Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

Implementation of a motion planning algorithm for the USAD vehicle within the ROS navigation stack

Relatore: *Prof. Domenico G. Sorrenti*

Co-relatore: *Dr. Augusto Luis Ballardini*

Relazione della prova finale di:

Fabio Nava

Matricola 793136

Anno Accademico 2016-2017

Indice

Introduzione **1**

1 Tool utilizzati **3**
1.1 ROS (Robotic Operating System) 3
1.2 RViz 6
1.3 Gazebo 7
1.4 QtCreator 8
1.5 Git 8

2 Veicolo USAD **9**

3 Navigation Stack **11**
3.1 Move Base 12
3.2 Costmap 14
3.3 Global Planner 15
3.4 Local Planner 16
3.5 Recovery Behaviours 17

4 Struttura progetto **18**
4.1 Sampling-Based Motion Planning 18
4.2 Panoramica generale 21
4.3 Strutture utilizzate 22
4.4 Libreria IraLocalPlanner 25
4.5 Libreria MotionController 26

5 Test **30**
5.1 ira_local_planner/IraLocalPlanner e global_planner/GlobalPlanner 31
5.2 ira_local_planner/IraLocalPlanner e percorso predefinito 32
5.3 ira_local_planner/IraLocalPlanner e ira_global_planner/IraGlobalPlanner 33
5.4 Prove sul cart 34

6 Conclusioni e sviluppi futuri **36**

A Codice **38**

A.1 ackermanForwardSimulation 38

A.2 simulateTreeEdge 40

A.3 evaluateTrajectories 41

Introduzione

> *"Un giorno le macchine riusciranno a risolvere tutti i problemi, ma mai nessuna di esse potrà porne uno."* - A. Einstein

Lo scopo di questo lavoro è l'implementazione dell'algoritmo di pianificazione e controllo del moto per un veicolo a cinematica Ackerman. L'algoritmo è stato implementato all'interno dello stack di navigazione del framework ROS come un nuovo modulo di pianificazione locale, partendo da un prototipo sviluppato in MATLAB secondo un approccio noto in letteratura (U. Schwesinger, M. Rufli, P. Furgale, R. Siegwart - "A Sampling-Based Partial Motion Planning Framework for System-Compliant Navigation along a Reference Path"). La pianificazione del moto avviene basandosi su un percorso calcolato in precedenza da un altro modulo dello stack di navigazione, che si occupa della pianificazione globale.
Per conseguire lo scopo di questo lavoro si è partiti con l'analisi del funzionamento di ROS e in particolare dello stack di navigazione, esaminando tutte le sue componenti interne. È stata data molta importanza al nodo chiamato move_base, il quale comprende, tra le altre, il plugin sviluppato durante il percorso di stage. Successivamente si è passati all'analisi degli algoritmi per la pianificazione e il controllo del moto, e alla loro implementazione in MATLAB. Quindi è stato possibile iniziare lo sviluppo del mio plugin partendo dalla rielaborazione del prototipo in ROS e dall'integrazione con lo stack di navigazione. Durante lo sviluppo del progetto sono stati effettuati molti test, utilizzando dei programmi per la simulazione, quali RViz e Gazebo, per simulare la pianificazione e il controllo del veicolo. Infine sono state fatte delle demo sul veicolo USAD per provare direttamente su strada e verificare i risultati ottenuti con le simulazioni.
La relazione di stage sarà suddivisa nei seguenti capitoli:

- **Capitolo 1** - vengono presentati tutti gli strumenti utilizzati durante lo sviluppo del mio progetto. Tra questi il framework ROS, con i meccanismi base del suo funzionamento, il tool di visualizzazione RViz e il simulatore Gazebo utilizzati per effettuare le simulazioni, e l'integrazione con l'IDE QTCreator.
- **Capitolo 2** - viene presentata una breve descrizione del veicolo USAD, citando le sue particolarità e i sensori a bordo.

- **Capitolo 3** - viene descritto il pacchetto relativo alla navigazione presente in ROS, Inizialmente verrà fatta una panoramica generale su tutte le sue componenti, spiegando la comunicazione tra di esse, successivamente verranno analizzate più nel dettaglio quelle principali.

- **Capitolo 4** - viene descritto il progetto svolto nel periodo di stage. Innanzitutto viene analizzato nel dettaglio il framework di partenza, con una spiegazione completa e la relativa implementazione in MATLAB. Si passa poi alla descrizione del progetto sviluppato, *IraLocalPlanner*, di cui viene fornita una descrizione esaustiva. Ogni funzione implementata verrà spiegata dettagliatamente, con particolare attenzione a quelle di pianificazione e controllo del moto. Vengono presentate le due librerie del progetto, *IraLocalPlanner* e *MotionController*.

- **Capitolo 5** - vengono presentati i test di simulazione e i risultati ottenuti eseguendo più simulazioni con parametri differenti. Vengono elencati anche i parametri ideali per ottenere un risultato migliore. Infine viene descritta anche una fase di sperimentazione sul veicolo

- **Capitolo 6** - vengono tratte le conclusioni del lavoro svolto durante lo stage e proposti dei possibili sviluppi futuri.

Infine nell'appendice A sono inclusi i codici degli algoritmi principali per la pianificazione del moto.

1 Tool utilizzati

Per lo sviluppo del mio progetto sono stati utilizzati diversi tool, inizialmente si è preso confidenza con ROS, il framework per sviluppare applicazioni robotiche, poi si è utilizzato RViz e Gazebo per le simulazioni. Per lo sviluppo del progetto sono stati utilizzati l'IDE QTCreator e Git per tracciare tutte le modifiche di volta in volta e per facilitare lo sviluppo congiunto con i colleghi.

1.1 ROS (Robotic Operating System)

ROS, acronimo di Robot Operating System, è un framework open-source che include molti strumenti utili per lo sviluppo di applicazioni robotiche.

Figura 1.1: Rispettivamente a sinistra il logo di ROS e a destra il logo della distribuzione utilizzata, Kinetic Kame

È stato sviluppato da alcuni ricercatori, tra cui Willow Garage, ed è nato con lo scopo di semplificare la ricerca sulla robotica di servizio.
Questo framework può essere installato su un computer avente come sistema operativo una "distro" di Linux, in particolare noi abbiamo utilizzato la versione Kinetic[1] installata su Ubuntu 16.04.

[1]Disponibile al seguente link: `http://wiki.ros.org/kinetic/Installation`

Ogni sistema ROS è basato su un network di processi eseguiti in parallelo che comunicano tra loro utilizzando un'architettura Peer-to-Peer. Essi sono chiamati Nodi e possono avere la funzione di server o client. Prima di utilizzare un qualsiasi sistema ROS è necessario eseguire un server principale con il comando "*roscore*". Questo server, chiamato anche ROS Master, permette ai processi di comunicare tra loro.

La comunicazione tra i nodi avviene scambiando "*messaggi*", predefiniti in ROS oppure definiti dallo sviluppatore. Questi vengono scambiati attraverso dei canali chiamati "*Topic*". Esistono nodi detti Publisher, i quali creano il canale e possono spedire messaggi, e nodi detti Subscriber, i quali devono sottoscriversi al canale di interesse per riceverne i messaggi pubblicati. Da un punto di vista prettamente tecnico, quando viene ricevuto un messaggio, il nodo sottoscritto richiama una Callback per elaborare le informazioni ricevute.
Vengono ora descritti nello specifico alcuni elementi base di ROS:

- **Nodi**: processi che svolgono un'attività specifica, ogni nodo viene registrato nella rete dei processi con un ID univoco e può comunicare con altri nodi scambiando messaggi attraverso i topic. In un sistema di controllo di un robot un nodo rappresenta una componente specifica quale un sensore, un motore o un'attività di localizzazione.

- **Messaggi** : pacchetti di dati strutturati per essere usati dai nodi durante la comunicazione. Vengono pubblicati e ricevuti attraverso i topic. Un Publisher e un Subscriber devono inviare e ricevere lo stesso tipo di messaggio per poter comunicare.

- **Topics** : canale di comunicazione tra due nodi con semantica Publish and Subscrive. Un nodo può inviare un messaggio su un topic connettendosi come Publisher, oppure può ricevere un messaggio connettendosi come Subscriber. Per un singolo topic possono esserci più Publisher e Subscriber, allo stesso tempo un nodo può pubblicare e/o sottoscrivere a più topic. Sono un sistema di trasporto asincrono e monodirezionale, si può ottenere un sistema di trasporto sincrono utilizzando due topic o un servizio.

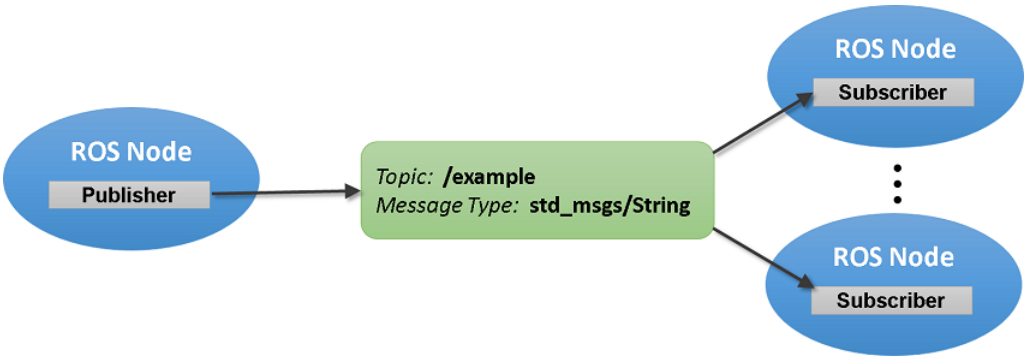

Figura 1.2: Funzionamento della comunicazione tra Publisher e Subscriber attraverso un topic

- **Servizi** : canale di comunicazione tra due nodi con semantica Request and Response. Un nodo può richiedere un servizio mandando una query e riceverà un messaggio di risposta.

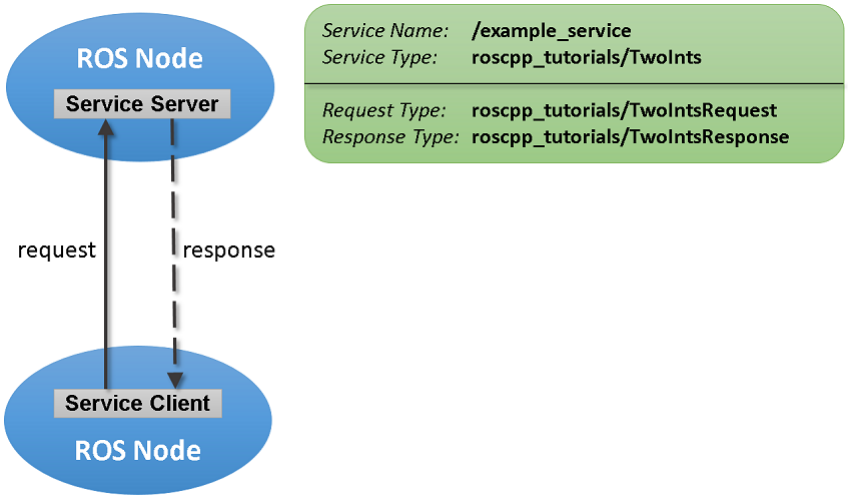

Figura 1.3: Funzionamento della comunicazione tra un Service Server e un Client

- **Plugin** : sono classi caricabili dinamicamente da una libreria a runtime, non è necessario linkare esplicitamente la libreria contenente una classe basta soltanto richiamare il nome del plugin corrispondente. Tutte le componenti del move base sono sotto forma di plugin.

1.2 RViz

RVIZ è uno strumento che consente la visualizzazione delle componenti geometriche del sistema. Possiamo osservare graficamente molte informazioni tra cui percorsi, posizioni e traiettorie.

