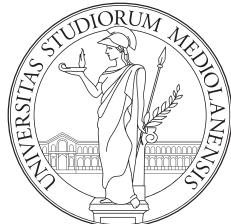


Università degli Studi di Milano
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI



CORSO DI LAUREA TRIENNALE IN INFORMATICA
CORSO DI LAUREA

STRATEGIE DI
RACELINE OPTIMIZATION
IN F1TENTH AUTONOMOUS RACING

Relatore: Nicola Basilico

Correlatore: Michele Antonazzi

Tesi di Laurea di:
Emanuele Manca
Matr. Nr. 978785

ANNO ACCADEMICO 2023-2024

Indice

1	Introduzione	1
1.1	F1TENTH	1
1.2	Autonomous Driving Pipeline	2
1.2.1	Perception	3
1.2.2	Planning	4
1.2.3	Control	5
1.3	ROS2	6
1.4	RViz	9
2	Planning	10
2.1	Local Planner	11
2.2	Behavioural Planner	13
2.3	Global Planner	13
3	Raceline Optimization	14
3.1	Le curve	14
3.2	Descrizione del problema	17
3.3	Path planning	20
3.4	Velocity planning	23
4	Implementazione	24
4.1	Generazione centerline	25
4.2	Esecuzione dell'ottimizzazione	28
4.2.1	Tuning dei parametri	30
5	Analisi dei risultati	33
5.1	Metriche	34
5.2	Spa	34
5.3	Monza	39
5.4	Tempo di Esecuzione	42
6	Conclusioni	44
6.1	Risultati ottenuti	44
6.2	Sviluppi futuri	44
Ringraziamenti		46
Bibliografia		47

Capitolo 1

Introduzione

1.1 F1TENTH

F1TENTH è una community internazionale di ricercatori, ingegneri e appassionati di sistemi autonomi che organizza competizioni di corsa di veicoli-robot con la peculiare caratteristica di essere grandi un *un decimo* di quelle di F1, da cui il nome. Oltre a questo, promuove la ricerca nell'ambito della guida autonoma e altri campi tra cui reinforcement learning, sistemi di comunicazione e robotica; offre, inoltre, una infrastruttura per costruire l'auto da corsa e sviluppare il software necessario per farla gareggiare. [10]

La community è stata fondata all'Università della Pennsylvania nel 2016 ma ha iniziato rapidamente collaborazioni con altre istituzioni e università in tutto il mondo – in Italia, alla data di stesura, solo con l'Università degli Studi di Modena e Reggio Emilia. [6]



Robot di F1TENTH [16]

Studio preliminare Al fine di comprendere al meglio gli argomenti della tesi, è stato effettuato uno studio preliminare su alcuni macro-argomenti che forniscono le basi fondamentali della guida autonoma e l'utilizzo della piattaforma di F1TENTH. Di seguito si elencano le aree tematiche:

- Sviluppo con ROS e l'ambiente simulatore di F1TENTH gym;
- **Metodi reattivi e dinamiche del veicolo:**
Algoritmo *PID Control* ed esempio di applicazione in Wall following, algoritmo reattivo *Follow the Gap*;

- **Mapping & Localization:**

Localizzazione e mapping simultaneo con *SLAM* e algoritmo di localizzazione *Particle Filter*;

- **Planning & Control:**

Stack di pianificazione e controllo di un veicolo autonomo, algoritmo di *path tracking Pure Pursuit*, introduzione ai *local planner* e algoritmi come RRT.

F1TENTH gym La community sviluppa e mantiene un simulatore open-source specifico per F1TENTH [7] in modo da poter sviluppare e testare comodamente senza l'uso di hardware specifico definito per la costruzione del robot.

Il simulatore permette di definire la dinamica del veicolo in modo tale da avere una simulazione più reale possibile.

1.2 Autonomous Driving Pipeline

Un progetto ben strutturato è più efficiente da mantenere e ha maggiori probabilità di funzionare correttamente: ciò si applica anche in questo caso.

Un progetto complesso come quello dei veicoli autonomi deve essere suddiviso in sotto-problemi più specifici e in modo tale che uniti insieme risultino nella soluzione del problema.

Il sistema di controllo *software* dei veicoli autonomi segue un *ciclo* di operazioni in cui il prodotto di una fase è input della successiva; si identificano in ordine tre fasi:

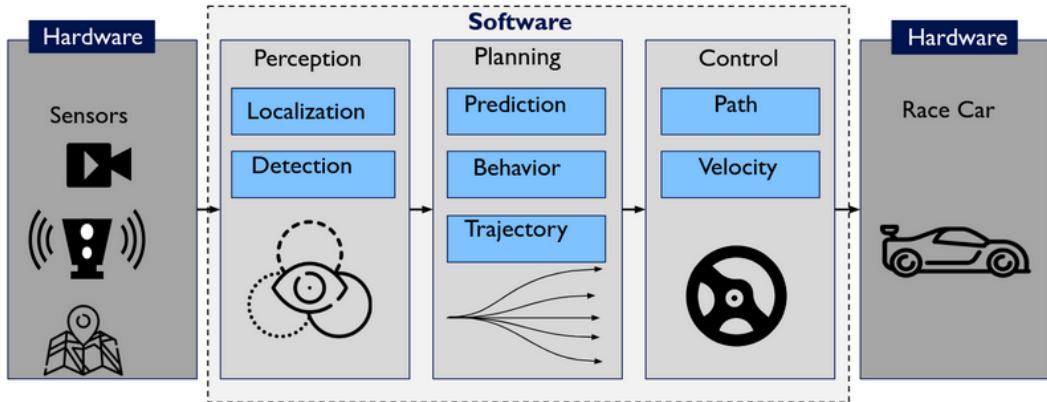
1. **Perception** – Attraverso sensori ottici e radar si *percepisce* il mondo attorno a sé, ci si localizza nella mappa e si identificano eventuali ostacoli o rivali;
2. **Planning** – Stando ai dati generati nella fase precedente, si determinano quali saranno le *mosse future* seguendo delle policy prescritte;
3. **Control** – Si generano dei comandi di sterzata e di velocità per attuare le scelte determinate nella fase precedente.

L'input per la prima fase (*Perception*) e l'output dell'ultima (*Control*) è direttamente l'hardware: dunque nella prima sono i sensori ottici e radar, come citato precedentemente, mentre nella seconda sono gli attuatori per i controllo della velocità e angolo delle ruote.

