsmjukvara, för att simulera punktmoln som kartläggningsmodulen använder för att kartlägga truckens omgivning. Varje mätning från LIDAR-enheten representeras som (d_L, θ_L), där d_L är avståndet och θ_L är vinkeln till mätpunkten, se Figur 3. Denna data tillsammans med truckens position och orientering, betecknad som (x, y, θ) i Figur 3, används för att uppdatera sannolikheten av varje ruta i ett diskret rutnät som utgör kartan.

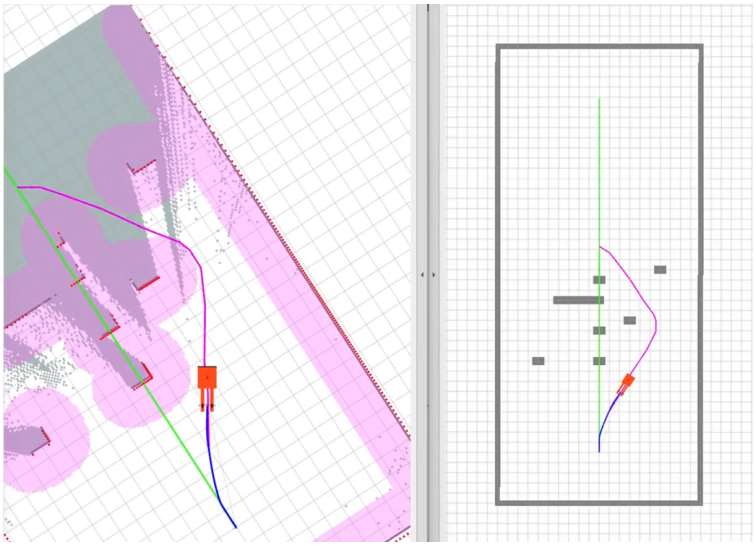


Figur 3: Truckens position och orientering i globala koordinatsystemet, tillsammans med en LIDAR-mätning (markerad i rött).

4.2 Kartläggning

För att skapa den diskreta kartan på ett smidigt sätt används *SLAM Toolbox* vilket är ett verktyg i ROS2. Verktygets indata består av rådata från LIDAR-sensorn samt truckens position och orientering. Kartans upplösning valdes till 0,1 meter, där varje ruta kan anta värden mellan -1 och 100. Här innebär -1 att rutan är helt okänd, medan värden mellan 0 och 100 representerar sannolikheten för att rutan är upptagen av ett objekt. I användargränssnittet visualiseras denna sannolikhet med hjälp av färgskalor, där svart indikerar en hög sannolikhet och vit en låg sannolikhet för att rutan är upptagen. Figur 4 visar hur den diskreta kartan ser ut i användargränssnittet.

Tyvärr så har *SLAM Toolbox* begränsat oss till användning av endast ett LIDAR. För att kunna implementera tre sensorer behöver man implementera egen kod för hela kartläggningen. Ett försök på detta har gjort i python, men blivit övergiven på grund av att den gjorde kartläggningen alldeles för långsam.

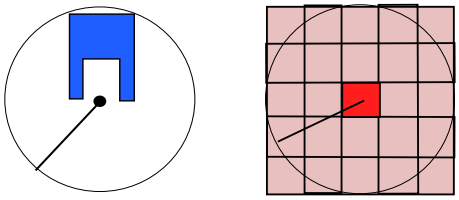


Figur 4: Upplägget av kartan (höger) och den diskreta kartan som trucken ser (vänster). De rosa färgade rutor är del av kostnadkartan, mer om det i kapitel 4.3.

4.3 Kostnadkartan

För att förhindra att trucken kolliderar med föremål skapas en alternativ karta med ett utökat otillåtet område, vilket används av ruttplaneringsalgoritmen. Algoritmen tar den ursprungliga kartan och expanderar de otillåtna områdena för att säkerställa att den planerade rutten undviker kollisioner. För att skapa dessa utökade områden och kontrollera truckens placering genomförs flera steg.

För att expandera kartan genereras en kernel, som är en liten bild. Denna kernel består av en cirkulär matris av ettor, där cirkelns radie motsvarar det värde med vilket kartan ska expandera. Sedan sker en "effektiviserad faltning" där cirkeln adderas på varje position där det finns ett hinder. Till sist sätts alla värden större än ett till ett. På detta sätt skapas en karta där de upptagna rutorna är expanderad lika mycket i alla riktningar. Expansionsradien är det längsta avståndet från rotationscentrum till truckens hörn, se den vänstra illustrationen i Figur 5. En säkerhetsmarginal läggs också till expansionsradien. Det säkerställer att trucken inte kommer att kollidera med hinder oberoende av hur den är vriden. Det expanderande området är de ljusrosa markering i Figur 4.



Figur 5: Expansion av det otillåtna området när en cirkel används för att omsluta trucken.

5 HINDERDETEKTION OCH MÅLNODSSÄTTNING

För att möjliggöra kontinuerlig omplanering så utvärderas den rutt som trucken i nuläget följer varje gång som kostnadskartan uppdateras. Rutten utvärderas genom att successivt kolla igenom positionerna på rutten, med start från truckens nuvarande position och framåt. Varje position jämförs mot kostnadskartan, och om en position visar sig vara blockerad indikeras ett hinder, vilket kräver att en alternativ rutt planeras. Målpositionen väljs genom att gå igenom originalrutten baklänges tills en position är ockuperad, den positionen är den bakre änden på hindret utifrån den information som finns tillgänglig från kartan. Därefter väljs målpositionen några steg bortom detta hinder för att underlätta en smidigare återanslutning till originalrutten.