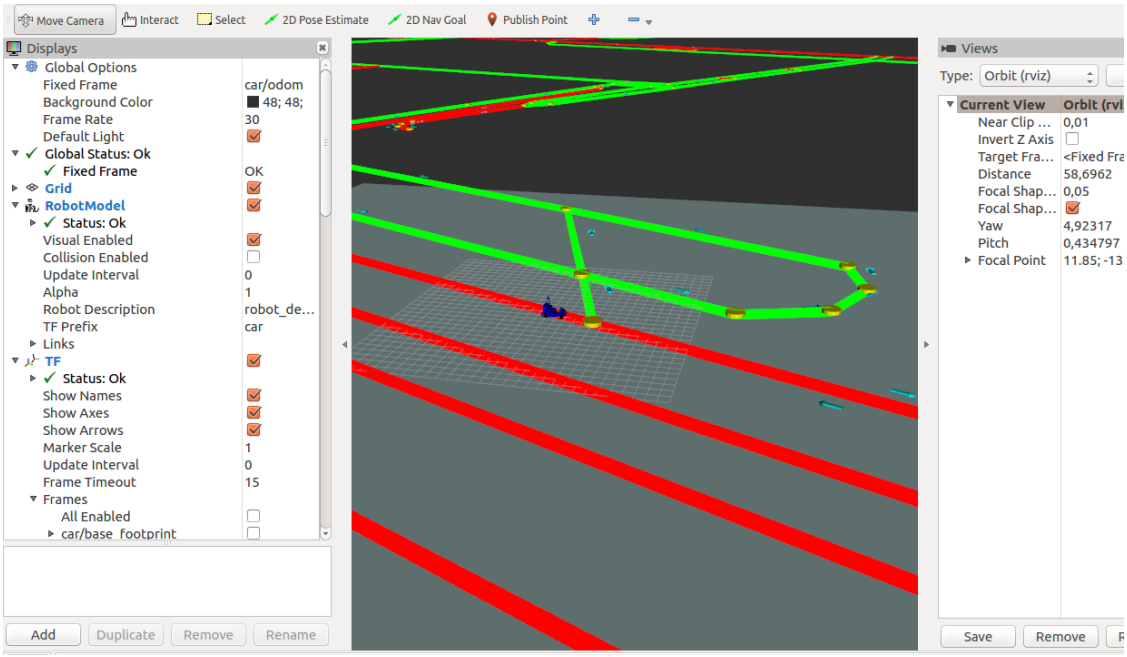

Figura 1.4: Interfaccia utente del programma RViz

Come si può notare dall'immagine 1.4, tralasciando la sezione centrale in cui vengono visualizzate le informazioni, RViz è così suddiviso nei seguenti pannelli :

- **Displays**: vengono elencati gli oggetti visualizzati nella sezione centrale, possono esserne aggiunti altri o rimossi dall'utente.
- **Views**: ci permette di scegliere in quale angolazione visualizzare il mondo tridimensionale.
- **Tools**: fornisce diverse funzionalità con le quali interagire nel mondo tridimensionale, tra cui 2d Nav Goal che ci permette di dare una posizione di arrivo desiderata per il nostro veicolo.

Questo strumento è stato utilizzato ogniqualvolta è stata eseguita una simulazione su pc in laboratorio, ma anche nella fase di sperimentazione direttamente sul veicolo. Insieme ad RViz è stato utilizzato anche Gazebo per simulare il veicolo.

1.3 Gazebo

Gazebo è uno strumento di simulazione che permette di creare ed "animare" oggetti tridimensionali e aggiungerli nel mondo virtuale creando l'ambiente desiderato. Come si può notare dall'immagine 1.5, in questo lavoro è stato preso un modello 3D di un Golf Cart molto simile al nostro, sviluppato dall'università UPC di Barcellona[2], in modo da rendere più veritiere possibili le nostre simulazioni.

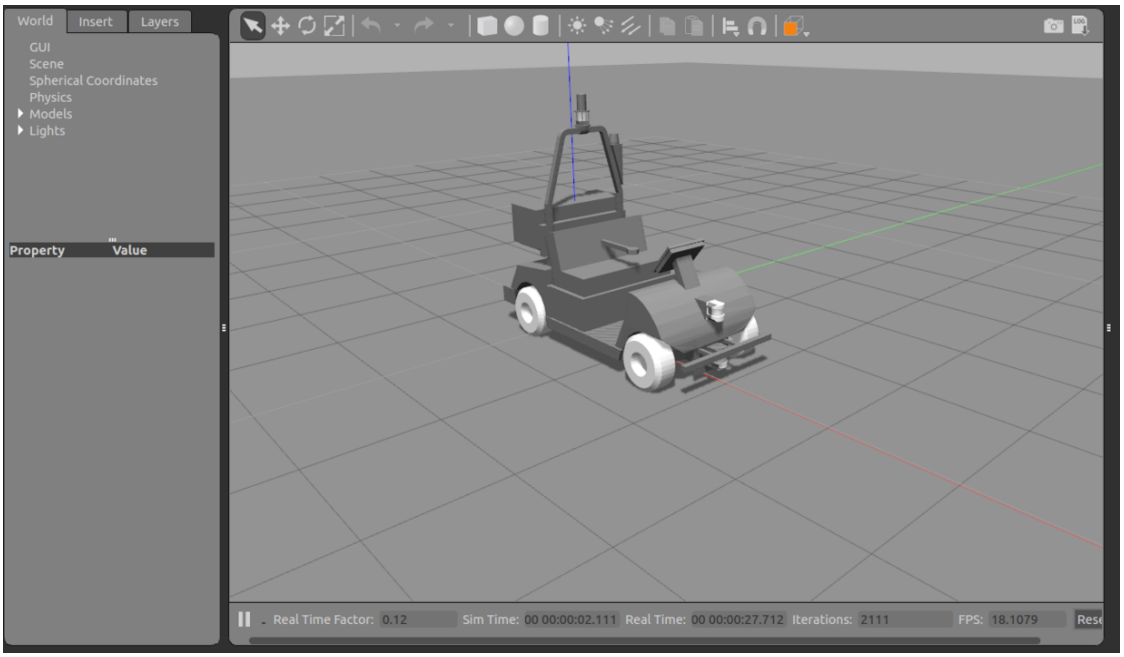

Figura 1.5: Interfaccia utente del programma Gazebo ed il modello del cart utilizzato

[2]Disponibile al seguente link:`https://devel.iri.upc.edu/pub/labrobotica/ros/iri-ros-pkg_hydro/metapackages/car_robot`

1.4 QtCreator

Gli oggetti di ROS possono essere scritti in Pyton o in C++, e nel mio progetto è stato scelto C++. Per facilitare le operazioni di implementazione del software è stato necessario l'utilizzo di un IDE (Integrated Development Environment), ed è stato scelto QtCreator. Esso è multipiattaforma e open-source, fornisce molti tool utili per la programmazione. Per importare un progetto ROS all'interno di QTCreator basta aprirlo dal file CMakeList.txt[3], file che viene automaticamente generato alla creazione di un progetto ROS.

1.5 Git

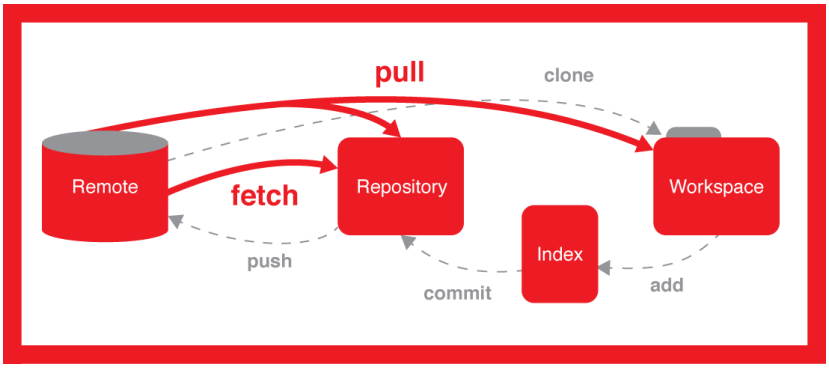

Figura 1.6: funzionamento di Git

Git è un sistema di controllo di versione distribuito creato da Linus Torvalds per supportare lo sviluppo del kernel di Linux. Consiste in un sistema che registra nel tempo i cambiamenti ad un file o ad una serie di file, per poter richiamare una specifica versione in qualsiasi momento. Ogni modifica del codice viene registrata con una descrizione in modo tale da essere reperibile in un secondo momento.

[3]Come creare i nodi al seguente link:`http://wiki.ros.org/it/ROS/Tutorials`

2 Veicolo USAD

Figura 2.1: Veicolo USAD con rappresentati i sensori. In azzurro le due camere anteriori, in rosso i tre lidar montati sempre anteriormente, in giallo due encoder montati sulle ruote posteriori, in nero il motore montato sullo sterzo.

Il veicolo USAD (Urban Shuttles Autonomously Driven) è un progetto del laboratorio di ricerca IRALAB dell'Università degli Studi di Milano Bicocca, il quale si occupa di sviluppare applicativi per la guida autonoma in contesto urbano. Questo veicolo è un golf cart elettrico sensorizzato.
Come possiamo vedere nell'immagine 2.1, i sensori montati a bordo sono:

- due telecamere presenti all'interno dell'abitacolo e rivolte in avanti che permettono la visione stereoscopica per individuare le linee stradali, il piano stradale e la segnaletica verticale.

- tre lidar montati nella parte anteriore, due ai lati e uno al centro, permettono di localizzare gli ostacoli e quindi calcolare la distanza tra il veicolo e tutti gli oggetti che lo circondano.
- quattro IMU posizionate in diversi punti del veicolo utilizzate per ottenere i dati riguardanti l'orientamento e il cambio di accelerazione del veicolo.
- un motore montato sullo sterzo contenente un encoder per permettere una rotazione autonoma precisa.
- due Encoder montati sulle ruote posteriori, utilizzati per tracciare il movimento delle ruote e calcolare lo spostamento del veicolo.
- due GPS per avere una localizzazione precisa del veicolo.

3 Navigation Stack

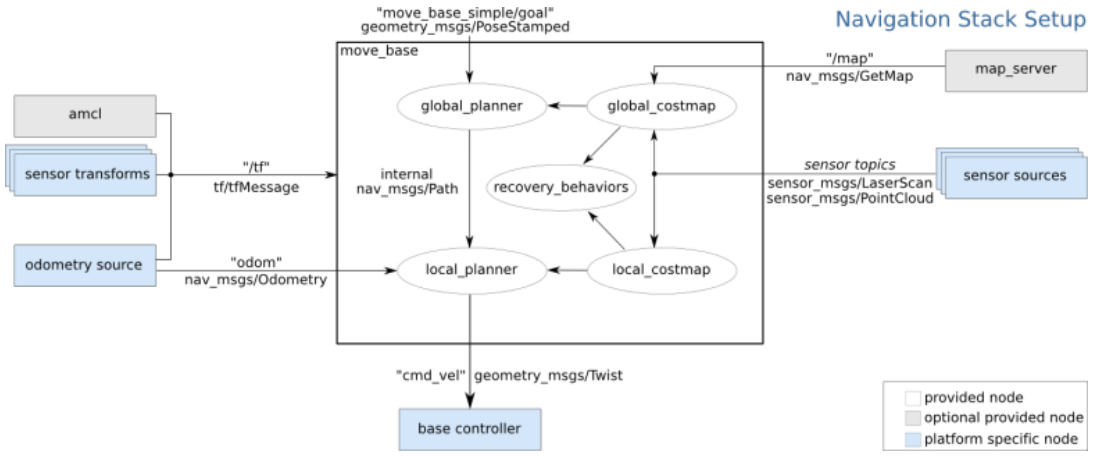

Figura 3.1: Schema sintetico del funzionamento del Navigation Stack

Il Navigation Stack è un package di ROS, che acquisisce dati da odometria, flussi di sensori e pose di obiettivi, quali setpoint o waypoint da raggiungere, per generare comandi di velocità da inviare al robot. Gestendo tutte le fasi di navigazione permette di far muovere un robot in una posizione desiderata.
Nella figura 3.1 sono rappresentate le interazioni tra i componenti di questo pacchetto. Le componenti bianche sono considerate fondamentali, mentre quelle grigie sono opzionali. Queste due categorie sono già presenti in ROS, mentre quelle blu devono essere create specificatamente per ogni applicazione robotica.
"Amcl" e "map server" sono componenti opzionali. La prima, Adaptive Monte Carlo Localization, è un sistema di localizzazione probabilistico per robot in un ambiente bidimensionale, la seconda invece fornisce a ROS i dati necessari per caricare le mappe 2d dell'ambiente a runtime. Al posto di "Amcl" è stato utilizzato il lavoro sviluppato da un ex stagista, Pietro Colombo, chiamato *"Localizzazione Robusta per il veicolo USAD"*, il quale utilizza tutti i dati presi da GPS, IMU e Odometria per una localizzazione più precisa. "Map server" invece non è stato utilizzato perché nella navigazione reale su strada non sempre è possibile navigare in una mappa bidimensionale a griglia nota, e lo scopo finale del lavoro è riuscire a navigare in un

ambiente sconosciuto utilizzando comuni mappe stradali.
"Sensor sources", "sensor transforms", "odometry sources" e "base controller", sono componenti specifiche per ogni applicazione robotica. "Sensor sources" fornisce informazioni riguarado gli ostacoli nella mappa, sia statici che dinamici, e li manda alla global costmap e alla local costmap, le quali verranno spiegate successivamente nella sezione apposita. "Sensor Transform" mantiene correttamente le matrici di trasformazione tra i frame di tutti i sensori. "Odometry sources" racchiude tutte le informazioni riguardanti l'odometria per localizzare il robot. "Base controller" si occupa di convertire i messaggi inviati dal move base attraverso il topic cmd_vel in comandi da inviare al motore.