Il risultato che si vuole ottenere, quindi, è quello di una sequenza di comandi di sterzata e accelerazione in un contesto, quello delle corse, che richiede una certa velocità di esecuzione per poter essere il più reattivi possibile, per questo motivo

è preferibile eseguire il ciclo tra le 20 e 50 volte al secondo. Una rappresentazione grafica della pipeline intera si trova alla figura 1.

Figura 1: Pipeline per i veicoli autonomi [16]



Di seguito si analizzano più nel dettaglio le fasi sopra citate.

1.2.1 Perception

La prima fase ha il compito di *analizzare* i dati prodotti da diversi tipi di sensori come quelli ottici, laser e IMU (Inertial Measurement Unit) e ha come obiettivi quelli di: [1]

- determinare l'ambiente statico, chiamato anche *mappa*, che in questo contesto rappresenta i confini del tracciato in cui il robot gareggia;
- eventuali ostacoli o altri robot rivali, che quindi compongono la parte dinamica;
- *localizzarsi* all'interno della mappa, ovvero determinare posizione e orientamento del robot rispetto alla mappa.

In questa fase, quindi, si ritrovano algoritmi di object detection – di cui però non è stato argomento di studio per questa tesi – algoritmi di mapping e localization, come Particle Filter e SLAM (*Simultaneous Localization And Mapping*).

Localization Sebbene possano essere sfruttate tecnologie come il GPS, la ricerca nelle corse autonome si concentra maggiormente su una soluzione che richieda solo

l'uso di sensori ottici, laser e odometria. [1] Per effettuare la localizzazione del robot è necessario avere un modello della mappa.

Una tecnica spesso usata in questi ambiti è l'odometria, che permette di stimare la variazione della posizione del robot, del suo orientamento o della velocità da sensori che misurano lo spazio percorso delle ruote e dell'angolo di sterzata. Dunque, una prima soluzione potrebbe essere quella di partire da una posizione nota e di calcolare la posizione successiva integrando le misurazioni dei vari sensori e l'odometria delle ruote: questa tecnica viene chiamata *Dead Reckoning* e, sebbene possa funzionare correttamente in un simulatore, nella realtà le misurazioni dei sensori contengono del rumore e l'odometria delle ruote generalmente non ha la precisione richiesta per via di un possibile slittamento delle ruote stesse: tutto ciò, quindi, introduce nei calcoli degli *errori* che si accumulano col tempo, portando a risultati errati.

La soluzione è quella di rendere più robusti i calcoli tenendo conto del rumore dell'odometria e dei sensori e compensare alla mancanza di informazioni riguardo la posizione iniziale. Gli algoritmi modellano questo aspetto usando il *calcolo delle probabilità*. I principali sono: il già citato Particle Filter, filtro di Bayes e derivati come Kalman filter.

SLAM SLAM è una tecnica che consente ai robot di eseguire simultaneamente la localizzazione e il mapping di un ambiente sconosciuto. Siccome i problemi di mapping e localization necessitano l'uno dell'altro per poter essere risolti, è necessario risolverli contemporaneamente se non si hanno dati né sull'ambiente né sulla posizione del robot.

Ad alto livello, l'algoritmo segue questi passaggi:

1. *prima scansione*: si usa la prima scansione come mappa iniziale;
2. *cambio della posizione*: il robot si muove di una certa quantità in un piccolo lasso di tempo, si registra quindi una nuova scansione della mappa;
3. *stima della posizione*: la nuova scansione viene correlata con la mappa per stimare il cambiamento di posizione;
4. *aggiornamento della mappa*: si integra la nuova scansione nella costruzione della mappa sovrapponendo le misurazioni. Successivamente il metodo riparte dal punto 2 fino a completamento della mappa.

1.2.2 Planning

La fase di Planning è stata argomento di studio approfondito di questa tesi, perciò verrà descritta completamente nel Capitolo 2 a pag. 10.

1.2.3 Control

La terza fase prevede di attuare le decisioni calcolate nella fase di planning: deve quindi generare, in questo contesto, dei controlli di accelerazione e angolazione delle ruote. In realtà questa fase rientra in un più vasto e più generale campo di studi: l'*ingegneria del controllo* (o dell'automazione).

I principali problemi che bisogna affrontare in questa fase sono:

- come seguire una traiettoria data?
- come compensare gli errori degli attuatori?
- come guidare il più velocemente possibile?

Si fanno distinzione di due tipologie di algoritmi di controllo: gli *open-loop* e *closed-loop*, chiamati anche feedback. In questo contesto, la differenza tra i due è che la seconda tipologia di algoritmi riceve in input, appunto in *feedback*, anche lo stato corrente del veicolo rispetto agli output di controllo generati precedentemente; in questo modo è possibile calcolare l'errore tra il comando calcolato e l'effettivo risultato dell'applicazione di quel comando dagli attuatori, così da poterlo compensare al prossimo comando inviato.

PID Controller L'algoritmo più conosciuto, specialmente per la sua semplicità, è il *PID Controller* [18, 22] che viene usato ampiamente in tutta l'industria dell'ingegneria del controllo, non solo in questo contesto.

PID è un acronimo che sta per **P**roportional-**I**ntegral-**D**erivative e questo suggerisce che l'output è calcolato secondo tre parametri. Si tratta di un algoritmo closed-loop.

Con **proporzionale** si intende che il comando generato deve essere proporzionale all'errore tra lo stato corrente del robot e lo stato desiderato. Siccome ci si vuole che l'errore diminuisca col tempo, con **derivativo** si intende che si applica una correzione all'output che è proporzionale alla velocità con cui l'errore si riduce. Infine, con **integrale** si applica un'ulteriore correzione calcolata sia dalla magnitudine dell'errore sia dalla sua durata nel tempo, ovvero si correggono gli errori accumulati nel tempo che avrebbero dovuto essere corretti prima.

Pure Pursuit Un’altro algoritmo closed-loop più specifico per le corse di robot è Pure Pursuit.[19] In questo caso, il problema richiedere una sequenza di waypoint, ovvero punti nella mappa che formano un *tracciato*, che il robot deve seguire. L’algoritmo tiene conto della dinamica del veicolo, che in questo caso è *non-holonomic*, quindi calcola l’arco che congiunge il robot e il primo waypoint che si trova ad una distanza prefissata, il lookahead, dalla macchina.