5.1 Systeminteraktioner

Hinderdetektionen och målnodsättningen beror på indata från ett flertal ROS2-topics:

- **Prenumerationer:**
  - `/current_path`: Innehåller den nuvarande rutt som följs.
  - `/orig_path`: Innehåller originalrutten som trucken ska följa.
  - `/tricycle_controller/odom`: Tillhandahåller robotens aktuella position och orientering.
- **Publiceringar:**
  - `/goal_node`: Skickar målposition till ruttplaneringsalgoritmen.

5.2 För- och nackdelar

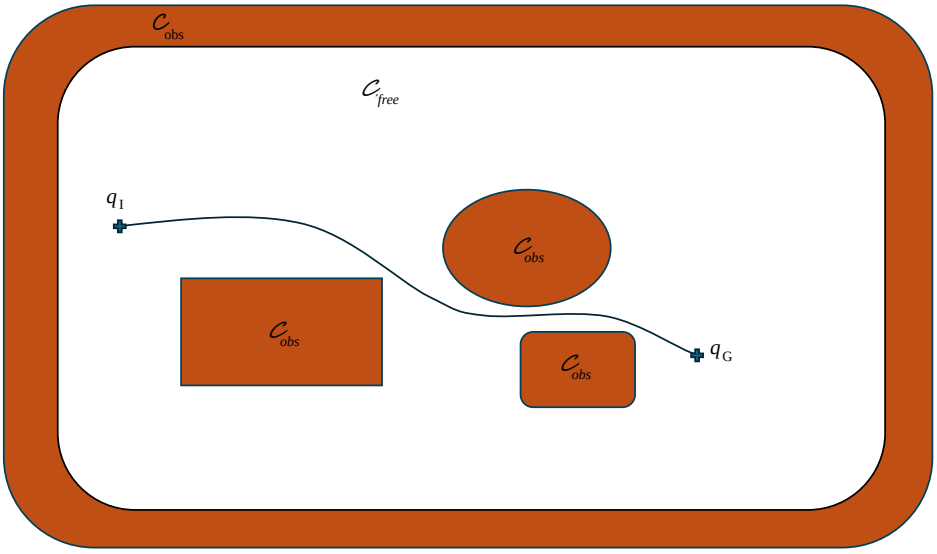
Systemet fungerar generellt mycket väl men det finns vissa situationer där den stöter på problem. Då expansionsradien på kostnadskartan är väldigt stor så kan ibland onödiga omplaneringar ske. Exempelvis om ett hinder står på sidan om rutten och egentligen inte är ivägen, så kan trucken ändå göra en omplanering. Vidare kan även valet av målposition bli problematiskt. Om det är ett hinder i slutet av kartan så kommer målnoden automatisk att väljas utanför kartan, och då kommer inte ruttplaneringen lyckas planera en rutt.

6 RUTTPLANERING

När hinderdetektering identifierar ett hinder, används en ruttplaneringsalgoritm för att skapa en alternativ rutt som kringgår hindret. Algoritmen tar hänsyn till den angivna målpositionen och kostnadskartan för att planera rutten. Nedan ges en översikt över teorin bakom ruttplanering samt en beskrivning av hur den praktiska implementeringen sker.

6.1 Tillståndsmiljön

Området som trucken kommer att verka i kallas *configuration space*, och betecknas $\mathcal{C}$. Det innehåller alla tillstånd som betecknas q, i vårt fall x koordinat och en y koordinat. $\mathcal{C}$ har delområden som innehåller hinder, $\mathcal{C}_{obs} \subseteq \mathcal{C}$. Det som blir kvar är det fria området $\mathcal{C}_{free} = \mathcal{C} \backslash \mathcal{C}_{obs}$. Målet för ruttplaneringen är att hitta en rutt från starttillståndet q_I till sluttillståndet q_G. Under rutten måste hela trucken befinna sig i $\mathcal{C}_{free}$. Ett bildexempel för tillståndsmiljön går att se i Figur 6. Ruttplaneringen använder truckens nuvarande position som starttillstånd och måltillståndet ges av hinderdetektionssystemet.



Figur 6: Tillståndsmiljön för ruttplanering

6.2 Ruttplaneringsalgoritm

Det finns flera ruttplaneringsalgoritmer som beräknar en rutt från en startnod till en slutnod. Nedan följer en förklaring av syftet med algoritmen och vilken algoritm som användes.

6.2.1 *Syftet med algoritmen*

Algritmens syfte är att hitta en rutt som avviker från den förutbestämda originalrutten runt hindret och sedan återansluter till den förutbestämda rutten. Algoritmen kommer att kontinuerligt försöka hitta en så kort sträcka som möjligt runt hindret givet övriga krav på systemet såsom säkerhetsmarginaler.