Le restanti componenti "Global Planner", "Local Planner", "Global Costmap", "Local Costmap" e "Recovery Behaviours" fanno tutte parte del "Move Base". Nelle sezioni successive andrò ad introdurre nel dettaglio tutte queste componenti presenti nel "Move Base".

3.1 Move Base

Il "Move Base" è il nodo principale di tutto il pacchetto, fornisce l'implementazione necessaria per controllare l'insieme di azioni da eseguire per raggiungere un traguardo. Al suo interno troviamo diverse componenti:

- **"Costmap (Local e Global)"** che sono delle "mappe" con all'interno gli ostacoli rilevati
- **"Global Planner"** che si occupa di tracciare un percorso dalla posizione attuale a un qualsiasi punto desiderato sulla mappa
- **"Local Planner"** che attua il percorso calcolato sopra
- **"Recovery Behaviours"** che entra in azione qualora il cart incontri un problema e cerca di riportarlo sulla strada giusta

Nei prossimi paragrafi verranno analizzati i componenti del move_base, per capirne meglio il funzionamento abbiamo realizzato un grosso diagramma, visibile nell'immagine 3.2, in cui vengono schematizzate tutte le interazioni tra il move_base e i suoi componenti. Siccome questo schema è troppo grosso è stato diviso in più parti e descritto negli appositi paragrafi.

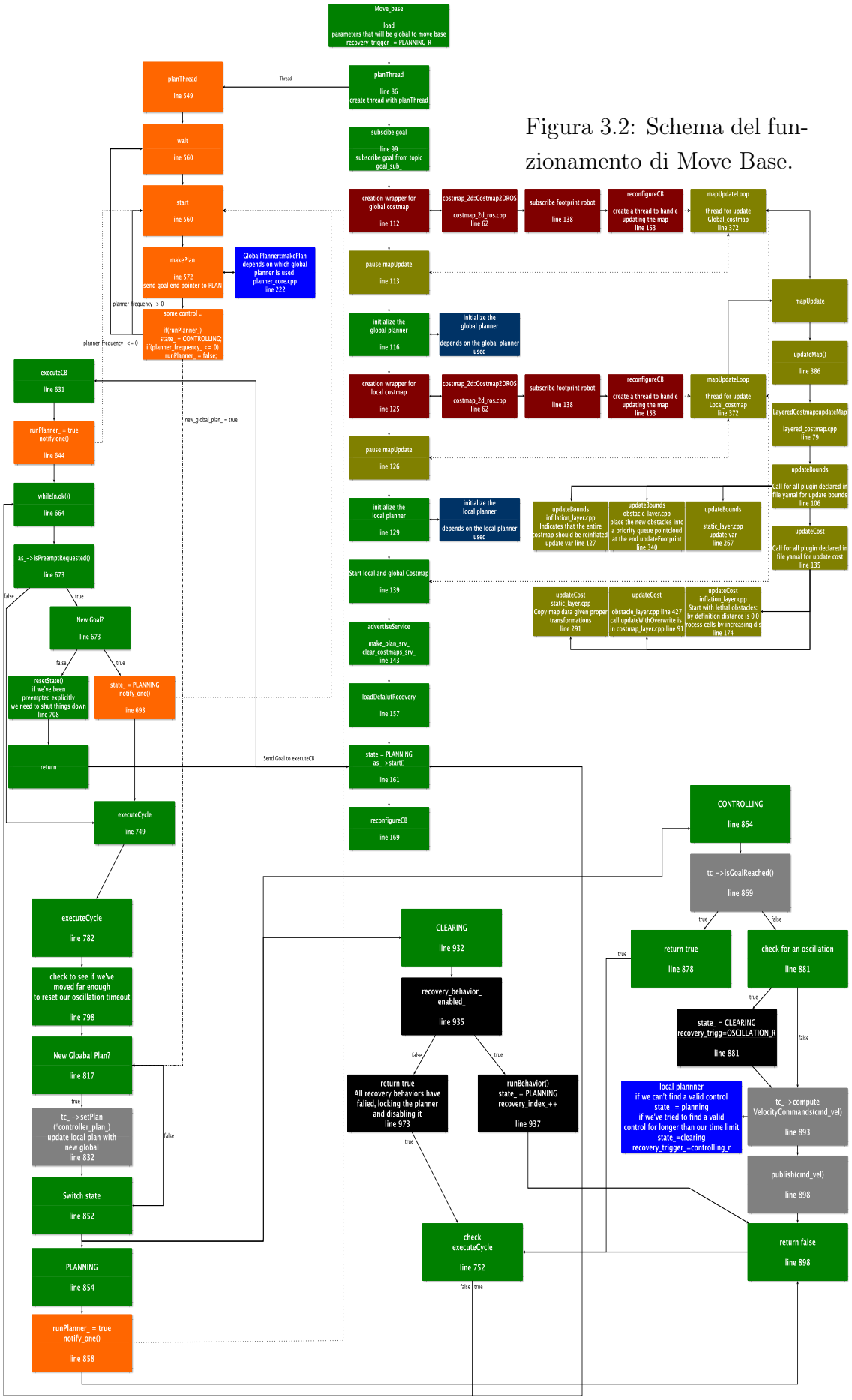

Figura 3.2: Schema del funzionamento di Move Base.

3.2 Costmap

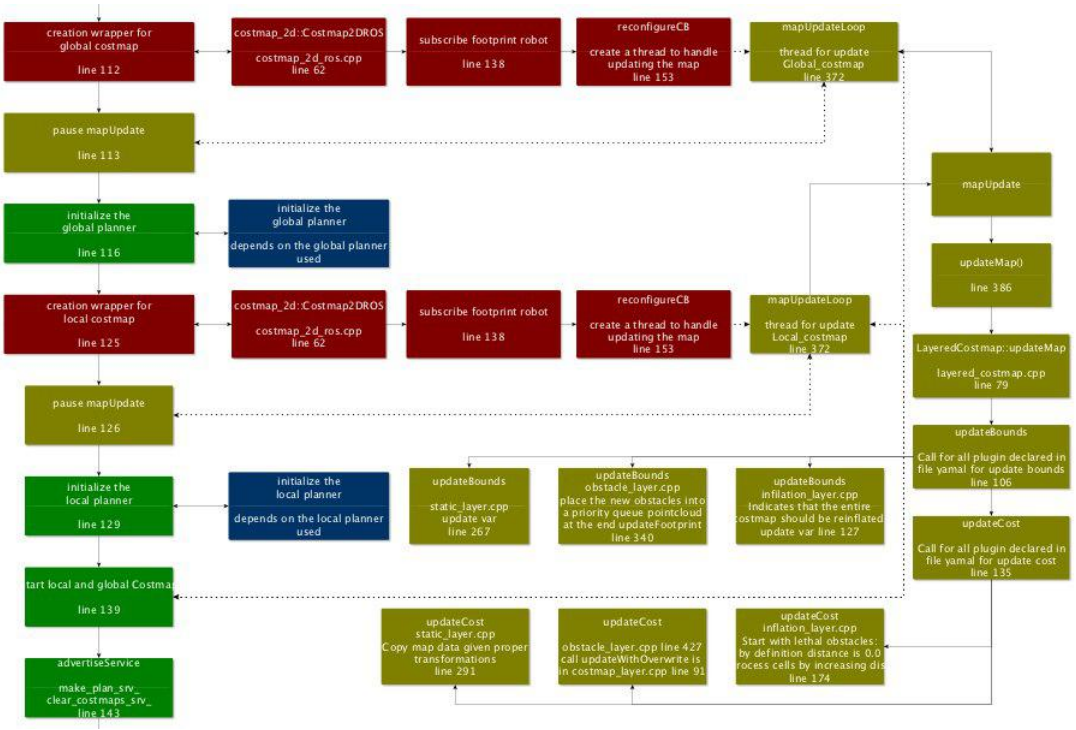

Figura 3.3: In rosso le interazioni di *move_base* con *costmap_2d* e in verde le chiamate al thread *mapUpdateLoop*.

Supponendo di suddividere lo spazio in celle, questo pacchetto fornisce alcuni plugin per convertire le celle occupate in primitive geometriche. Queste primitive, che possono essere punti, linee o poligoni, costituiscono gli ostacoli nella mappa. Sono utilizzate sia dal global costmap, in modo statico, sia dal local costmap in continuo aggiornamento. Il package standard utilizzato nel Navigation Stack è *costmap_2d* nel quale sono presenti diversi plugin.
In particolare vengono eseguiti dei cicli di aggiornamento della mappa ad una frequenza settabile dall'utente. Questi aggiornamenti avvengono ogniqualvolta venga chiamato il thread citato nella figura 3.3.

3.3 Global Planner

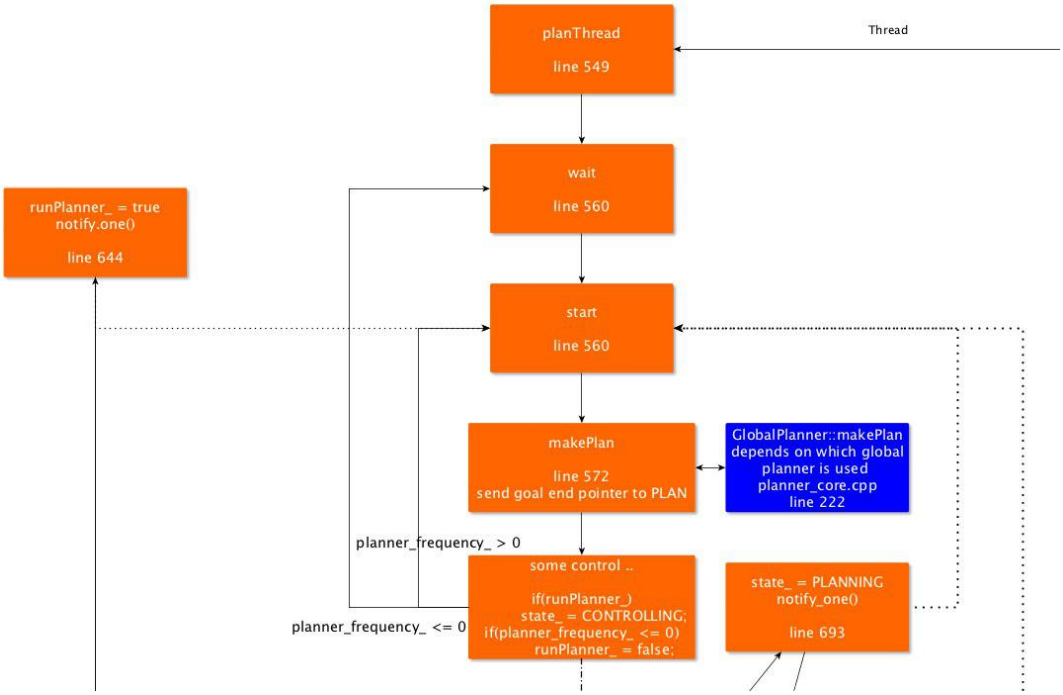

Figura 3.4: In arancione le interazioni di *move_ base* con un qualsiasi *global_ planner* e in blu il metodo principale.

Questo pacchetto fornisce un'implementazione di un pianificatore globale veloce per la navigazione. Ci sono diversi pacchetti, prendendo in input la posizione attuale e la posizione di arrivo desiderata ognuno utilizza determinati algoritmi di calcolo per trovare il percorso migliore. I package standard sono NavFn e GlobalPlanner.