Questo è stato il principale algoritmo usato nei laboratori e nello studio della tesi, seppur con qualche modifica migliorativa.

1.3 ROS2

Il lavoro di tesi e l’infrastruttura di F1TENTH sono basati su ROS, un acronimo che sta per **R**obot **O**perating **S**ystem e, sebbene il nome possa trarre in inganno, non è un sistema operativo nel senso tradizionale, bensì si presenta come un mediatore – un **middleware** – tra l’applicazione robot e l’hardware del robot, ma non sostituisce il sistema operativo sottostante. [30, 14]

Viene usato dalla comunità robotica per via della sua alta modularità: dalla ricerca all’insegnamento, da progetti di un gruppo di persone a progetti più importanti di grosse aziende. ROS fornisce i *building blocks* da usare così da facilitare e velocizzare la realizzazione dell’applicativo e fornisce un metodo di comunicazione efficiente tra le varie componenti.

Il progetto è *open-source* ed ha una grossa community di sviluppatori e ricercatori internazionale che lo mantiene e distribuisce pacchetti integrativi per diverse funzionalità come driver per i sensori e attuatori, algoritmi noti, compatibilità con altri programmi, visualizzazioni in tempo reale e tante altre funzioni di utilità. In questo senso, ROS si presenta più simile a un *SDK* (Software Development Kit) accompagnato da una serie di tools, principalmente a linea di comando.

ROS viene rilasciato con la stessa filosofia delle *distro linux*, con più distribuzioni mantenute nello stesso tempo. ROS e ROS2 [4] supportano due linguaggi: C++ e Python. In questa tesi è stato usato Python. Durante lo studio di questa tesi è stato usato ROS2, distribuzione “*Humble*”, per necessità di compatibilità con alcuni pacchetti.

ROS, fondamentalmente, fornisce un metodo di comunicazione tra le componenti dell’applicativo, che vengono chiamati **Nodi**, attraverso diversi sistemi tra cui i **Topic**, i **Servizi** e le **Azioni**. L’insieme dei nodi e dei loro collegamenti formano il *ROS Graph*, il grafo dei nodi.

Di seguito si analizzano le componenti principali di cui ROS è composto.

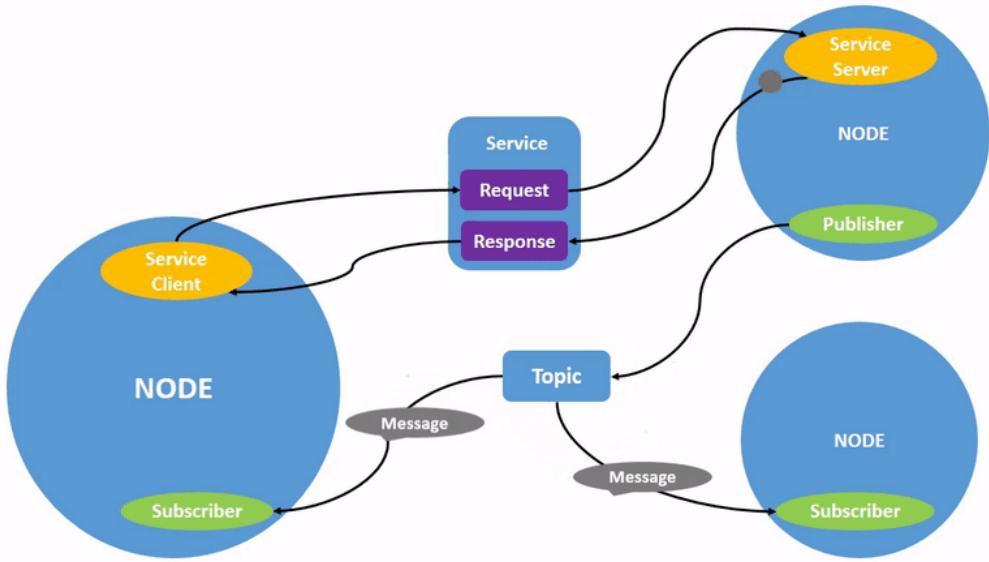


Figura 2: Esempio di ROS Graph [17]

Nodi [32, 26] I nodi, *Nodes* in inglese, sono l’unità computazionale che partecipa al ROS Graph; idealmente, un nodo dovrebbe essere responsabile di un solo compito, logicamente separato da altri altri, per esempio controllare le ruote, processare i dati grezzi derivati dalle misurazioni dei laser o implementare un algoritmo come Pure Pursuit. Un completo sistema robotico comprende, quindi, un insieme di nodi che lavorano congiuntamente, spesso contenuti nello stesso eseguibile.

Topic [33, 29] I topic sono il principale strumento con cui i nodi di un ROS Graph si *scambiano dati*. I topic sono entità **fortemente tipizzate** che implementano un pattern **publisher/subscriber** di tipo *anonimo*, ciò significa che i nodi conoscono una interfaccia comune con cui scambiarsi messaggi ben definiti. *Anonimo* significa che chi invia o riceve dei messaggi da o verso un topic non ha modo di identificare mittente o destinatario perché, in questo contesto, non ha una grossa rilevanza, sebbene esistano comunque dei modi per scoprirlo; inoltre, ogni messaggio generato da un publisher viene ricevuto da ogni subscriber, nell’idea che un nodo si sottoscriva a un topic solo se ne è interessato; questo ha il vantaggio di una grossa flessibilità in termini di manutenibilità.

I nodi possono *pubblicare* e *sottoscriversi* ad un numero arbitrario di topic per

inviare e ricevere dati da altri componenti del grafo: si può avere quindi una relazione uno a uno, uno a molti, molti a uno e molti a molti.

I topic vengono identificati da una stringa univoca che ne rappresenta il nome.

Messaggi [25] I messaggi sono l’*interfaccia*, agnosta al linguaggio, con la quale i nodi comunicano tra di loro. Descrivono un tipo di dato strutturato **fortemente tipizzato** in un file con estensione `.msg`. Questo tipo di file contiene una lista di attributi definiti da tipo dell’attributo, nome ed eventuale valore di default. Il tipo può essere un tipo base (`short`, `double`, `string`) o un tipo composto, ovvero un’altro tipo messaggio.