6.2.2 *RRT*

RRT står för *Rapidly exploring Random Trees*, och är en algoritm som med hjälp av slumpmässigt placerade punkter successivt bygger ett träd som täcker tillståndsmiljön utifrån startnoden. Ett träd är en graf där två godtyckligt valda noder endast kopplas samman med en väg. Algoritmen slumpar ett tillstånd i det fria utrymmet $\mathcal{C}_{free}$ som benämns q_{rand}. Sedan kollar algoritmen vilket tillstånd i trädet som befinner sig närmast som benämns $q_{nearest}$. Sedan försöker algoritmen koppla samman det närmaste tillståndet med det slumpade tillståndet. Algoritmen har dock en begränsad steglängd, så tillståndet kommer att placeras ut innan om den maximala steglängden är uppnådd. Detta tillstånd benämns q_{new}. Det nya tillståndet kopplas till det närmaste $q_{nearest}$ förutsatt att vägen mellan tillståndet inte går genom ett objekt, d.v.s. $\mathcal{C}_{obs}$. Ett exempel på detta går att se i Figur 7a och Figur 7b. Detta fortsätter tills q_{new} placeras inom en given radie från måltillståndet q_g. Då har en väg från starttillståndet q_i till q_g hittats.

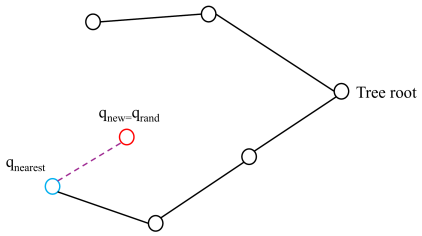
(a) RRT algoritmen utan hinder i vägen

(b) RRT algoritmen med hinder i vägen

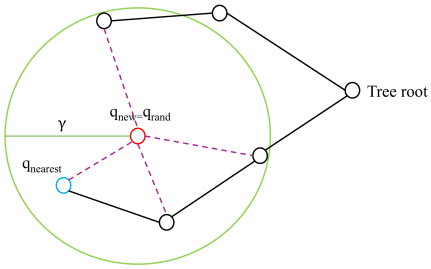
Figur 7: RRT

6.2.3 *RRT**

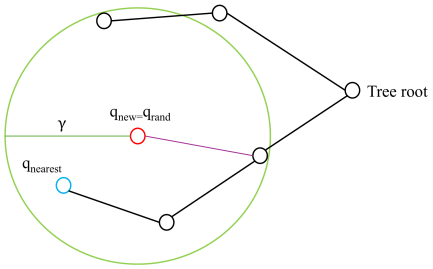
RRT* är en förbättring av RRT som försöker närma sig en optimal ruttplanering. Algoritmen fungerar som RRT fram tills det nya tillståndet placerats ut vilket går att se i Figur 8a. Istället för att koppla till det närmaste tillståndet undersöks vilket tillstånd inom en radie γ som ger lägst kostnad, i det här fallet kortast väg till q_{new} från q_i, och kopplas till den. Detta går att se i Figur 8b och Figur 8c Vidare kopplas övriga tillstånd inom radien om till det nya tillståndet om även de får en lägre kostnad. Detta går att se i Figur 8d. Allteftersom antalet iterationerna går mot oändligheten närmar sig algoritmen en optimal lösning. Det är den algoritmen som slutligen implementerades på trucken.



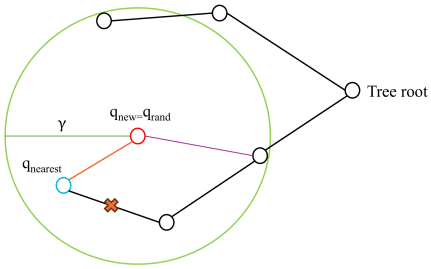
(a) Första steget i RRT* som sker på samma sätt som RRT.



(b) Andra steget i RRT* där den nya noden undersöker vilka noder som den kan kopplas till inom en radie γ.



(c) Tredje steget i RRT* där den nya noden kopplas till det tillstånd som ger den kortast väg till initial tillståndet.



(d) Fjärde steget i RRT* där tillstånd inom radien γ kopplas om via det nya tillståndet, om dess sträcka till initialtillståndet blir kortare.

Figur 8: RRT* översikt

6.3 Systeminteraktioner

Ruttplaneringsalgoritmen använder indata och utdata från flera ROS2-topics:

- **Prenumerationer:**
  - `/tricycle_controller/odom`: Tillhandahåller robotens aktuella position som används som startnod.
  - `/goal_node`: Tillhandahåller målnoden till ruttplaneringen.
- **Publiceringar:**
  - `/alternate_path`: Skickar den planerade rutten.

6.4 För- och nackdelar

Ruttplaneraren fungerar bra. Den hittar ofta den bästa rutten och är nära optimalt utifrån kartan som finns majoriteten av gångerna. Nackdelen är att den ibland kan vara lite långsam, något som kan förbättras genom att slumpa noder smartare. En klar förbättring hade även varit att introducera kinematik i planeringen vilket även skulle kunna ge positiva förbättringar på expansionsarean.

7 RUTTBYTE

När den alternativa rutten publiceras, byter trucken till den nya rutten om dess startpunkt ligger tillräckligt nära truckens aktuella position. Därefter, när trucken befinner sig inom ett visst intervall från slutpunkten av den alternativa rutten, återgår den till originalrutten. Intervallet används för att säkerställa en smidig övergång tillbaka till originalrutten. Denna logik sköts av ruttbytessystemet.