3.4 Local Planner

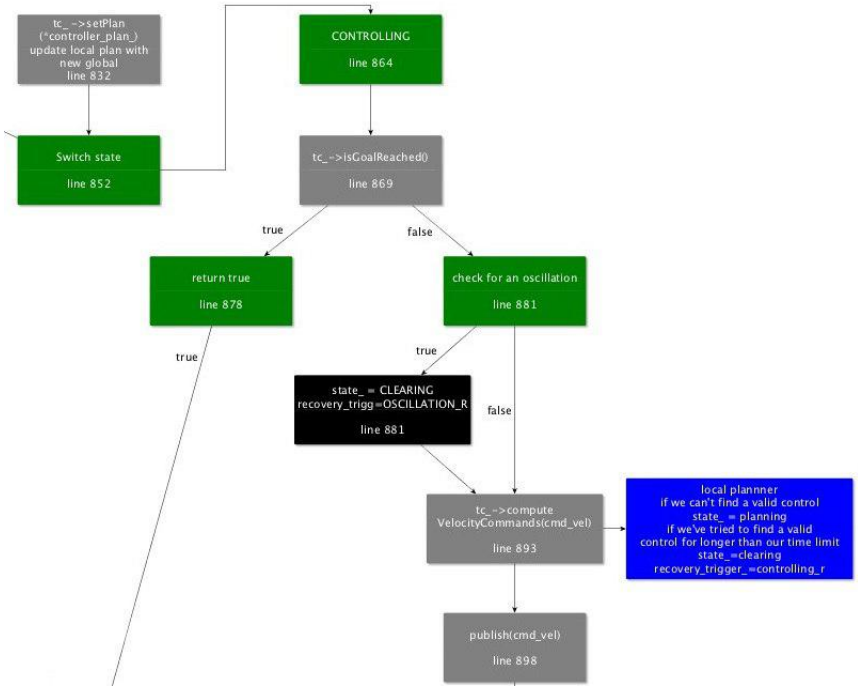

Figura 3.5: In grigio le interazioni di *move_base* con un qualsiasi *local_planner* e in blu l'output del metodo principale.

Questo pacchetto fornisce un'implementazione di un pianificatore locale per la navigazione di un robot su un piano. Dato un piano da seguire, preso dal global planner, e una costmap, mappa contenente gli ostacoli, il controller produce comandi di velocità da mandare al base_controller. Può essere specializzato per diversi tipi di robot mobili, sia con ruote holonomiche che non-holonomiche.

3.5 Recovery Behaviours

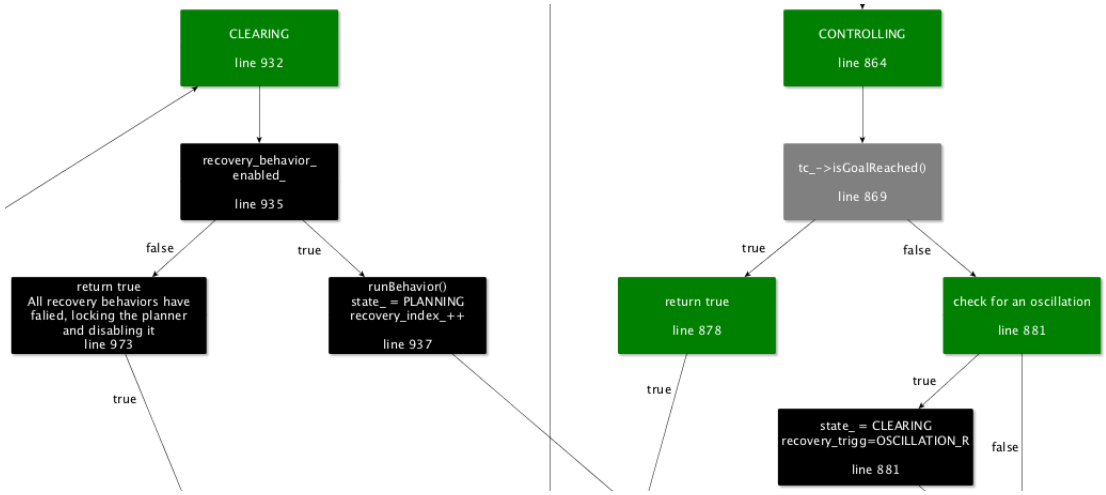

Figura 3.6: In nero le iterazioni di *move_base* col pacchetto *recovery_behaviours*

Questo pacchetto fornisce una procedura di ripristino nel tentativo di liberare spazio nelle "Costmap". Viene richiamato quando il robot è incapace di seguire il percorso computato dal "Move Base", una delle cause principali è la presenza di ostacoli, sia statici che dinamici.

4 Struttura progetto

In questo capitolo verrà presentato il package sviluppato per la pianificazione locale su un veicolo Ackerman.
Per sviluppare il mio progetto mi sono basato sul programma "Ackerman Forward Simulation" sviluppato da Axel Furlan, incentrato sulla creazione di un algoritmo di pianificazione e controllo del moto per una macchina a guida autonoma. L'algoritmo è stato implementato partendo da un prototipo sviluppato in MATLAB e una versione approssimativa in ROS, secondo un approccio in letteratura (U. Schwesinger, M. Rufli, P. Furgale, R. Siegwart - *"A Sampling-Based Partial Motion Planning Framework for System-Compliant Navigation along a Reference Path"*).

4.1 Sampling-Based Motion Planning

Il termine Motion Planning (pianificazione del moto) significa scomporre il moto desiderato in una serie di movimenti discreti che soddisfino i vincoli di movimento, sulla velocità, accelerazione o sterzo. Un algoritmo di questo tipo deve considerare i task che il robot deve svolgere per ottenere un movimento continuo dal punto di partenza al punto di arrivo, tenendo presente la dinamica del robot e le eventuali regole dell'ambiente in cui lavora, ostacoli statici o aree vietate.

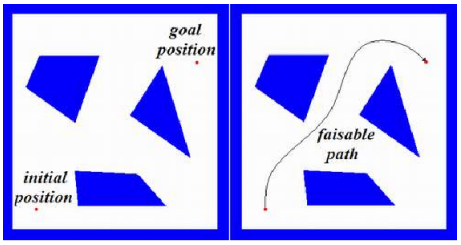

Figura 4.1: Problema della pianificazione del moto

Questo algoritmo si basa su un metodo di campionamento nel tempo, suddividendo il tempo di pianificazione in tanti piccoli Δt e simulando l'andamento un passo alla volta. Questo metodo è dato dalle specifiche di un modello di veicolo a cinematica Ackerman e di una legge di controllo, scelti dall'utente. Le specifiche impostabili riguardano i limiti e le caratteristiche del veicolo, quali massimo e minimo angolo di sterzo, massima velocità di sterzo, massima accelerazione e decelerazione e la lunghezza inter-asse. I controlli vertono su come suddividere il tempo, su quanti livelli sviluppare la simulazione, su quanto peso dare agli errori di allineamento e di distanza e su quali velocità e offset utilizzare per allineare le traiettorie. Per veicolo a cinematica Ackerman si intende un qualsiasi veicolo con quattro ruote, di cui le due anteriori sterzanti. Una volta terminata la simulazione, il modello del veicolo è regolato su un insieme di possibili stati terminali scelti ed allineati con i valori di riferimento, per la velocità e per l'offset laterale rispetto al percorso, generando traiettorie che considerino il veicolo e i suoi vincoli. Successivamente queste traiettorie vengono analizzate e quella migliore viene attuata.
L'algoritmo di pianificazione è composto da:

- **una routine di ricerca su grafo**
- **un modello di sistema**
- **una legge di controllo**

Viene utilizzata una struttura ad albero costruita durante l'esecuzione. Come possiamo notare nella figura 4.2 nodo n, appartenente all'insieme dei nodi **N**, rappresenta lo stato della macchina nel tempo, $(x,\ t)$, mentre ogni arco e, appartenente all'insieme degli archi **E**, rappresenta una traiettoria.
L'algoritmo prevede come prima fase la localizzazione del veicolo per aggiornare lo stato iniziale denominato x_0, a partire dal nodo $(x_0,\ t_0)$ possiamo iniziare a creare l'albero. Dopo aver simulato l'andamento per il tempo di pianificazione, T_{cycle}, si aggiorna la posizione corrente con $(x_1,\ t_0 + T_{cycle})$ e viene inserito nella coda **Q**. Per ciascun nodo in un determinato livello dell'albero viene eseguita una pop dalla coda.

Lo sviluppo dei nodi dell'albero avviene basandosi su un insieme **V** di velocità di riferimento, v_{ref}, e un altro di offset laterali **D**, d_{ref}, rispetto al path di riferimento. Grazie al modello di simulazione, alla legge di controllo e al tempo di simulazione, si crea un nodo per lo stato del veicolo e un arco entrante per la traiettoria. La coppia $(n,\ e)$ viene inserita nella coda **Q**.
Una volta completato l'albero vengono calcolati i costi di ogni arco, quindi viene scelta la traiettoria migliore, ovvero quella col costo più basso.

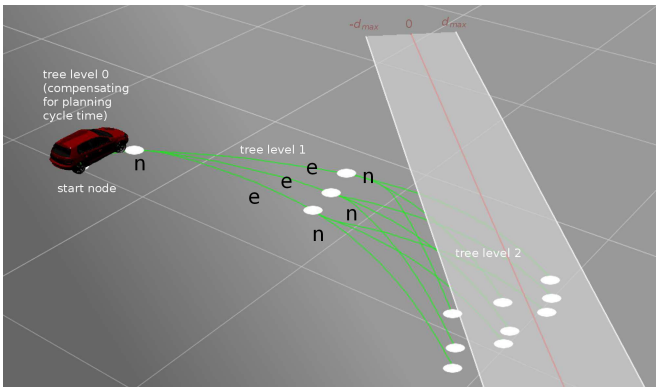

Figura 4.2: Un albero composto da segmenti di traiettoria conformi al sistema. Il livello zero dell'albero corrisponde alla parte iniziale della soluzione trovata nell'iterazione di ricerca precedente. Per i livelli successivi dell'albero il terminale è discretizzato in un insieme di tre campioni {-d max , 0, d max }.

Il vettore di stato utilizzato contiene:

- Posizione in 2D -> x e y
- θ -> direzione
- ϕ -> angolo di sterzo
- v -> velocità longitudinale

La dinamica del veicolo è data da:

$$\dot{x} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{\phi} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} cos\theta \cdot v \\ sin\theta \cdot v \\ \frac{v}{L} tan\phi \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$$

Nell'equazione sopra utilizziamo la velocità di sterzo, u_1, e l'accelerazione lineare, u_2, esse fanno parte del vettore di controllo u. La distanza inter-asse del veicolo è rappresentata da L. Inoltre sono imposti dei controlli in valore assoluto su diversi parametri: angolo di sterzata $\phi \leq \phi_{\max}$, velocità di sterzata $u_1 \leq \dot{\phi}_{\max}$, velocità longitudinale $v \leq v_{max}$, accelerazione longitudinale $u_2 \leq \dot{v}_{\max}$, decelerazione longitudinale $u_2 \geq \dot{v}_{\min}$.