Parametri e Launch files [24, 27] È possibile parametrizzare i nodi in modo da configurarli sia all’avvio sia durante l’esecuzione senza cambiare codice e ricompilarlo. I parametri consistono in una coppia chiave-valore ed una opzionale descrizione, che può indicare eventuali vincoli di tipo e range di valori.

Spesso i parametri vengono inizializzati all’avvio del nodo attraverso quelli che vengono chiamati *launch file* usati col comando `ros2 launch` e che quindi permettono di avere *diverse configurazioni dello stesso nodo* senza doverlo ricompilare. Un’altro caso d’uso è durante la fase di tuning per alcuni algoritmi: se il contesto lo permette è possibile modificare online i valori dei parametri (per esempio da CLI), a patto che venga gestito nel codice del nodo. I launch file possono essere usati anche per avviare più nodi contemporaneamente, in modo da automatizzare l’intero applicativo.

Services e Actions [28, 23] Altri due sistemi di comunicazioni fra nodi sono i Servizi e le Azioni. Questi sono sostanzialmente chiamate remote – ovvero ad altri nodi – a procedure seguendo il paradigma client/server. La loro principale differenza è concettuale e si discrimina dalla velocità di esecuzione: nei servizi, infatti, la risposta deve essere rapida, mentre nelle azioni ci si aspetta una esecuzione più lunga, anche di minuti, e si ha la possibilità di cancellarla prima del suo termine. Entrambi vengono definiti con strutture simili ai messaggi, con file di diversa estensione.

Per citare qualche esempio, nei servizi troviamo operazioni di trasformazioni di coordinate tra frame, mentre nelle azioni operazioni di più alto livello come “raggiungi questo waypoint”.

La figura 2 rappresenta un ROS Graph comprendente tre nodi, un topic e un servizio; al topic sono sottoscritti i nodi in basso e ricevono i messaggi prodotti dal nodo in alto a destra che fa anche da server al servizio da cui riceve richieste dal nodo client.

1.4 RViz

RViz (Ros Visualizer) è il principale strumento di visualizzazione 3D per gli applicativi ROS. Durante lo studio di questa tesi è stato usato per visualizzare la simulazione di F1TENTH gym, il modello del robot, la mappa del circuito ed eventuali visualizzazioni dei vari algoritmi in studio, per esempio i waypoint di goal per Pure Pursuit, l'albero dei punti random generati da RRT e i waypoint che compongono la raceline ottima. La figura 3 ne mostra un'esempio.

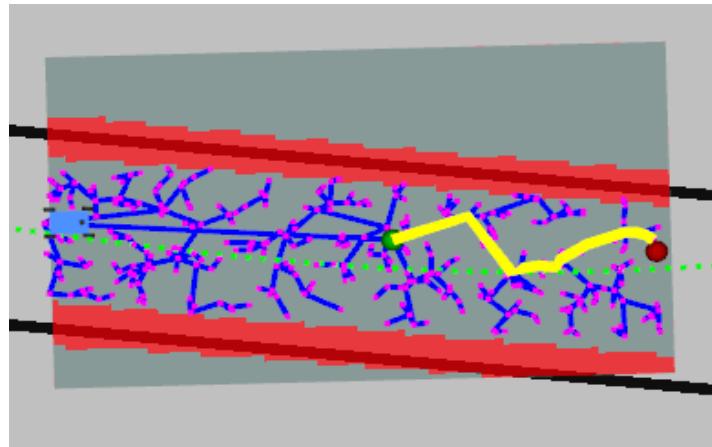


Figura 3: Esempio di caso d'uso di Rviz per visualizzare l'algoritmo di local planning RRT e l'algoritmo di controllo Pure Pursuit; in rosso gli ostacoli rilevati dal sensore ottico, in verde chiaro i waypoint che compongono il percorso globale da seguire, l'albero di RRT è visualizzato con archi blu e nodi magenta, il punto verde è il waypoint di Pure Pursuit ad una certa distanza L e il punto rosso il goal waypoint di PurePursuit, in giallo il percorso sull'albero seguito per ricercare il punto verde.

Capitolo 2

Planning

In questo capitolo si andranno ad analizzare i compiti della seconda fase all'interno del ciclo della pipeline dei veicoli autonomi, ovvero il planning. In questa fase viene inserito l'argomento di studio di questa tesi: *l'ottimizzazione della traiettoria globale*. Il planning, come suggerisce il nome, ha il compito di pianificare le mosse successive del robot affinché possa spostarsi in modo ottimale evitando eventuali ostacoli.

Gli algoritmi si differenziano per tecniche e obiettivi, in particolare si distinguono tre macro-categorie che rispondono a domande diverse:

- **Mission/Global Planner:** *Qual è l'obiettivo generale del veicolo?* Per esempio trovare il percorso più breve, o quello più corto;
- **Behavioural Planner:** *Come dovrebbe comportarsi il veicolo in diverse situazioni?* Per esempio, in una gara con più concorrenti, quando superare un avversario, come e dove;
- **Local Planner:** *Quali sono le traiettorie possibili dalla posizione attuale al goal?* Per esempio trovare la miglior traiettoria tale che rientri nella possibilità fisica del veicolo.

L'oggetto di questa tesi, l'ottimizzazione della traiettoria di corsa, rientra nella prima categoria.

Workspace vs Configuration Space

Si distinguono due rappresentazioni degli ostacoli e dell'ambiente: il workspace rappresenta tutte le *azioni possibili* a un robot, si pensi al volume occupato da tutti i possibili movimenti di un braccio meccanico, mentre il configuration space, detto anche C-space, rappresenta tutte le possibili configurazioni di una data *definizione*

di stato. In questo contesto, all'atto pratico, ciò si traduce che nel secondo, la rappresentazione dell'ambiente e degli ostacoli tiene già conto della *dimensione* del robot e quindi può essere considerato come un singolo punto, mentre ciò non accade nel primo. La figura 4 ne mostra un esempio.