7.1 Systeminteraktioner

Bytesfunktionens indata och utdata från flera ROS2-topics:

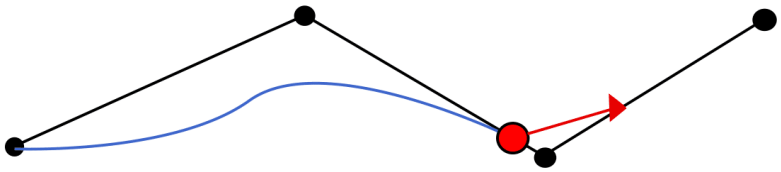
- **Prenumerationer:**
  - `/orig_path`: Innehåller originalrutten.
  - `/alternate_path`: Innehåller den planerade rutten runt hindret.
  - `/tricycle_controller/odom`: Tillhandahåller robotens aktuella position och orientering.
- **Publiceringar:**
  - `/current_path`: Skickar den nuvarande rutten som trucken ska följa.

7.2 För- och nackdelar

Systemet presterar mycket bra under majoriteten av tiden. Ett potentiellt problem uppstår dock om ruttplaneringen tar för lång tid. I sådana fall kan trucken hamna för långt från startpunkten för den alternativa rutten och därmed inte genomföra bytet.

8 RUTTFÖLJARE

Som ruttföljare används *Pure Pursuit* som är en geometribaserad algoritm. Algoritmen beräknar en styrvinkel som används för att rikta in roboten längs banan genom att sikta på en punkt framför dess aktuella position, kallad *pursuit point*, se Figur 9. Detta säkerställer att roboten rör sig jämnt och minimerar avvikelser från banan.



Figur 9: Illustration av *Pure pursuit*. Utifrån robotens aktuella position identifieras en punkt på banan som ligger på ett visst *lookahead distance*, som är den algoritmen ”jagar”.

8.1 Kinematisk modell

Trucken approximeras med en trehjuls-modell som går att se i Figur 10. I bilden är δ styrvinkeln, V_w framhjulets hastighet, V bakaxelns hastighet, θ är vinkeln mellan den globala x-axeln och bakaxelns hastighet och slutligen är (x, y) positionen på trucken som ska styras. Utifrån den kinematiska modellen tas rörelseekvationerna fram som går att se i sektionen nedan. Dessa ekvationer används av ruttföljningsalgoritmen för att konvertera den önskade linjära hastigheten och styrvinkel till en vinkelhastighet.

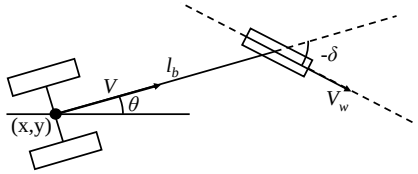
8.1.1 *Rörelseekvationer*

Den kinematiska modellen i globala koordinater tas fram med ekvation (1) och (2). Sedan tas ekvationen för vinkelhastighet genom omskrivning av dessa i ekvation (3).

$$V = V_w cos(\delta) \tag{1}$$

$$\dot{\theta} = \frac{V_w sin(\delta)}{l_b} \tag{2}$$

$$\dot{\theta} = \frac{V_s in(\delta)}{l_b cos(\delta)} \tag{3}$$



Figur 10: En bild över en kinematisk trehjuls-modell.

8.2 Funktionalitet

Pure Pursuit-controllern fungerar genom att:

1. Ta emot en planerad bana i form av en sekvens av waypoints från `/current_path`.
2. Uppdatera robotens aktuella position och orientering från odometri-data mottagna via `/tricycle_controller/odom`.
3. Identifiera en *pursuit point* längs banan som ligger på eller bortom ett förutbestämt *lookahead distance*.
4. Beräkna styrvinkeln baserat på robotens orientering och avståndet till *pursuit point*.
5. Anpassa den linjära hastigheten beroende på styrvinkeln för att säkerställa stabila svängar.
6. Använda styrvinkeln och den linjära hastigheten för att beräkna vinkelhastighet.
7. Publicera hastighetskommandon (`Twist`) till `/cmd_vel` för att justera robotens rörelse.

8.3 Nyckelprinciper

Algoritmen bygger på följande principer:

- **Lookahead distance:** Avståndet framåt längs banan där roboten ska sikta avgör hur mjukt och stabilt rörelsen blir. Ett större avstånd resulterar i smidigare rörelser men kan minska precisionen nära skarpa svängar.
- **Pursuit point:** Punkten på banan som roboten försöker nå. Denna beräknas genom att hitta den närmaste punkten som uppfyller *lookahead distance*.

- **Steering control:** Styrvinkeln beräknas genom att approximera banans kurvatur vid *pursuit point*, vilket säkerställer att robotens rörelse förblir jämn och följsam.
- **Adaptiv linjär hastighet:** Den linjära hastigheten justeras dynamiskt för att minska risken för instabilitet vid skarpa svängar. Vid större styrvinklar dämpas den linjära hastigheten med en faktor proportionell mot styrvinkeln. Detta implementeras med:

$$v_{\text{linear}} = v_{\text{max}} \cdot \frac{1}{(1.6 \cdot |\delta|)^2 + 1} \tag{4}$$

  där v_{linear} är den resulterande linjära hastigheten, v_{max} är den maximala hastigheten, och δ är styrvinkeln.