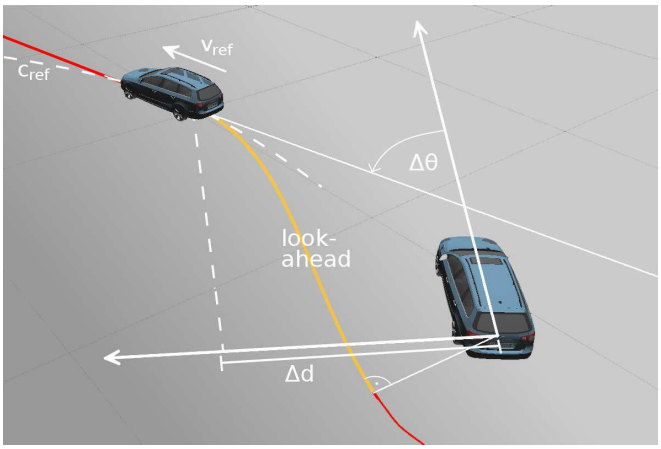

Figura 4.3: Variabili per il controllo laterale

Il controllore è progettato per seguire un Reference Cart, la cui posizione di riferimento è data dal centro del suo asse posteriore. Si tiene conto degli stati d'errore, differenza di orientamento, $\delta\theta$, e distanza laterale, δd. La posizione di riferimento viene calcolata in base al lookahead in funzione della velocità di riferimento lungo il percorso. Il controllo laterale per il grado di rotazione ω è dato da :

$$\omega = v_{\text{ref}} c_{\text{ref}} + k_1 v_{\text{ref}} \frac{sin\Delta\theta}{\Delta\theta} - k_2 \Delta\theta$$

4.2 Panoramica generale

Il programma ROS analizzato è composto da due classi fondamentali, la prima, MotionController, che contiene tutti i metodi relativi al calcolo delle traiettorie e alla simulazione dell'andamento e la seconda, MotionVisualizer, che contiene i metodi relativi alla visualizzazione dei risultati su RViz.
Inizialmente è stata analizzata la struttura del BaseLocalPlanner standard per il package Navigation, ed è stato deciso di suddividere il progetto su due classi principali, una in cui effettuare tutti i calcoli e una per interfacciarsi con il "Move Base" e scambiare dati. È stata reimplementata la classe MotionController con i suoi algoritmi di calcolo, precedentemente semi-sviluppata, e introdotta la classe IraLocalPlanner, per l'interfaccia con il sistema. Il mio progetto è composto dai seguenti file sorgente:

- structures.h - contiene le strutture utilizzate
- ira_local_planner.h - header della classe IraLocalPlanner
- ira_local_planner.cpp - nodo che comunica con il move_base
- motion_controller.h - header della classe MotionController

- motion_controller.cpp - nodo che si occupa dei calcoli

4.3 Strutture utilizzate

La struttura StateVector rappresenta lo stato del veicolo ed è così definita :

- **x** - coordinata x (x) - [double]
- **y** - coordinata y (y) - [double]
- **theta** - angolo di orientamento del veicolo (θ) - [double]
- **phi** - angolo di sterzo (ϕ) - [double]
- **v** - velocità longitudinale (v) - [double]
- **u1** - velocità di sterzo ($u1$) - [double]
- **u2** - accelerazione longitudinale ($u2$) - [double]

La struttura ErrorState rappresenta gli stati di errore del controllo laterale :

- **err_theta** - differenza di orientamento ($\Delta\theta$) - [double]
- **err_dist** - distanza laterale (Δd) - [double]

La struttura Reference il riferimento al reference path :

- **x_ref** - coordinata x di riferimento (x_{ref}) - [double]
- **y_ref** - coordinata y di riferimento (y_{ref}) - [double]
- **theta_ref** - orientamento di riferimento (θ_{ref}) - [double]
- **v_ref** - velocità di riferimento (v_{ref}) - [double]

La struttura Trajectory rappresenta una traiettoria e viene connessa alle altre tramite il vettore parentNodes:

- **startX** - coordinata x iniziale (x_{start}) - [double]
- **startY** - coordinata y iniziale (y_{start}) - [double]
- **endX** - coordinata x finale (x_{end}) - [double]
- **endY** - coordinata y finale (y_{end}) - [double]
- **endX** - coordinata x finale (x_{end}) - [double]

- **theta** - orientamento (θ) - [double]
- **phi** - angolo di sterzo (ϕ) - [double]
- **v** - velocità (v) - [double]
- **reference** - struttura Reference (*Reference*) - [Reference]
- **errorState** - struttura ErrorState (*ErrorState*) - [ErrorState]
- **cumErrDist** - distanze laterali cumulate ($\sum \Delta d$) - [double]
- **trajLength** - lunghezza della traiettoria (*TrajLength*) - [double]
- **cumTrajLength** - lunghezza cumulata ($\sum TrajLength$) - [double]
- **treeLevel** - livello dell'albero (*TreeLevel*) - [double]
- **parentNodes** - traiettorie associate (*ParentNodes*) - [vector<int>]
- **steerSpeedVect** - velocità di sterzo in ogni frazione della traiettoria (*SteerSpeedVect*) - [vector<double>]
- **accVect** - accelerazione in ogni frazione della traiettoria (*AccVect*) - [vector<double>]

I Topic pubblicati dal sistema sono:

- **l_plan_pub_** - gestisce e pubblica in RViz messaggi del tipo nav_msgs/Path, contenenti di volta in volta la traiettoria migliore calcolata
- **orientation** - gestisce e pubblica in RViz messaggi del tipo visualization_msgs/MarkerArray, contenenti la posizione futura simulata da cui iniziare a calcolare le nuove traiettorie mentre si finisce l'esecuzione di quella vecchia
- **trajectoryPublisher** - gestisce e pubblica in RViz messaggi del tipo visualization_msgs/MarkerArray, contenenti tutte le traiettorie calcolate

Mentre i Topic sottoscritti dal sistema sono:

- **tf_** - gestisce le trasformazioni tra i frame
- **odom_helper_** - utilizzato per ricevere la posizione corrente del veicolo tramite l'odometria
- **sub** - sottoscrive messaggi di tipo std_msgs/Float32 o gazebo_msgs/LinkStates

Infine è stato creato un file .yaml per poter passare determinati parametri all'esecuzione del programma. Per gestire questi parametri è stato necessario l'utilizzo del parameter server per leggere a runtime i parametri. Il parameter server è utilizzato dai nodi per immagazzinare e prelevare i parametri.
I parametri settabili sono:

- **delta_t** - tempo in cui suddividere il lookahead, deve essere uguale a 1 diviso la frequenza di chiamata del "Move Base", valore anch'esso settabile sempre da file .yaml ma come parametro del "Move Base" (*controller_frequency*) - [double]
- **lookahed_t** - tempo per cui eseguire la simulazione per ogni tree level - [double]
- **maxPathDistance** - massima distanza dal path entro cui una traiettoria può essere accettata - [double]
- **maxSteerAngle** - massimo angolo di sterzo - [double]
- **minSteerAngle** - minimo angolo di sterzo - [double]
- **maxSteerSpeed** - massima accelerazione angolare - [double]
- **maxAcceleration** - massima accelerazione longitudinale - [double]
- **maxDeceleration** - massima decelerazione angolare - [double]
- **treeLevels** - profondità di propagazione dell'albero - [int]
- **errorTheta** - soglia sotto cui l'errore viene considerato zero - [double]
- **k1** - peso d'errore sulla distanza - [double]
- **k2** - peso d'errore sull'orientamento - [double]
- **l** - lunghezza inter-asse del veicolo - [double]
- **steer_kP** - controllo di sterzata - [double]
- **speed_kP** - controllo di velocità - [double]
- **p_len_scale** - scala per gli errori di orientamento - [double]
- **p_dist_scale** - scala per gli errori di distanza - [double]
- **p_dist_perc** - percentuale di peso per gli errori sulla distanza - [double]
- **p_align_perc** - percentuale di peso per gli errori sull'orientamento - [double]

- **vref** - vettore contenente le velocità di riferimento desiderate - [vector<double>]
- **lateralOffset** - vettore contenente - [vector<double>]
- **cart_flag** - variabile che indica se la simulazione è sul cart o meno - [boolean]
- **odom_topic** - nome del topic da sottoscrivere per l'odometria - [string]
- **robot_frame** - nome del frame del sistema di riferimento del cart - [string]
- **local_fix_frame** - nome del frame del sistema di riferimento della mappa - [string]

4.4 Libreria IraLocalPlanner

Questa classe si interfaccia direttamente con move_base, si occupa di chiamare il motion_controller per la pianificazione e rielabora i dati ricevuti per mandarli al base_controller rispettando la frequenza impostata.
Il metodo costruttore prende in input il nome del nodo, utilizzato per il nodehandle, tf per le trasformazioni e l'oggetto delle costmap, il distruttore si occupa di rimuovere dalla memoria gli oggetti allocati dinamicamente, in questo caso solo l'oggetto della classe MotionController. È stata seguita la struttura del local planner predefinito, tenendo i metodi necessari che vengono chiamati dal Move Base durante l'esecuzione :

- **initialize**: Questo metodo è chiamato solo quando viene creata un'istanza della classe sotto forma di plugin. Ha tre parametri che sono il nome della classe settare il nodeHandle (un oggetto che controlla il nodo corrente), tf che è utilizzato per fare trasformazioni tra più sistemi di riferimento e un oggetto della classe Costmap. Al suo interno leggiamo i parametri inseriti attraverso il file di configurazione ".yaml" e quindi li assegniamo alle variabili corrispondenti, quindi instanziamo un oggetto della classe MotionController con i parametri necessari.
- **setPlan**: metodo chiamato ogniqualvolta viene creato un nuovo piano dal global plan, esso riceve come parametro il global plan, che consiste in un vettore di punti, e lo salva in una variabile locale. Ogni punto è un oggetto di tipo *geometry_msgs/PoseStamped* consiste in due oggetti uno per la posizione, x e y, e uno per l'orientamento. Utilizzando un oggetto di tipo *visualization_msgs/Marker* stampo il percorso ricevuto su RViz.

- **isGoalReached**: viene chiamato prima di richiedere delle velocità per verificare se il gol sia stato raggiunto o meno. Riceve la posizione corrente utilizzando le costmap e richiama il metodo goalDistance per verificare se è stato raggiunto.

- **computeVelocityCommands**: questa funzione viene chiamata da move_base per chiedere i comandi di velocità, la frequenza di chiamata può essere settata da file di configurazione. Inizialmente viene controllato subito il livello attuale dell'albero di simulazione, se questo coincide con l'ultimo livello allora viene chiamato il thread *ackerSimulationBoost*. Questo thread è utilizzato per chiamare *MotionController/ackermanForwardSimulation* in modo tale da calcolare la traiettoria successiva mentre sta ancora eseguendo quella precedente così da non accumulare ritardi. All'interno vengono letti i parametri attuali del veicolo, i valori delle velocità ricevuti dall'odometria, *robot_vel*, e la posizione attuale nel sistema di riferimento globale calcolata tramite tf, *global_pose*. Quindi viene simulato l'andamento del cart per trovare i dati della posizione di arrivo da passare come parametri al thread che verrà chiamato.
  Una volta terminata l'esecuzione del thread viene ricevuta la traiettoria da eseguire e viene stampata su RViz sottoforma di *nav_msgs/Path*, questa traiettoria in sostanza può essere definita local plan.
  Successivamente vengono utilizzate due variabili di appoggio per mantengono il valore della velocità, *prevel_*, e dell'angolo di sterzo, *prevphi_*, dell'ultima traiettoria eseguita. Se viene eseguita la prima traiettoria allora saranno inizializzate con i valori letti in precedenza altrimenti non sarà necessaria un'inizializzazione. Quindi vengono aggiornate *prevel* sommando *accVect* moltiplicato per il tempo di simulazione e *prevphi* sommando *steerSpeedVect* moltiplicato sempre per il tempo di simulazione. Infine viene riempito il messaggio di tipo *geometry_msgs/Twist* con la velocità longitudinale e la velocità di sterzo, il quale verrà letto dal base_controller che attuerà i comandi.

Sono stati implementati anche altri due metodi di supporto, clearTraj che si occupa di pulire la variabile contenente la traiettoria corrente e goalDistance che calcola la distanza attuale dal goal.

4.5 Libreria MotionController

È stata presa l'omonima classe alla quale sono state applicate molte modifiche, sia per aggiungere funzionalità che per ottimizzare e velocizzare l'esecuzione. Questa classe è dedita alla pianificazione e al calcolo del moto della macchina autonoma, per sviluppare l'algoritmo di calcolo è stato utilizzato il framework descritto nel

paragrafo 4.1. Il metodo costruttore prende in input tutti i parametri descritti nel paragrafo 4.3 e settabili dall'utente tramite il file ".yaml".