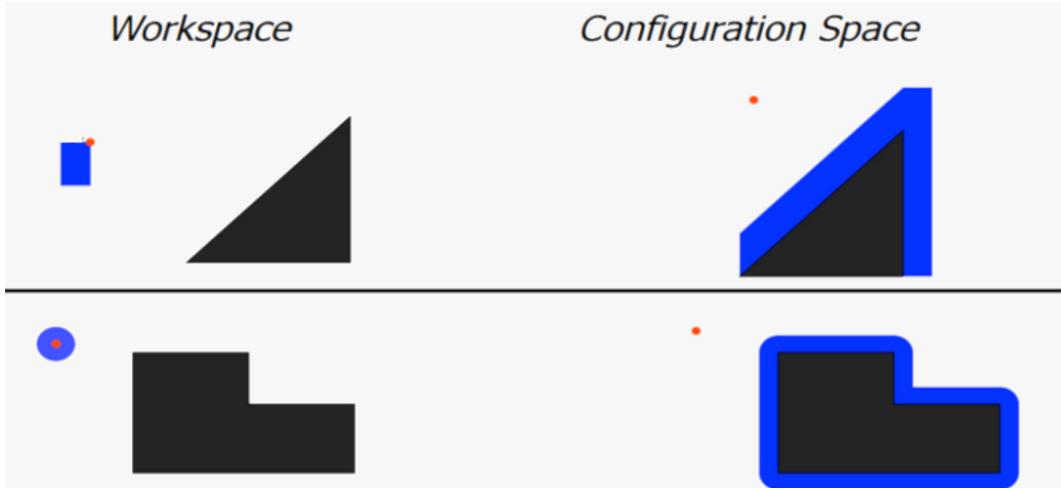


Figura 4: Differenze tra Workspace e C-Space [21]; in nero la dimensione degli ostacoli reali, in rosso il punto di controllo del robot, in blu, nel caso workspace, lo spazio occupato dal robot, mentre per C-Space, lo spazio che occuperebbe vicino agli ostacoli

Nel contesto delle corse si preferisce l'uso del C-space per via della sua *espansività* nella descrizione dello stato, che può non solo descrivere un piano o uno spazio tridimensionale, ma è possibile esprimere ulteriori variabili come la velocità e l'orientamento: è così possibile esprimere *ulteriori vincoli*.

Di seguito si analizzano le tre macro-categorie sopra citate.

2.1 Local Planner

L'obiettivo principale del local planner è quello pianificare i movimenti del robot fino ad un dato orizzonte finito evitando collisioni con l'ambiente ed eventuali avversari. Gli algoritmi si differenziano per tre categorie principali di risoluzione:

1. Applicando *modifiche* la raceline globale;
2. *Generando diverse traiettorie* vincolate alla dinamica del robot e scegliendo quella migliore;

3. *Campionando* lo spazio libero e trovando un percorso attorno agli ostacoli.

Nella prima categoria rientrano, generalmente, algoritmi che si basano su MPC che, sebbene sia un algoritmo di controllo ottimale, può essere adattato e utilizzato per ricercare il percorso ottimo per evitare un ostacolo o per migliorare la traiettoria della raceline globale di riferimento in base alla posizione attuale del veicolo.

Nella seconda categoria ricadono algoritmi che calcolano, fino a un dato orizzonte temporale, lo stato successivo del veicolo per diversi input di accelerazione e angolo delle ruote; questa operazione produce diverse traiettorie, dinamicamente praticabili per il robot, da cui si sceglie la migliore secondo una funzione di costo. Altri algoritmi, come State Lattice, generano punti equidistanti nell'ambiente tra di loro connessi da delle spline che seguono la dinamica del veicolo, poi viene ricercato il percorso migliore. Un esempio grafico è mostrato in figura 5.

La terza categoria, data una occupancy grid e un punto di goal, algoritmi come PRM (Probabilistic Road Map) e derivati come RRT (Rapidly-Exploring Random Tree) campionano lo spazio libero e generano un grafo che connette i punti campionati per poi cercare il percorso che avvicina il robot al goal. RRT è stato un caso di studio durante questa tesi durante un laboratorio del Modulo D. Un esempio di RRT si può trovare alla figura 3. Altre soluzioni usano i classici algoritmi di ricerca su grafo, come A* e Dijkstra, sulla stessa occupancy grid; il grafo costruito a partire da essa esprime sui nodi le posizioni nell'ambiente e gli archi i possibili input da quelle posizioni. Queste ultime sono soluzioni che lavorano nel discreto, molto semplici da implementare ma che non hanno la stessa espressività dei metodi continui e che non hanno la possibilità di descrivere la dinamica del robot.

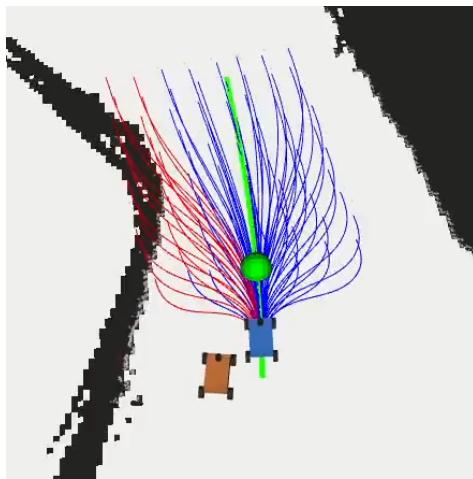


Figura 5: Esempio grafico dell'algoritmo State Lattice [8]

2.2 Behavioural Planner

Il focus di questo planner è generalmente selezionare un peso appropriato a diversi obiettivi o combinare il risultato del local planner con metodi della teoria del gioco così da pianificare delle mosse atte a impedire il progresso degli avversari.

Nel primo caso, gli obiettivi rappresentano valori come il progresso sul circuito, la vicinanza con gli ostacoli e contendenti, la deviazione dal percorso ottimale, la velocità massima; il costo totale di una traiettoria viene quindi calcolato combinando secondo vari pesi gli obiettivi presi in considerazione, viene quindi scelto la traiettoria col costo minore.

Nel secondo caso vengono sfruttate metodologie della teoria dei giochi per trovare l'azione migliore in un ambiente con due o più giocatori. Il problema viene trasformato in gioco competitivo asincrono dove un singolo giocatore può "muoversi" per volta. Questi approcci spesso includono nel calcolo della soluzione il concetto di *regret* per trovare la migliore risposta per vincere la gara.

2.3 Global Planner

Come citato più volte precedentemente, questa tipologia di planner è stato oggetto di studio di questa tesi e quindi ha dedicato un capitolo, il numero 3 (pag. 14).

Il global planner ha una visione d'insieme del circuito ed è agnostico alla singola gara, perciò i suoi obiettivi sono legati a proprietà della traiettoria (globale) da seguire; generalmente l'obiettivo principale è eseguire il tracciato in meno tempo possibile, ma esistono altre possibilità come il minor consumo d'energia e traiettorie con particolari proprietà geometriche, come la minor curvatura.