8.4 Systeminteraktioner

Regulators beteende styrs av indata och utdata från flera ROS2-topics:

- **Prenumerationer:**
  - `/current_path`: Innehåller den banan som följs i nuläget.
  - `/tricycle_controller/odom`: Tillhandahåller robotens aktuella position och orientering.
  - `/forklift_control/enable`: Aktiverar eller inaktiverar controllern.
- **Publiceringar:**
  - `/cmd_vel`: Skickar styr- och hastighetskommandon till roboten.

8.5 Fördelar och Begränsningar

Pure Pursuit är en enkel och effektiv metod för banföljning i robotar. En av dess främsta styrkor är den adaptiva hastighetsregleringen, som säkerställer stabil rörelse även vid svängar. Dock kan algoritmen ha begränsningar vid höga hastigheter eller på banor med skarpa kurvor, vilket ökar risken för att roboten glider av banan.

8.6 Användning i Projektet

I projektet användes Pure Pursuit för att följa en planerad bana medan roboten undviker hinder. Den adaptiva hastighetsregleringen är särskilt viktig för att balansera precision och stabilitet vid rörelse genom komplexa miljöer. Parametrarna för *lookahead distance*, *maximal linjär hastighet*, och *maximal vinkelhastighet* har optimerats för vår specifika applikation.

9 SIMULERINGSMILJÖ

Miljön, dess hinder samt truckmodellen simuleras i Gazebo [1], en öppen källkodsplattform för robotsimulering. Gazebo erbjuder en fysikmotor och en simulerad miljö som kan läsas av med hjälp av en simulerad LIDAR.

9.1 Simuleringsverktyg

För att möjliggöra realistisk simulering och visualisering av systemet används programmen Gazebo och RViz2. Dessa verktyg integreras med ROS2 och används för att simulera miljön och sensorinformation samt för att visualisera truckens beteende i realtid.

Gazebo är ett fysikbaserat simuleringsverktyg som används för att modellera den fysiska miljön där trucken befinner sig. Simuleringsmiljön i Gazebo modellerar truckens kinematik och omgivning. Dessutom genererar Gazebo realistisk sensordata från LIDAR, som används för kartläggning. Gazebo är integrerat med ROS2 genom att publicera och prenumerera på ROS2-topics, vilket möjliggör simulering av robotens sensorer och styrning i realtid.

RViz2 används för att visualisera och övervaka truckens tillstånd och sensordata. RViz2 är integrerat med ROS2 via ROS2-topics som innehåller truckens position, sensordata, karta och planerad rutt.

9.2 Miljö och hinder

Simuleringsmiljön är utformad för att vara enkel och fokusera på grundläggande funktioner som hinderdetektion och ruttplanering. Miljön består av ett enkelt rum med fyra väggar och ett eller flera lådliknande hinder (se Figur 11). Antalet hinder, deras dimensioner och positioner kan konfigureras för att justera miljöns komplexitet.

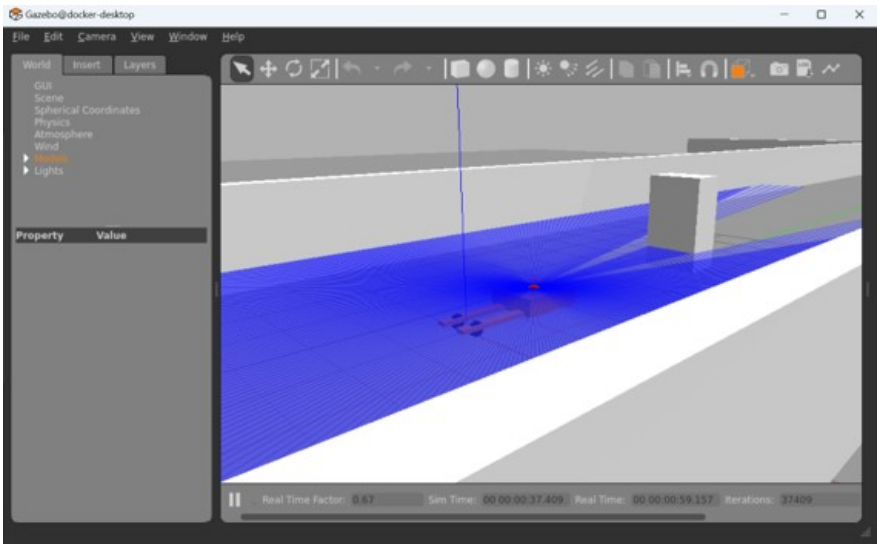
Miljön och hindren konfigureras och placeras i användargränsnittet som sedan skapar en SDF-fil (Simulation Description Format) [2], som används för att beskriva simuleringensmiljön som används i Gazebo. Denna fil används även för att skapa en karta som visualiseras i Rviz2.

9.3 Koordinatsystem och Transformationer

Systemet innehåller tre olika koordinatsystem: ett som är fast vid truckens kropp, ett kopplat till LIDAR-sensorn som sitter på trucken samt ett globalt referenssystem som representerar omgivningen. För att spåra truckens position och orientering hanteras transformationer mellan dessa koordinatsystem med *TF2-biblioteket* i ROS2.