- **setPath**: viene chiamato ogni qualvolta si imposta un nuovo goal di arrivo per aggiornare il global plan, riceve in ingresso un messaggio nav_msgs/Path. Questo path viene salvato in un rTree, gli rTree sono un tipo di albero usato per indicizzare spazi multidimensionali, come coordinate spaziali per dati geografici - [void]

- **getRealPhi**: viene chiamato per leggere il valore attuale di ϕ, l'angolo di sterzo - [double]

- **updateSteerCallback**: è utilizzato per aggiornare lo stato di ϕ. ci sono due metodi diversi in base all'ambiente di simulazione, in ingresso uno riceve un messaggio di tipo gazebo_msgs/LinkStates mentre l'altro un messaggio di tipo std_msgs/Float32 - [void]

- **normalPdf**: calcola la distribuzione normale di probabilità. In ingresso riceve i seguenti parametri: un valore x, la media μ e la deviazione standard σ - [double]

- **getLateralOffsetAtCurvilinearAbscissa**: calcola i punti laterali utilizzati come punti di arrivo nelle traiettorie, prende in ingresso una Pose di riferimento, ovvero x y e θ, e l'offset di riferimento, d_{ref} - [Reference]

- **computePathErrors**: calcola lo stato d'errore, ErrorState, prendendo il punto di riferimento terminale, x_{ref} e y_{ref}, la velocita di riferimento, v_{ref}, e l'orientamento di riferimento, θ_{ref}. Viene calcolata la differenza di orientamento, $\Delta\theta$, e la distanza laterale, Δd - [ErrorState]

- **computeDesiredTurningRate**: utilizza la formula vista nel paragrafo 4.1 per calcolare ω, il grado di rotazione desiderato. Prende in ingresso il vettore con gli stati d'errore, ErrorState, la velocità longitudinale corrente, v, la velocità di riferimento, v_{ref} e l'ascissa curvilinea di riferimento, c_{ref} - [double]

- **computeEndIndexInPathDistance**: calcola la posizione terminale sul path a cui fare riferimento per la futura generazione di traiettorie. Inizialmente viene calcolata la distanza massima che può essere percorsa in base al *lookahead* e a v_{ref}, quindi utilizzando una query sul rtree troviamo tutti i possibili punti raggiungibili in quella distanza e calcoliamo il più vicino alla posizione calcolata - [node_pair]

- **computeF**: computa lo stato del veicolo in un periodo di tempo Δt, prende in input un vettore di stato, StateVector, e verrà restituito aggiornato - [StateVector]

I prossimi metodi sono quelli più importanti e quindi verranno analizzati più nel dettaglio.
Questi metodi sono: *ackermanForwardSimulation*, *simulateTreeEdge* e *evaluateTrajectories*.

- **ackermanForwardSimulation**: questa funzione, presente nell'appendice A.1, si occupa della generazione delle traiettorie, della pianificazione del moto, successivamente della valutazione e dell'attuazione del movimento.
  Viene subito inizializzato l'algoritmo ripulendo i due vettori *trajectories*, che memorizza le strutture di tipo Trajectory, e *trajectoryQueue*, che memorizza gli indici delle traiettorie e funge da coda. Con una query sul rtree viene trovato il punto del path più vicino alla posizione attuale e inserita la distanza nella variabile *pathDistance*. Vengono inizializzati il contatore delle traiettorie a zero, *trajectoryCounter*, e una struttura *trajectory* con i valori dello stato attuale del veicolo. Questa traiettoria viene aggiunta al vettore *trajectories* e il contatore delle traiettorie aggiunto in coda nel vettore *trajectoryQueue* e incrementato di uno.
  Nei quattro cicli for annidati vengono generate le traiettorie. Il primo effettua un ciclo per ogni livello dell'albero, al suo interno viene creato un vettore *tempIndexes*, che verrà riempito con gli indici delle traiettorie la cui distanza rispetto al path è inferiore alla distanza massima, settata come parametro dall'utente.
  Il secondo effettua un'iterazione per ogni indice i contenuto in *tempIndexes*, ogni ciclo viene prelevata una traiettoria dal vettore *trajectories* nella posizione i-esima. Il terzo preleva i valori contenuti in *vRef_*, esso rappresenta le velocità di riferimento che il veicolo può applicare. Il quarto si basa sulle distanze laterali dal path, contenute nel vettore *lateralOffset_*.
  All'interno viene calcolata l'ipotetica posizione finale sul path, richiamando la funzione *computeEndIndexInPathDistance*. Successivamente con la funzione *getLateralOffsetAtCurvilinearAbscissa* viene calcolato un punto di riferimento di arrivo della nuova traiettoria, con anche il suo orientamento. Quindi viene eseguita la simulazione di percorrenza della traiettoria con la funzione *simulateTreeEdge*, la quale restituisce un vettore di stato.
  A questo punto si crea una nuova traiettoria utilizzando come punti di partenza quelli della traiettoria iniziale, e quelli terminali basandosi sul nuovo vettore di stato calcolato sopra. Vengono impostati i parametri della struttura ErrorState con le relative differenze sulla distanza, *pathDistance*, e sull'orientamento, *heading_error*, infine viene impostato il livello dell'albero e aggiornati i *parentNodes*. Quindi viene creato un oggetto visualization_msgs/Marker per stampare su rViz la traiettoria calcolata. Terminati i cicli viene determinata la traiettoria migliore utilizzando il metodo *evaluateTrajectories*.

- **simulateTreeEdge**: questa funzione, presente nell'appendice A.2, ha un ruolo fondamentale nella pianificazione del movimento del veicolo. Suddivide la simulazione di una traiettoria in più parti e per ogni parte ne calcola il moto da applicare. Infine restituisce il vettore di stato, StateVector, calcolato dopo aver percorso la traiettoria simulata.
  Viene subito inizializzato un vettore di stato con alcuni parametri di ingresso (x, y, θ, v, ϕ), la lunghezza della traiettoria è inizializzata a zero.
  Quindi viene fatto partire un ciclo suddividendo il tempo di *lookahead* in frazioni temporali più piccole, Δt, e per ogni istante di tempo verrà calcolato lo stato del veicolo. Con la funzione *computePathErrors* vengono calcolati gli errori di orientamento e distanza laterale, $\Delta\theta$ e Δd. Questi valori sono utilizzati nella funzione *computeDesiredTurningRate* per calcolare la velocità angolare del veicolo, ω. Vengono poi calcolati gli errori di sterzo, $\Delta\phi$, e di velocità, Δv. Utilizzando i controlli di sterzata, *steer_kp*, e di velocità, *speed_kp*, vengono calcolate le velocità di sterzo, *u1*, e l'accelerazione longitudinale, *u2*.
  Infine viene aggiornata la nuova lunghezza dell'arco nella variabile *trajectoryLength*. Terminato il tempo di simulazione viene fornito il vettore di stato del veicolo finale.

- **evaluateTrajectories**: questa funzione, presente nell'appendice A.3, valuta le traiettorie generate e restituisce l'indice della traiettoria migliore da applicare. Prende come parametri il vettore contenente le traiettorie calcolate, *allTrajectories*, e la profondità dell'albero, *tree_level*.
  All'inizio creo tre vettori: *cum_lenghts* per le lunghezze cumulate, *cum_distances* per le distanze cumulate e *alignement_errs* per gli errori di allineamento. Verranno poi popolati con i dati rispettivi delle traiettorie all'ultimo livello dell'albero di ricerca.
  Quindi viene instanziato e popolato il vettore delle distanze cumulate pesate, *cum_distances_weighted*. Vengono impostati *p_d* e *p_l* per pesare l'errore medio di distanza e di allineamento, che vengono calcolati e inseriti nei vettori *p_dist* e *p_align*. Questi valori saranno poi pesati e sommati nel vettore finale, *p_final*.
  Infine viene restituito l'indice della migliore traiettoria, trovato prendendo l'indice del massimo valore contenuto nel vettore *p_final* e lo scalare *sum*.

5 Test

In questo capitolo verranno spiegati i vari passaggi che sono stati eseguiti nella fase di test del programma. Questa fase può essere suddivisa in due parti:

- simulazione su PC in laboratorio, che è servita molto all'inizio per capire il comportamento del veicolo con il mio progetto
- prove reali direttamente sul cart, per applicare quanto testato in laboratorio nell'ambiente reale

Per simulare il programma bisogna lanciare il nodo di RViz e quello di Gazebo. È stato utilizzato un file di lancio per usare il modello 3d citato nel paragrafo 1.3. Inoltre è stato necessario creare un file di lancio, ".launch", per eseguire il nodo move_base e impostare tutti i suoi argomenti. In questo modo si possono scegliere quali plugin utilizzare e richiamare i rispettivi file ".yaml" contenenti i parametri per ogni plugin. I plugin impostati sono, per ciascun "interfaccia":

- **base_global_planner**:
  sono state effettuate prove con diversi global planner, inizialmente è stato usato il plugin *global_planner/GlobalPlanner* standard, poi successivamente quello sviluppato dal mio collega Riccardo Candrina nell'elaborato *"Implementation of a global planner operating with OSM for the USAD vehicle"*, chiamato *ira_global_planner/IraGlobalPlanner*. Rispetto al plugin base, il secondo fornisce un percorso di navigazione basato sulle mappe di OSM[1], Open Street Map, queste mappe, immagine 5.1 vengono scaricate in locale, rielaborate e poi viene calcolato un percorso su di esse.
- **base_local_planner**:
  inizialmente è stato provato *base_local_planner/TrajectoryPlannerROS* standard per capirne il funzionamento. Una volta iniziato lo sviluppo del mio progetto è iniziata pari passo la fase di sperimentazione del plugin *ira_local_planner/IraLocalPlan*

[1]Raggiungibile al seguente link:`https://www.openstreetmap.org/`

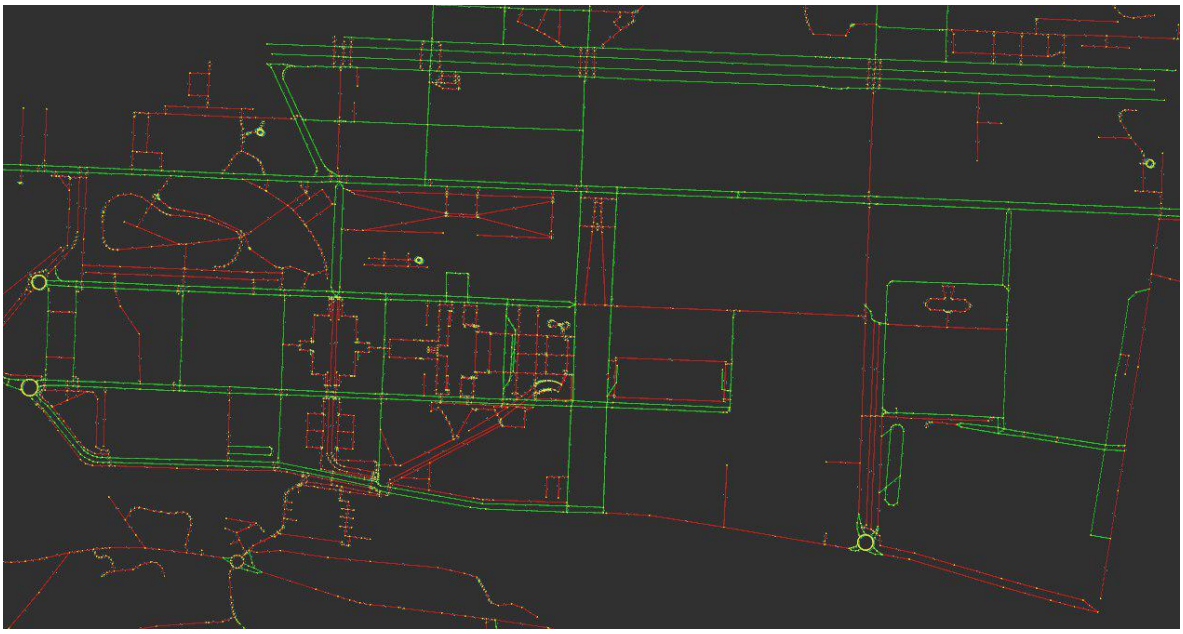

Figura 5.1: Questa è una porzione della mappa utilizzata nel plugin *ira_global_planner/IraGlobalPlanner* per la navigazione.

Nei paragrafi successivi verranno elencate le simulazioni eseguite utilizzando sia *ira_local_planner/IraLocalPlanner* che i vari *global_planner* appena descritti.

5.1 ira_local_planner/IraLocalPlanner e global_planner/GlobalPlanner

Come si può vedere nell'immagine 5.2, il percorso calcolato da questo global è adatto principalmente a veicolo dotati di cinematica olonomica, la quale permette una rotazione del veicolo su sé stesso. Usando il global_planner/GlobalPlanner, il problema principale consiste in un errato orientamento, poiché il percorso viene calcolato tenendo presente solo la posizione di partenza e non l'orientamento. Per rendere più veritiere possibile le simulazioni è stato necessario cambiare il *global_planner*.