Capitolo 3

Raceline Optimization

Lo studio di questa tesi si è concentrato a trovare una soluzione al seguente problema: *data una mappa, trovare la sua raceline globale ottima, secondo un criterio scelto*. Il criterio di ottimizzazione principale, in questi casi, è il tempo di percorrenza del tracciato, ovvero il *lap time*.

Le porzioni interessanti di un circuito da gara sono le *curve*: ne esistono di diversi tipi e altrettanti modi per affrontarle in base al tipo; trovare una percorso ottimo, in questo senso, è un compito complesso e il percorso più breve non coincide necessariamente con quello che richiede meno tempo, a causa delle curve che riducono la velocità media. Una possibile strategia per mantenere alta la velocità in curva è quella di percorrere la traiettoria con la curvatura minore. Questo approccio, tuttavia, non prende in considerazione una sequenza di curve e la velocità di uscita da una curva. Il percorso ottimo in termini di tempo, quindi, è un compromesso tra queste due strategie, ovvero quella che minimizza la distanza percorsa mantenendo alta la velocità nelle curve ed eventualmente anche in uscita da esse.

Trovare la raceline ottima è generalmente un'operazione preliminare alla gara, dunque *non può considerare* eventuali avversari o ostacoli dinamici; per questo motivo viene calcolata nell'assunzione di un *singolo robot* sul tracciato. Tuttavia non è utile solo nel contesto considerato: la raceline globale, avendo una visione intera sulla mappa, può essere sfruttata da un local o behavioural planner come linea guida.

3.1 Le curve

Le principali complessità di un circuito di cui l'ottimizzazione si occupa è come affrontare le curve.

Caratteristica principale di una curva è il suo *raggio*: un raggio maggiore corrisponde a velocità più alte e viceversa; inoltre, si distinguono il raggio esterno e quello interno e la loro differenza risulta nell'ampiezza del circuito in quel punto. Esistono anche caratteristiche di natura tridimensionale, come la variazione dell'altezza e l'inclinazione, tuttavia non sono state oggetto di studio in questa tesi dal fatto che il contesto di F1TENTH non include questa variabilità, come invece accade nelle gare di F1.

In linea generale, una curva può essere suddivisa in *quattro sequenze* principali, come mostrato in figura 6:

1. approccio e decelerazione;
2. inizio della curva;
3. raggiunta dell'apice;
4. uscita della curva e accelerazione.

Anche la traiettoria, dato che segue una curva, può essere definita con un raggio che può essere *costante* o *variabile* dall'inizio della curva fino alla fine.

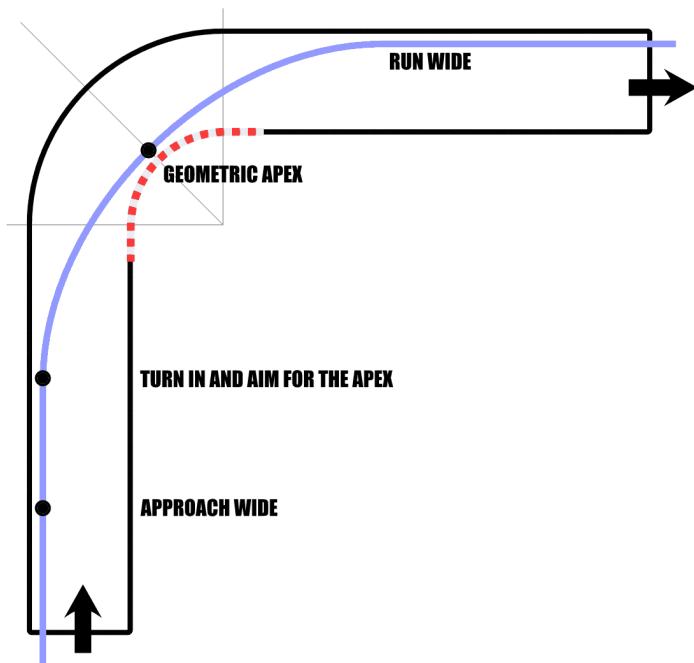


Figura 6: Rappresentazione delle fasi di una raceline geometrica per una curva a destra [5]

Raceline geometrica L’approccio più semplice e immediato per affrontare una curva genera una raceline che prende il nome di *raceline geometrica*. La strategia prevede queste sequenze, rappresentate in figura 6:

1. approccio largo e verso l’esterno del circuito;
2. sterzata verso il centro del raggio interno della curva;
3. uscita larga e verso l’esterno del circuito.

Si tratta, quindi, di una raceline a raggio *costante*.

I principali vantaggi di questo approccio rispetto alla raceline ottima sono la sua semplicità, sia in termini di modellazione sia in termini di computazione, e la possibilità di mantenere un’alta velocità durante la curva. Tuttavia, la raceline geometrica ha importanti svantaggi quali una decelerazione prematura all’entrata della curva, una accelerazione ritardata all’uscita e la possibilità di ritrovarsi in uno stato svantaggiato per affrontare la prossima curva, nel caso di curve in sequenza.

Dunque, la raceline ottimale non dovrebbe tener conto solamente della singola curva ma eventualmente anche di quelle successive: per estensione, quindi, deve essere considerato tutto il circuito.

Raceline ottimale Al contrario della raceline geometrica, non è possibile definire a priori le sequenze da seguire come fatto in precedenza, proprio perché dipende dalle caratteristiche della curva. Riprendendo come esempio la curva in figura 6, la raceline ottimale – ovvero quella che riesce ad eseguire la curva nel minor tempo possibile – tarda il più possibile l’inizio della curva per mantenere più velocità all’entrata, decelera più velocemente e ha una virata più stretta e con un apice più distante da quello geometrico, così facendo si ha la possibilità di aver più tempo per accelerare di nuovo all’uscita della curva con la possibilità di mantenere una traiettoria più centrata rispetto al circuito.

In figura 7 vi è una comparazione grafica tra le due raceline.

La raceline ottima, inoltre, adegua la traiettoria di uscita in base alla curva successiva, se presente, in modo tale da affrontarla anch’essa nel modo ottimale. Si nota la distinzione in figura 8.