9.4 Sensorsimulering

Simuleringen inkluderar en LIDAR-sensor som är monterad på truckens tak för att samla in data om omgivningen. Denna data används för att upptäcka hinder, skapa en karta över miljön och möjliggöra ruttplanering. Skanningsområdet kan konfigureras genom att välja vilka vinklar sensorn ska täcka. LIDAR-sensorns utdata representeras som avstånd och vinklar, och denna data bearbetas för att generera ett 2D-punktmoln.



Figur 11: Trucken i en enkel miljö.

9.5 Truckmodell

Trucken är modellerad som en enkel rektangulär box med två bakhjul och ett framhjul. Den kinematiska modellen är implementerad enligt beskrivningen i Avsnitt 8.1.

10 GRÄNSSNITT

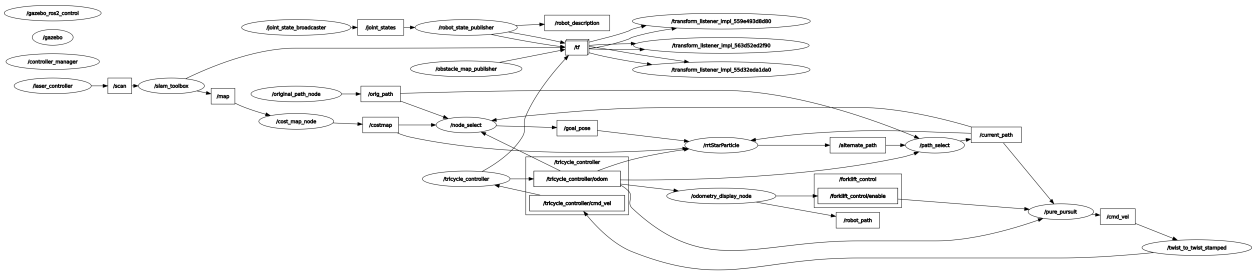
Detta avsnitt handlar om de gränssnitt som utvecklades under projektet.

10.1 ROS2

ROS2 är ett gränssnitt som möjliggör kommunikation mellan olika programvarukomponenter kallade noder. Genom sin ”Publish/Subscribe”-arkitektur kan utvecklingen enkelt göras modulär. ROS2 har även omfattande funktionalitet för att möjliggöra simuleringar, integration av sensorer, felsökning och rörelseplanering. I detta projekt användes ROS2 som gränssnitt för ruttplanering, sensormodul, hinderdetektion och simulering.

10.2 Noder och *topics*

I Tabell 2 visas alla noder listade med en beskrivning av vad de har för funktion samt hur informationsflödet ser ut mellan dem. Noderna och topics illustreras i Figur 12.



Figur 12: Noder och topics.

Nodnamn	**Topics**	**Beskrivning av noden**
`joint_state_broadcaster`	`/joint_states` (output)	Publicerar ledernas aktuella tillstånd.
`joint_state_publisher`	`/joint_states` (input) `/robot_description` (output) `/tf` (output)	Publicerar en beskrivning av trucken.
`obstacle_map_publisher`	`/tf` (output)	Publicerar hinder.
`controller_manager`	Diverse (input/output)	Hanterar controller-styrning för robotens ledmoduler.
`gazebo`	`/gazebo/...`	Simulerar miljön och robotens fysik.
`laser_controller`	`/scan` (output)	Publicerar LIDAR-skanningar för hinderdetektion.
`slam_toolbox`	`/scan` (input) `/map` (output) `/tf` (output)	Skapar en karta över omgivningen med SLAM-metod.
`cost_map_node`	`/map` (input) `/costmap` (output)	Genererar en kostnadskarta för planering av vägval.
`rrtStarParticle`	`/costmap` (input) `/goal_pose` (input) `/tricycle_controller...` `/odom` (input) `/path` (output)	Utför ruttplanering med en RRT* algoritm.
`original_path_node`	`/orig_path`(output)	Genererar en ursprunglig väg innan alternativa val görs.
`node_select`	`/orig_path` (input) `/alternate_path` (input) `/costmap` (input) `tricycle_controller...` `/odom` (input) `/goal_pose` (output)	Väljer vilken väg roboten ska följa.
`pure_pursuit`	`/current_path` (input) `/tricycle_controller...` `/odom` (input) `/cmd_vel` (output)	Implementerar Pure Pursuit-kontroll för att följa den valda vägen.
`tricycle_controller`	`/tricycle_controller...` `/cmd_vel` (input) `/tricycle_controller...` `/odom` (output) `/tf` (output)	Styr robotens hjul och hanterar odometri.

`odometry_display_node`	`/tricycle_controller...`
	`/odom` (input)
	`/forklift_control...`
	`/enable` (output)
	`/robot_path` (output)
		Illustrerar robotens utförda rutt.

`transform_listener_impl`	`/tf` (input/output)	Hanterar transformering av koordinater mellan olika ramar.
`twist_to_twist_stamped`	`/cmd_vel` (input) `/tricycle_controller...` `/cmd_vel` (output)	Omvandlar twist-meddelanden till twist-stamped-meddelanden.
`path_select`	`/orig_path` (input) `/alternate_path` (input) `/tricycle_controller...` `/odom` (input) `/current_path` (output)	Väljer om trucken ska följa orginalrutten eller den alternativa rutten.