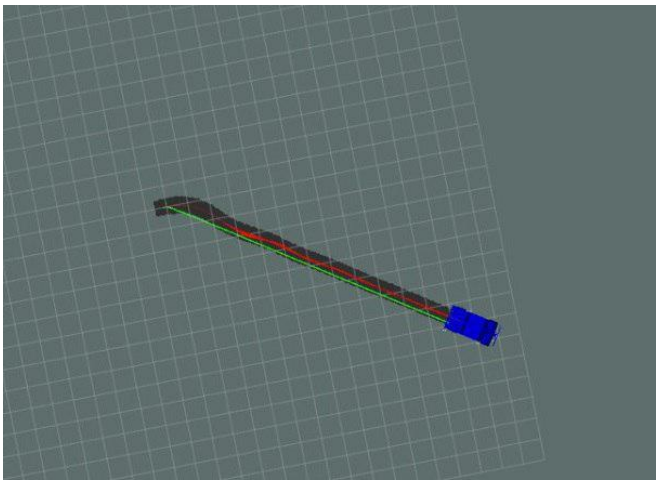

Figura 5.2: Simulazione effettuata utilizzando *ira_local_planner/IraLocalPlanner* e *global_planner/GlobalPlanner*. In verde viene tracciato il global plan, mentre in rosso sono tracciate le traiettorie calcolate in ogni posizione. La zona grigio scuro rappresenta il percorso dal punto di partenza all'arrivo.

5.2 ira_local_planner/IraLocalPlanner e percorso predefinito

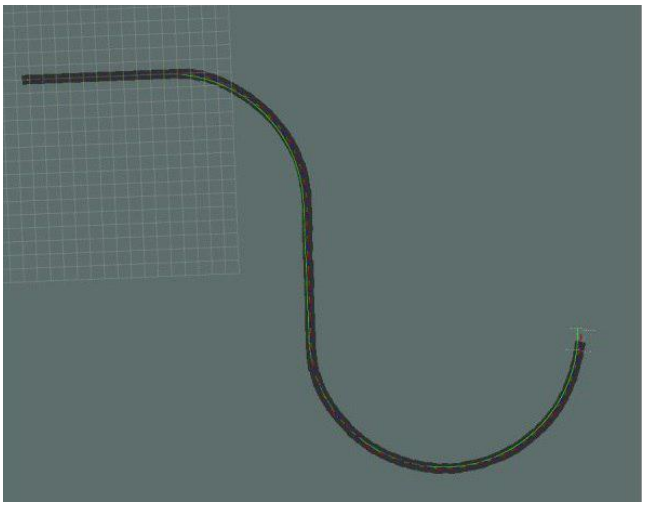

Figura 5.3: Simulazione effettuata utilizzando *ira_local_planner/IraLocalPlanner* e un percorso base creato direttamente nel codice per simulare un percorso realizzabile dal veicolo. Come nell'immagine precedente si può notare il global plan in verde e la zona grigio scuro rappresentante il percorso effettuato.

In questo esperimento, è stato implementato un percorso direttamente nel codice con lo scopo di riuscire a simulare l'andamento su un piano realistico. Come si può

notare nell'immagine 5.3, il percorso presenta delle curve molto ampie e riproducibili nella realtà. Il veicolo riesce a seguire bene il percorso, variando l'impostazione dei parametri il risultato cambia. Sono state effettuate molte prove per riuscire a trovare una combinazione ideale di parametri.

5.3 ira_local_planner/IraLocalPlanner e ira_global_planner/IraGlobalPlanner

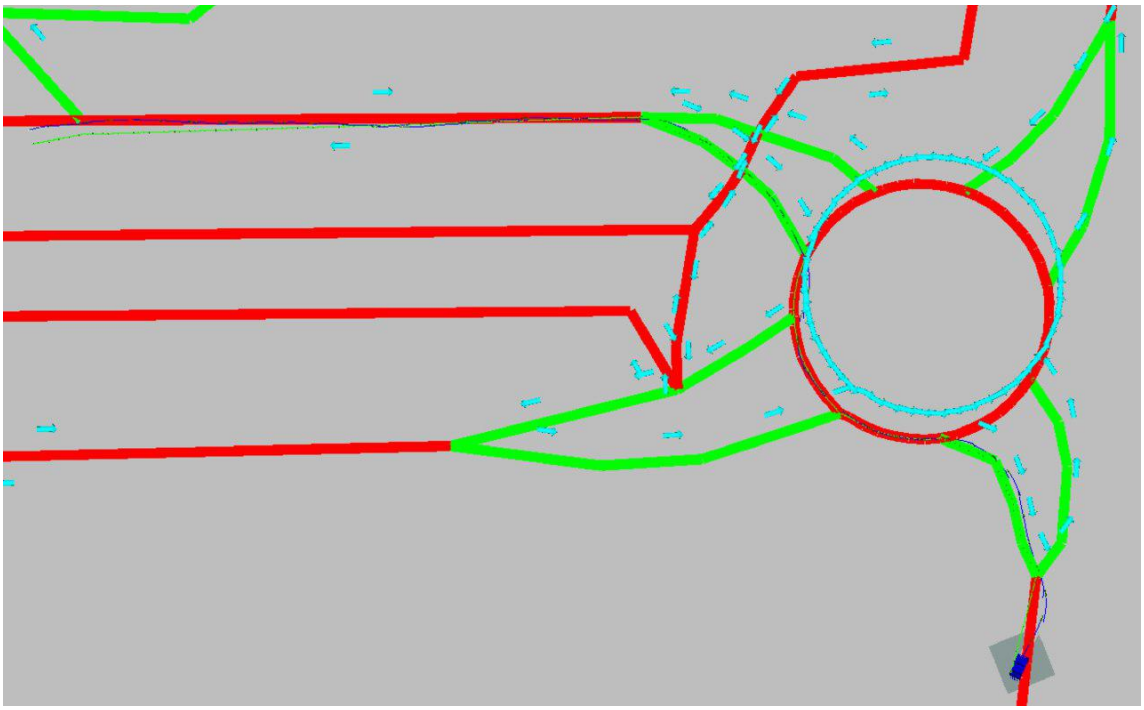

Figura 5.4: Simulazione effettuata utilizzando *ira_local_planner/IraLocalPlanner* e *ira_global_planner/IraGlobalPlanner*. Si possono notare le strade verdi che sono a senso unico, mentre quelle rosse a doppio senso, il percorso calcolato dal global è tracciato in verde mentre il percorso effettuato dal local in blu.

Come si può notare nell'immagine 5.4, grazie a questo *global_planner* è possibile avere una mappa geografica realistica di un ambiente. Abbiamo utilizzato sempre la stessa mappa che raffigura all'incirca tutta la zona bicocca di Milano e abbiamo simulato dei percorsi tra gli edifici dell'università.

5.4 Prove sul cart

Nell'ultimo periodo di stage sono state svolte una serie di prove direttamente sul veicolo. Inizialmente sono stati riadattati i topic per il sistema del cart. Questa fase ha richiesto l'identificazione manuale dei parametri giusti per essere in regola con le caratteristiche del veicolo. In particolare sono stati modificati anche i topic di comunicazione con gli altri nodi del sistema.
Si è notato che purtroppo, come mostrato nell'immagine 5.5, non sempre il veicolo riesce a seguire alla perfezione il percorso calcolato. C'è un minimo errore nell'esecuzione, per cui più veniva aumentato il tempo di simulazione, *lookahead*, maggiore era l'errore nell'applicazione della traiettoria.

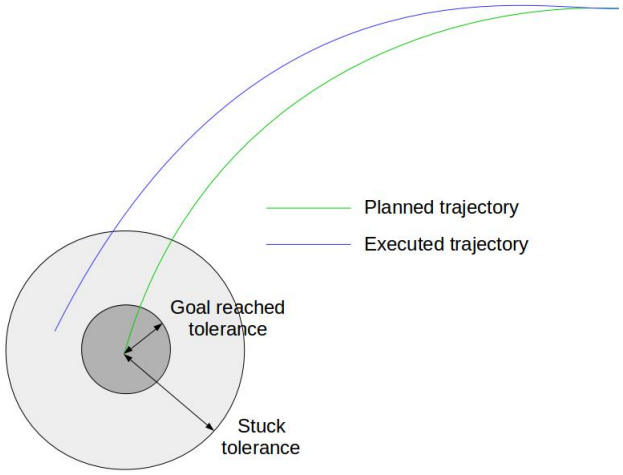

Figura 5.5: Esempio di situazione in cui il robot non segue precisamente la traiettoria calcolata.

In sintesi i migliori risultati sono stati ottenuti con la seguente configurazione dei parametri principali:

- **lookahed_t**: un valore basso genera errori minori, in genere è stato tenuto tra 1 e 1.5.
- **delta_t**: il valore deve essere relazionato al parametro che gestisce la frequenza di chiamata da parte del move_base, controller_frequency. In genere è stato a tenuto a 0.1/0.5 con freuenza a 10/20.
- **maxSteerSpeed**: è stata impostata con un valore compreso tra 0.20 e 0.30.
- **maxAcceleration**: impostato con un valore compreso tra 0.5 e 1.5.
- **treeLevels**: impostato a 3, se abbassato l'errore diminuisce però la simulazione risulta più lenta.

- **vref**: sono stati usati due valori di vRef, 0,5 e 1,5. Abbiamo provato ad aggiungere un altro valore però l'algoritmo impiegava più tempo a computare.
- **lateralOffset**: sono stati impostati tre valori di offset, quello centrale è rimasto fisso a zero, gli altri due devono essere simmetrici e sono stati provati nel range da 0.5 a 5. I risultati ottimali sono stati ottenuti utilizzando 4.0, abbassandolo troppo le traiettorie nelle curve risultavano troppo disallineate, alzandolo troppo le traiettorie si discostavano dal percorso.

Sono giunto a questa conclusione analizzando il comportamento in seguito al cambio dei parametri.

6 Conclusioni e sviluppi futuri

Il progetto *Implementation of a motion planning algorithm for the USAD vehicle within the ROS navigation stack* ha riguardato lo sviluppo di un plugin in ROS per la pianificazione e il controllo del moto di una macchina autonoma. Si è partiti con lo studio del paper *"A Sampling-Based Partial Motion Planning Framework for System-Compliant Navigation along a Reference Path"*, in cui veniva presentato un algoritmo capace di pianificare e controllare il movimento di un veicolo lungo un percorso dato in input. Si è poi passati ad un'attenta analisi dell'implementazione in MATLAB di un prototipo basato sull'algoritmo descritto, cercando di capirne la logica e i risultati ottenuti.

Siccome il progetto prevedeva lo sviluppo di un plugin all'interno del pacchetto "move_base" di ROS è stato necessario uno studio del funzionamento di ROS e in particolare dello stack di navigazione. L'obiettivo perseguito era quello di creare un Local Planner che generasse traiettorie utilizzando l'algoritmo descritto, e renderlo integrabile e adattabile al sistema di navigazione *move_base*. Una volta identificati i punti cardine del prototipo, si è passati all'implementazione degli stessi utilizzando strutture apposite e replicandone il comportamento delle funzioni. Di pari passo è stato analizzato il funzionamento di "move_base" e una volta appreso appieno il funzionamento della comunicazione tra i componenti interni, è stato possibile strutturare il mio progetto.

Una volta concluso lo sviluppo del plugin in ROS si è passati alla fase di testing mediante il tool di visualizzazione RViz. Questo ha reso possibile visualizzare il modello del veicolo muoversi seguendo la traiettoria migliore tra quelle calcolate e visualizzate, cercando di seguire al meglio il percorso ricevuto in input. Sono state eseguite diverse simulazioni, utilizzando diversi tipi di percorsi in input e cambiando i parametri d'ingresso. Al variare dei parametri ogni esecuzione forniva risultati differenti e si è cercata una combinazione che seguisse al meglio il percorso.

Viste le tempistiche dello stage, ho potuto fare solo alcuni test, che comunque hanno validato quello che ho sviluppato in situazioni base. In futuro sarà necessaria una

ulteriore fase di sperimentazione su strada atta a raffinare il comportamento del sistema sul veicolo del laboratorio. Uno sviluppo futuro già avviato è legato alla gestione degli ostacoli, sia statici che mobili, per cui sarà necessaria una funzione per la rilevazione di questi e per il ricalcolo del percorso in modo dinamico. A questo proposito una funzione che ci sentiamo di suggerire è la creazione di un plugin di recovery per un veicolo Ackerman per la gestione degli ostacoli.

Questo progetto è stato utile per capire il funzionamento della programmazione del moto per le macchine a guida autonoma, un campo che negli ultimi anni è cresciuto molto e che presto vedrà lo sviluppo di automobili completamente autonome.