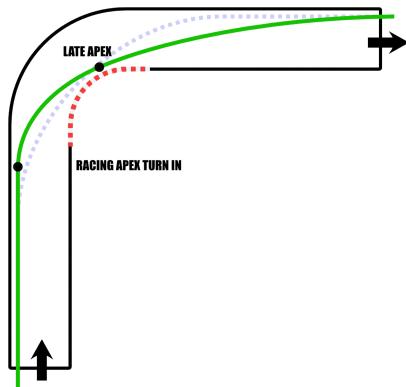


Figura 7: Comparazione tra race-line ottimale, in verde, e race-line geometrica, in blu tratteggiato [5]

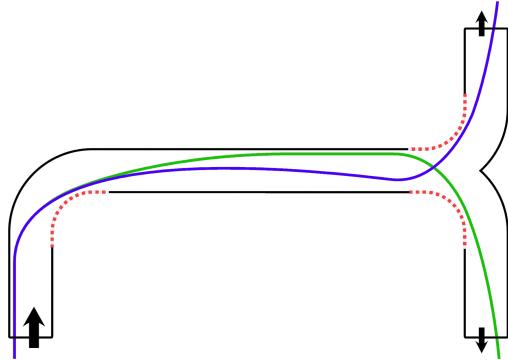


Figura 8: Confronto tra race-line ottime con due curve distinte in sequenza [5]

3.2 Descrizione del problema

Di seguito si presentano alcune definizioni più formali dei termini utilizzati precedentemente e il processo di generazione del percorso ottimo secondo un certo criterio.

In questa tesi sono state implementate e analizzate tre strategie differenti:

- Percorso più breve – *shortest path*;
- Curvatura minima – *minimum curvature*;
- Tempo minimo – *minimum time*.

Il problema in questione è un problema di ricerca su due variabili: il percorso effettivo sulla mappa e la velocità da mantenere per ogni punto del percorso. Queste due variabili assieme definiscono la *traiettoria* o *raceline*, in inglese.

I vincoli applicati alla ricerca sono due: i limiti dinamici del veicolo, per esempio quanta forza G il veicolo riesce a sostenere in sterzata, e gli ostacoli statici, rappresentati dai bordi del circuito.

Il problema viene suddiviso in due sotto-problemi, ovvero risolvere singolarmente la ricerca sulle due variabili in gioco. Si distinguono quindi, in sequenza:

1. Generazione del percorso secondo i vincoli desiderati – *path planning*
2. Calcolo della velocità sul percorso generato – *velocity planning*

In particolare, il path planning, almeno per quanto riguarda le prime due strategie, viene modellato come un problema di programmazione quadratica geometrica (Geometric QP), che ha i vantaggi di richiedere pochi parametri e di essere computazionalmente veloce; esistono anche altri tipi di algoritmi di ottimizzazione basati su evoluzione, come CMA-ES. Il velocity planning può essere risolto con algoritmi a stato quasi-stazionario come l'algoritmo forward-backward. Un ulteriore metodo per il velocity planning per il tempo più breve è descritto in *"Minimum-time speed optimization over a fixed path"* [13].

Per quanto riguarda la terza strategia, i due sotto-problemi devono essere risolti contemporaneamente usando algoritmi di controllo ottimale, questi hanno la possibilità di includere ulteriori vincoli, come per esempio limitazioni al consumo di energia o considerare l'attrito delle ruote per diversi terreni e condizioni meteo.[3] Sono algoritmi con modelli dinamici dell'auto più completi e fedeli alla realtà che raccolgono nel problema diverse variabili, per questo sono più complessi e richiedono più sforzo computazionale, ma hanno il vantaggio di essere più realistici nei risultati rispetto alle loro controparti modellate in problemi di programmazione quadratica.

Definizione di spline [15] [31] Un percorso, per poter essere computato da una macchina, deve necessariamente essere discretizzato in punti, detti *samples*. Tuttavia è necessaria una descrizione matematica del tracciato: questa viene formulata con l'uso delle *spline*. Una spline è una funzione definita a pezzi da un'insieme di polinomi dello stesso grado il cui scopo, in questo contesto, è quello di interpolare dei punti, ovvero i samples, in modo tale che la funzione sia continua fino ad un dato ordine.

Una spline, per poter rappresentare un percorso che deve essere seguito da un'auto, è necessario che rispetti alcuni vincoli, ovvero:

1. L'inizio di un polinomio deve essere un punto di discretizzazione, un sample;
2. La fine di un polinomio deve essere il sample successivo all'inizio;
3. L'orientamento alla fine di una funzione deve essere uguale all'orientamento della funzione successiva;
4. La curvatura alla fine di una funzione deve essere uguale alla curvatura della funzione successiva;

I primi due vincoli sono garantiti dal fatto che la spline interpola i sample, per gli ultimi due è necessario calcolare la derivata prima e seconda del polinomio in quei punti. I polinomi di grado tre offrono, quindi, una buona rappresentazione del percorso perché garantiscono la calcolabilità delle derivate prime e seconde per ogni punto del dominio.

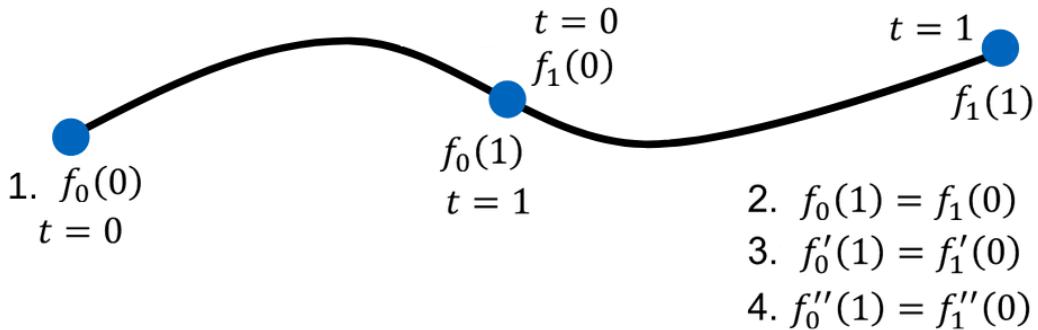


Figura 9: Esempio di spline formata da due polinomi e vincoli associati [20]

Il dominio di ogni polinomio viene normalizzato per la distanza tra il punto di partenza e quello di fine dal parametro t , dove $t = 0$ indica il punto di inizio e $t = 1$ il punto di fine. Sia $f_i(t) = a_i + b_i t + c_i t^3 + d_i t^3$ il polinomio che ha come inizio il sample all'indice i , l'immagine 9 mostra graficamente una spline e i vincoli associati.