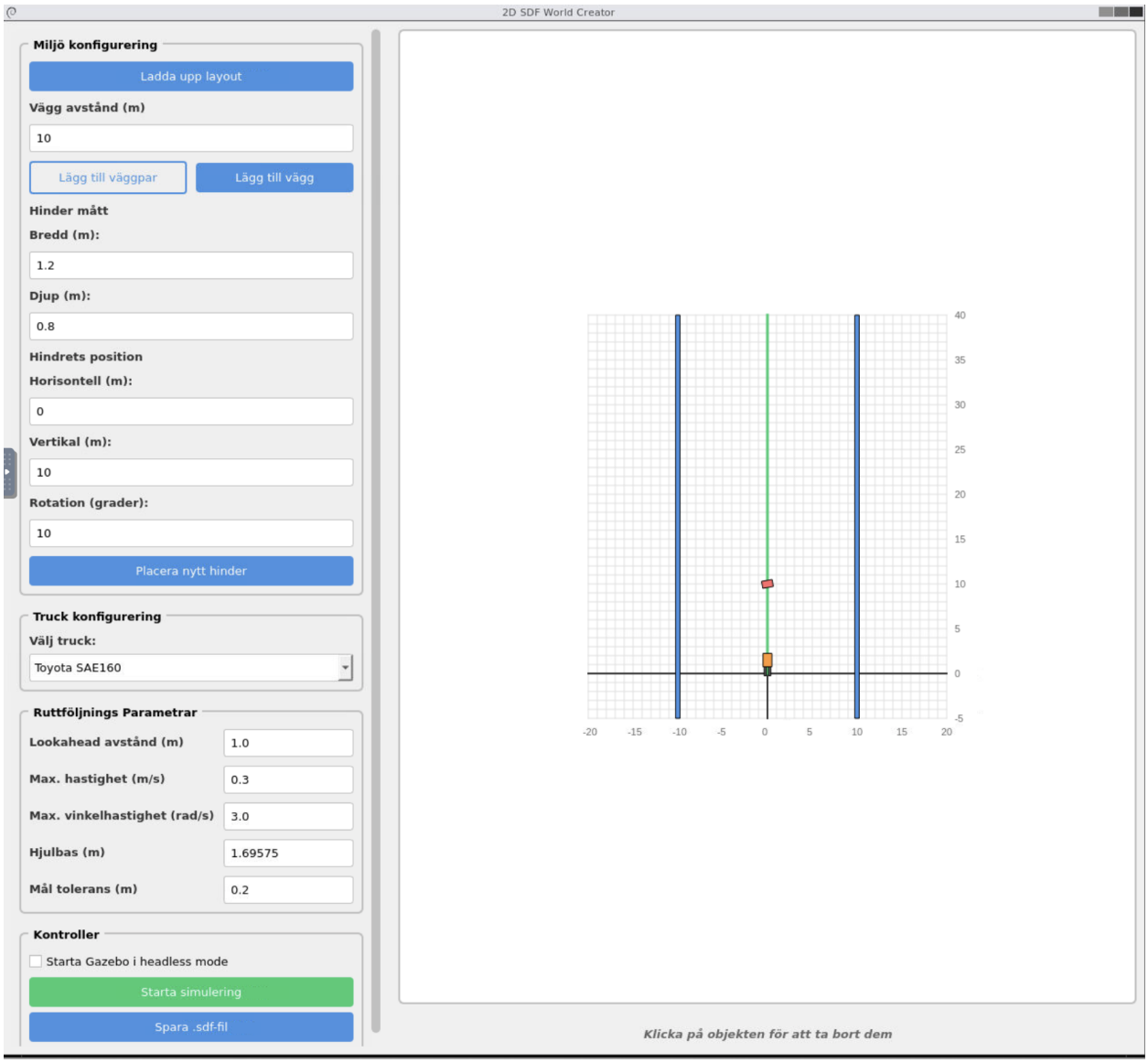
Tabell 2: Översikt över noder, deras *topics* och funktioner

10.3 Användargränssnitt

Ett Användargränssnitt används för att intuitivt konfigurera simuleringsmiljön och för att enkelt kunna observera simuleringen.

Användargränssnittet består av två paneler. Det första används för att definiera simuleringsvärlden, välja truckkonfiguration samt konfigurera parametrar för ruttföljaren. Det andra fönstret visar trucken under simuleringen tillsammans med dess omgivning, rutter och hinder. Dessutom visualiseras truckens beslutsgång och dess aktuella position i beslutsordningen.

I Figur 13 visas panelen som används för att konfigurera simuleringsmiljön och truckens inställningar. Den vänstra kolumnen används för att placera ut hinder och välja olika konfigurationer. Kartan till höger visar omgivningen och dess hinder. Där kan man klicka på objekten med muspekaren för att ta bort dem från världen. När simuleringsmiljön är färdigkonfigurerad kan man antingen starta simuleringen eller spara en .sdf-fil som definierar omgivningen. Det finns också en möjlighet att välja om användargränssnittet för simuleringsmotorn Gazebo ska startas.