A Codice

A.1 ackermanForwardSimulation

```
void MotionController::ackermanForwardSimulation(tf::Stamped<tf::Pose> global_pose,
     vector<Trajectory>& best_trajectories, double deltaX, double deltaY, double
    theta, double velocityFinal, double phi, double velStear)
{
    trajectoryQueue.clear();
    trajectories.clear();
    Eigen::Vector3f pos(global_pose.getOrigin().getX(),
          global_pose.getOrigin().getY(), tf::getYaw(global_pose.getRotation()));
    vector<node_pair> result;
    rtree_path.query(bgi::nearest(point(pos[0], pos[1]), 1),
          std::back_inserter(result));
    double pathDistance = sqrt(pow(pos[0] - result.front().first.get<0>(), 2) +
          pow(pos[1] - result.front().first.get<1>(), 2));
    int trajectoryCounter = 0;
    Trajectory trajectory;
    trajectory.startX = pos[0] + deltaX;
    trajectory.startY = pos[1] + deltaY;
    trajectory.endX = pos[0] + deltaX;
    trajectory.endY = pos[1] + deltaY;
    trajectory.theta = theta;
    trajectory.phi = phi;
    trajectory.v = velocityFinal;
    trajectory.steerSpeed = velStear;
    trajectory.errorState.err_dist = pathDistance;
    trajectory.cumErrDist = 0;
    trajectory.errorState.err_theta = 0;
    trajectory.trajLength = 0;
    trajectory.cumTrajLength = 0;
    trajectory.treeLevel = 0;
    trajectory.parentNodes.clear();
    trajectories.push_back(trajectory);
    trajectoryQueue.push_back(trajectoryCounter);
    trajectoryCounter++;
    int treeLevelIndex;
    for (treeLevelIndex = 0; treeLevelIndex < treeLevels_; treeLevelIndex++)
    {
        vector<int> tempIndexes;
        for (int i = 0; i < trajectoryQueue.size(); i++)
        {
            int tempIndex = trajectoryQueue.at(i);
```

```
            Trajectory aTrajectory = trajectories.at(tempIndex);
            if (aTrajectory.errorState.err_dist < maxPathDistance_)
                tempIndexes.push_back(tempIndex);
        }
        trajectoryQueue.clear();
        for (int i = 0; i < tempIndexes.size(); i++)
        {
            Trajectory tempTrajectory = trajectories[tempIndexes[i]];
            for (int vRefIndex = 0; vRefIndex < vRef_.size(); vRefIndex++)
            {
                for (int lateralOffsetIndex = 0; lateralOffsetIndex <
                    lateralOffset_.size(); lateralOffsetIndex++)
                {
                    node_pair end = computeEndIndexInPathDistance(
                    tempTrajectory.endX, tempTrajectory.endY, vRef_.at(vRefIndex));
                    geometry_msgs::PoseStamped aPoseStamped;
                    aPoseStamped.pose.position.x = end.first.get<0>();
                    aPoseStamped.pose.position.y = end.first.get<1>();
                    aPoseStamped.pose.orientation = end.second;
                    Reference newReference = getLateralOffsetAtCurvilinearAbscissa(
                        aPoseStamped.pose, lateralOffset_.at(lateralOffsetIndex));
                    double x_ref = newReference.x_ref;
                    double y_ref = newReference.y_ref;
                    double theta_ref = tf::getYaw(aPoseStamped.pose.orientation);
                    vector<double> sVect;
                    vector<double> aVect;
                    StateVector aStateVector = simulateTreeEdge(
                         tempTrajectory.endX, tempTrajectory.endY,
                         tempTrajectory.theta, tempTrajectory.v,
                         tempTrajectory.phi, vRef_[vRefIndex], x_ref,
                         y_ref, theta_ref, sVect, aVect);
                    result.clear();
                    rtree_path.query(bgi::nearest(point(aStateVector.x,
                        aStateVector.y), 1), std::back_inserter(result));
                    pathDistance = sqrt(
                         pow(aStateVector.x-result.front().first.get<0>(), 2) +
                         pow(aStateVector.y-result.front().first.get<1>(), 2));
                    double heading_error = tf::getYaw(result.front().second) -
                        aStateVector.theta;
                    Trajectory aTrajectory;
                    aTrajectory.startX = tempTrajectory.endX;
                    aTrajectory.startY = tempTrajectory.endY;
                    aTrajectory.endX = aStateVector.x;
                    aTrajectory.endY = aStateVector.y;
                    aTrajectory.theta = aStateVector.theta;
                    aTrajectory.phi = aStateVector.phi;
                    aTrajectory.v = aStateVector.v;
                    aTrajectory.steerSpeed = aStateVector.u1;
                    aTrajectory.accX = aStateVector.u2;
                    aTrajectory.reference.x_ref = x_ref;
                    aTrajectory.reference.y_ref = y_ref;
                    aTrajectory.reference.theta_ref = theta_ref;
                    aTrajectory.reference.v_ref = vRef_[vRefIndex];
                    aTrajectory.errorState.err_dist = pathDistance;
                    aTrajectory.cumErrDist = tempTrajectory.cumErrDist +
                         fabs(pathDistance);
                    aTrajectory.errorState.err_theta = heading_error;
                    aTrajectory.trajLength = trajectoryLenght;
                    aTrajectory.cumTrajLength = tempTrajectory.cumTrajLength +
                        trajectoryLenght;
                    aTrajectory.treeLevel = treeLevelIndex;
```

```
                    aTrajectory.parentNodes = tempTrajectory.parentNodes;
                    aTrajectory.parentNodes.push_back(trajectoryCounter);
                    aTrajectory.accVect = aVect;
                    aTrajectory.steerSpeedVect = sVect;
                    trajectories.push_back(aTrajectory);
                    trajectoryQueue.push_back(trajectoryCounter);
                    visualization_msgs::Marker item;
                    geometry_msgs::Point tail, head;
                    tail.x=aTrajectory.startX;
                    head.x=aTrajectory.endX;
                    tail.y=aTrajectory.startY;
                    head.y=aTrajectory.endY;
                    tail.z=4;
                    head.z=4;
                    item.points.clear();
                    item.action=visualization_msgs::Marker::ADD;
                    item.type = visualization_msgs::Marker::ARROW;
                    item.header.stamp=ros::Time::now();
                    item.header.frame_id= local_fix_frame_;
                    item.color.r=1.0f;
                    item.color.a=0.1f;
                    item.scale.x=0.01;
                    item.scale.y=0.01;
                    item.scale.z=0.01;
                    item.ns = "arrow";
                    item.id=id_marker++;
                    item.points.push_back(tail);
                    item.points.push_back(head);
                    array.markers.push_back(item);
                    trajectoryCounter += 1;
                }
            }
        }
    }
    int best_traj_idx = evaluateTrajectories(trajectories, treeLevelIndex - 1);
    Trajectory best_trajectory = trajectories.at(best_traj_idx);
    for(int i = 0; i < best_trajectory.parentNodes.size(); i++)
        best_trajectories.push_back(trajectories.at(
          best_trajectory.parentNodes.at(i)));
    trajectoryPublisher.publish(array);
}
```

A.2 simulateTreeEdge

```
StateVector MotionController::simulateTreeEdge(double inX, double inY, double
    inTheta, double inV, double inPhi, double inVRef, double inXRef, double inYRef,
     double inThetaRef, vector<double> &sVect, vector<double> &aVect)
{
    if((inThetaRef>=-0.1 && inTheta>=-0.1) || (inThetaRef<=0.1 && inTheta<=0.1))
           //ROS_WARN_STREAM("concordi");
        else
            if(inThetaRef>0 && inTheta<0)
                inTheta = 2*M_PI + inTheta;
                else
                if(inThetaRef<0 && inTheta>0)
```

```
                    inTheta = inTheta - 2*M_PI;
    StateVector aStateVector;
    aStateVector.x = inX;
    aStateVector.y = inY;
    aStateVector.theta = inTheta;
    aStateVector.v = inV;
    aStateVector.phi = inPhi;
    trajectoryLenght = 0;
    for (double i = 0; i <= lookahead_t_ ; i = i + delta_t_)
    {
        ErrorState anErrorState = computePathErrors(aStateVector, inVRef, inXRef,
            inYRef, inThetaRef, lookahead_t_ - i);
        double omega = computeDesiredTurningRate(anErrorState, aStateVector.v,
            inVRef, 0);
        double steering_error;
        steering_error = atan(omega * l_ / aStateVector.v) - aStateVector.phi;
        if (std::isinf(atan(omega * l_/ aStateVector.v)))
            steering_error = 0.001f;
        if (std::isnan(atan(omega * l_ / aStateVector.v)))
            steering_error = 0.001f;
        aStateVector.u1 = steer_kP_ * steering_error;
        if (fabs(aStateVector.u1) > maxSteerSpeed_)
            aStateVector.u1 = copysign(1.0, aStateVector.u1) * maxSteerSpeed_;
        sVect.push_back(aStateVector.u1);
        double speed_error = inVRef - aStateVector.v;
        aStateVector.u2 = speed_kP_ * speed_error;
        if (aStateVector.u2 > maxAcceleration_)
            aStateVector.u2 = maxAcceleration_;
        if (aStateVector.u2 < maxDeceleration_)
            aStateVector.u2 = maxDeceleration_;
        aVect.push_back(aStateVector.u2);
        double old_x = aStateVector.x;
        double old_y = aStateVector.y;
        aStateVector = computeF(aStateVector);
        trajectoryLenght += sqrt(pow((aStateVector.x - old_x), 2) +
                                 pow((aStateVector.y - old_y), 2));
    }
    return aStateVector;
}
```

A.3 evaluateTrajectories

```
double MotionController::evaluateTrajectories(vector<Trajectory> &allTrajectories,
    int tree_level)
{
    vector<double> cum_lenghts;
    vector<double> cum_distances;
    vector<double> alignement_errs;
    Trajectory firstTrajectory = allTrajectories.at(0);
    double sum = 0;
    for (int i = 0; i < allTrajectories.size(); i++)
    {
        Trajectory aTrajectory = allTrajectories.at(i);
        if (aTrajectory.treeLevel < 0)
            continue;
```

```
        if (aTrajectory.treeLevel == tree_level)
        {
            cum_lenghts.push_back(aTrajectory.cumTrajLength);
            cum_distances.push_back(aTrajectory.cumErrDist);
            alignement_errs.push_back(aTrajectory.errorState.err_theta);
        }
        if (aTrajectory.treeLevel < tree_level)
            sum++;
    }
    vector<double> cum_distances_weighted;
    for (int i = 0; i < cum_distances.size(); i++)
    {
        double value = cum_distances.at(i) / cum_lenghts.at(i);
        value /= firstTrajectory.errorState.err_dist;
        cum_distances_weighted.push_back(value);
    }
    double p_d = normalPdf(0, 0, p_dist_scale_);
    double p_l = normalPdf(0, 0, p_len_scale_);
    vector<double> p_dist;
    vector<double> p_align;
    for (int i = 0; i < cum_distances_weighted.size(); i++)
    {
        double result = normalPdf(cum_distances_weighted.at(i), 0, p_dist_scale_) /
             p_d;
        p_dist.push_back(result);
        result = normalPdf(alignement_errs.at(i), 0, p_len_scale_) / p_l;
        p_align.push_back(result);
    }
    vector<double> p_final;
    for (int i = 0; i < p_dist.size(); i++)
    {
        p_final.push_back(p_dist_perc_ * p_dist.at(i) +
          p_align_perc_ * p_align.at(i));
    }
    int tmp_bes_traj_idx = 0;
    for (int i = 0; i < p_final.size(); i++)
    {
        if (p_final.at(tmp_bes_traj_idx) < p_final.at(i))
            tmp_bes_traj_idx = i;
    }
    double best_traj_idx = tmp_bes_traj_idx + sum;
    if(best_traj_idx == 0)
        best_traj_idx = 1;
    return best_traj_idx;
}
```

A Bibliografia

[1] Tully Foote. «tf: The transform library». In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 1–6.

[2] Thomas Moore e Daniel Stouch. «A generalized extended kalman filter implementation for the robot operating system». In: *Intelligent Autonomous Systems 13*. Springer, 2016, pp. 335–348.

[3] *Ros.org*.

[4] Ulrich Schwesinger et al. «A sampling-based partial motion planning framework for system-compliant navigation along a reference path». In: *Intelligent Vehicles Symposium (IV), 2013 IEEE*. IEEE. 2013, pp. 391–396.

[5] Bruno Siciliano et al. *Kinematics*. Springer, 2009.

[6] Sebastian Thrun, Wolfram Burgard e Dieter Fox. *Probabilistic robotics*. MIT press, 2005.