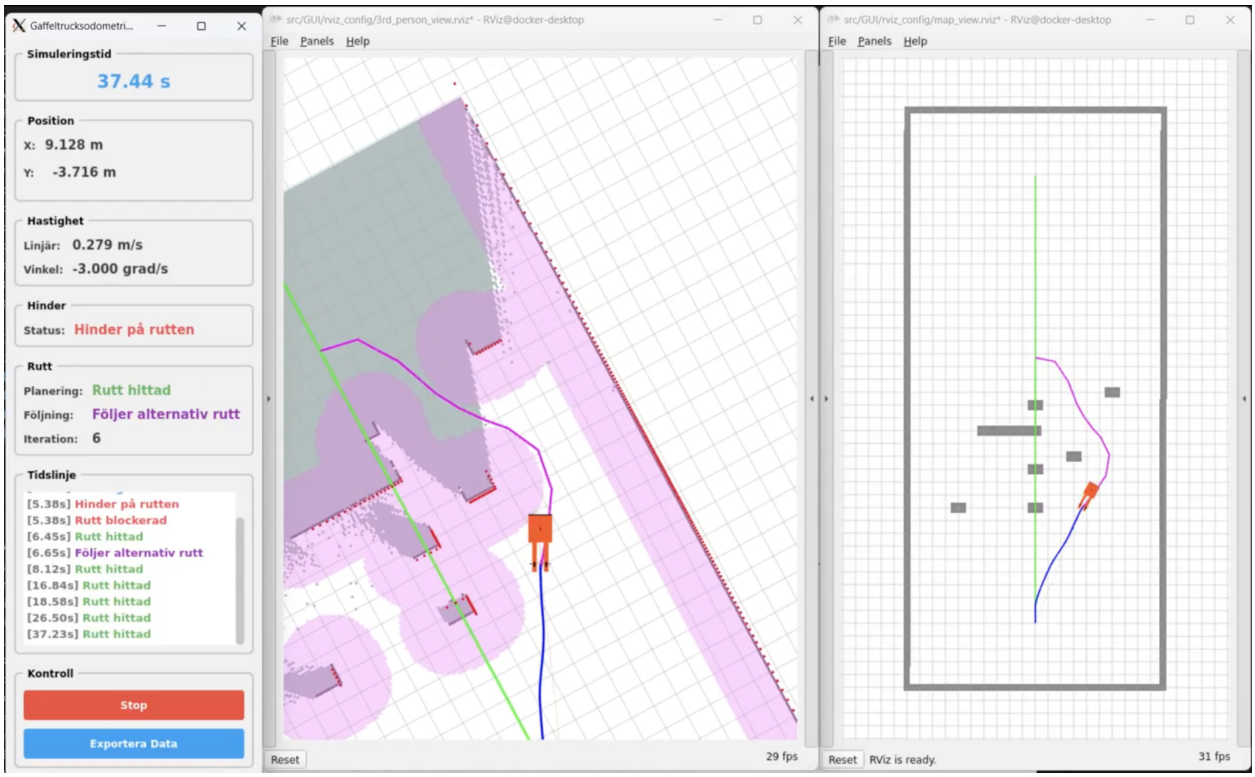


Figur 13: Användargränssnitt för att definiera och konfigurera simuleringsmiljö

När simuleringen startas visas en panel med tre fönster, som kan ses i Figur 14. Fönstret längst till vänster visar truckens position och hastigheter. Det visar även om trucken har upptäckt ett hinder på sin ursprungliga rutt, om den har beräknat en ny rutt och vilken rutt den för närvarande följer, samt vilken iteration av rutten som körs. Dessutom innehåller det en tidslinje som illustrerar händelseförloppet och när olika händelser inträffade. Här finns också knappar för att starta och stoppa trucken samt för att spara odometridata, händelseförlopp och .sdf-filen som definierar världen, ifall man vill utvärdera eller jämföra ruttföljningen mer numeriskt.

Fönstret i mitten är ett RViz2 fönster som visualiserar ROS2-topics som kan vara bra för utvärdering av truckens autonoma funktionalitet. De olika topics som visualiseras får att se i Tabell 3.

Fönstret längst till höger är ett RViz2 fönster som visar en karta över den faktiska omgivningen tillsammans med truckens olika rutter. De topics som visas och dess färger går också att se i Tabell 3. Notera att trucken inte kan se denna karta.



Figur 14: Användargränssnitt för att observera och utvärdera simulering. Denna bild är i mitten av simulering.

Topic	Färg	Mitten Fönster	Höger Fönster
`/robot_description`	Orange (Truck konfigurering)	X	
`/orig_path`	Grön linje	X	X
`/map`	Vitt/Grått område	X	
`/scan`	Röda prickar	X	X
`/alternate_path`	Lila linje	X	X
`/costmap`	Lila område	X	
`/robot_path`	Blå linje	X	X
`/obstacle_map`	Grått område		X

Tabell 3: Topics som visualiseras i RViz2-fönstrerna.

11 REFERENSER

REFERENSER

[1] Open Robotics. *Gazebo Simulation Platform*. https://gazebosim.org/home. 2024.

[2] Open Source Robotics Foundation. *Simulation Description Format (SDF)*. http://sdformat.org/. 2024.