Il calcolo dei valori delle x e delle y avviene separatamente ma usando gli stessi metodi, di seguito, quindi, verranno descritti solo la formulazione per x . Matematicamente i vincoli sopra descritti vengono formulati come:

1. $t = 0 \quad a_i = x_i$
2. $t = 1 \quad a_i + b_i + c_i + d_i = x_{i+1}$
3. $t = 1 \text{ e } t = 0 \quad b_i + 2c_i + 3d_i - b_{i+1} = 0$
4. $t = 1 \text{ e } t = 0 \quad 2c_i + 6d_i - 2c_{i+1} = 0$

dove per il 3. e 4. punto vengono rispettivamente ricavati dalla differenza $f'_i(1) - f'_{i+1}(0)$ e $f''_i(1) - f''_{i+1}(0)$

Definizione di centerline La centerline, o reference line, è il percorso che è equidistante dai bordi del circuito. Viene discretizzato in punti di egual distanza tra loro e si registrano le distanze ai bordi destro e sinistro, così da formare una linea immaginaria, ortogonale alla reference line, che unisce i due punti. L'immagine 10 ne mostra un esempio grafico.

La linea centrale di riferimento è necessaria agli algoritmi di ottimizzazione come punto di partenza per trovare la soluzione ottima. La distanza dei samples è un parametro dell'algoritmo e deve essere decisa in base al caso d'uso, come approfondito al paragrafo 4.2.1.

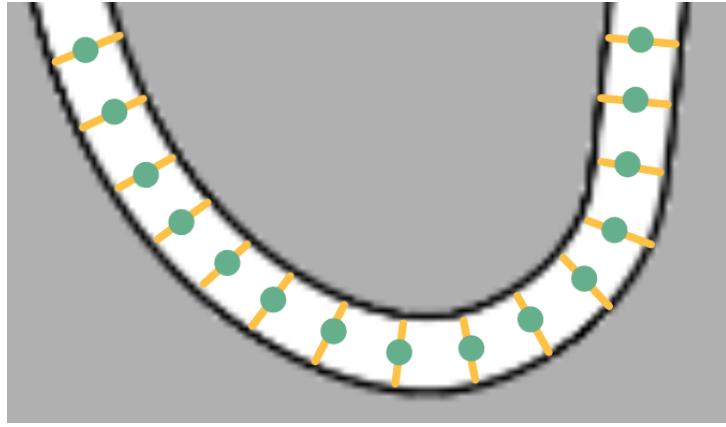


Figura 10: La figura mostra la rappresentazione discreta della centerline (i punti verdi) e le linee immaginarie (quelle gialle) che uniscono, ortogonalmente rispetto alla centerline, i bordi del circuito

Definizione di raceline I samples della raceline vengono calcolati a partire da quelli delle centerline "muovendoli" sulla linea immaginaria che unisce i due bordi. Formalmente:

$$\vec{r}_i = \vec{p}_i + \alpha_i \vec{n}_i$$

dove il sample i della raceline r , descritto dal vettore \vec{r}_i , viene traslato di un fattore a_i lungo il vettore \vec{n}_i , ovvero la linea che unisce i bordi del circuito. La figura 11 rappresenta graficamente questo processo.

È necessario, inoltre, imporre un vincolo sul fattore a in modo tale che non venga prodotto un sample al di fuori del circuito o che non permetta al robot di passare perché troppo vicino ai bordi. Formalmente:

$$a_i \in [-w_{tr_left,i} + \frac{w_{veh}}{2}, w_{tr_right,i} - \frac{w_{veh}}{2}] \quad (1)$$

dunque, dato un indice i , il fattore a deve rientrare nel range definito tra la distanza del circuito a sinistra w_{tr_left} del sample della centerline dello stesso indice e la distanza a destra w_{tr_right} aggiustato per la larghezza del veicolo w_{veh} .

3.3 Path planning

Ora verranno esaminate più in dettaglio il modello dei problemi di QP geometrico per le strategie di percorso più breve e minor curvatura.

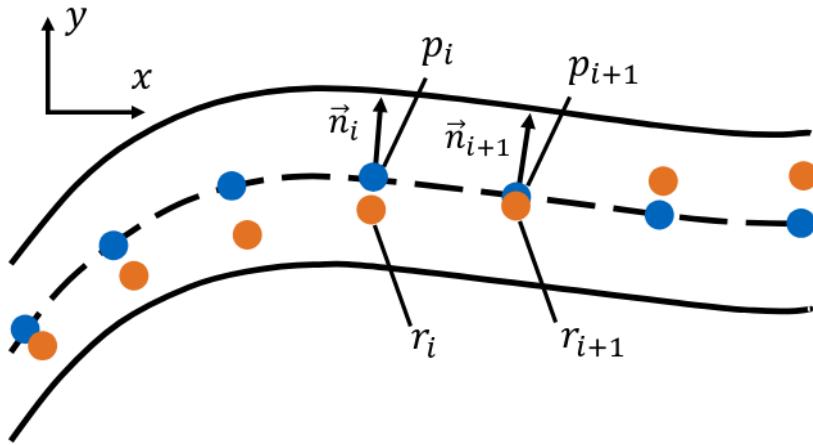


Figura 11: Rappresentazione grafica della raceline, dove i punti blu rappresentano i samples della centerline, mentre quelli arancioni sono quelli della raceline, traslati di un certo fattore lungo i vettori n corrispondenti. [20]

La programmazione quadratica è un processo che risolve problemi di ottimizzazione matematica descritti da funzioni quadratiche. L'obiettivo è quello di trovare un vettore che minimizzi o massimizzi il valore di una funzione obiettivo preservando contemporaneamente dei vincoli, descritti da disequazioni.

Percorso più breve [2][31] Il modello in questione deve descrivere il percorso più breve, ovvero che abbia la distanza minore tra i samples. Un sample della raceline viene formulato come:

$$\vec{P}_i = \begin{bmatrix} x_{r,i} + a_i(x_{l,i} - x_{r,i}) \\ y_{r,i} + a_i(y_{l,i} - y_{r,i}) \end{bmatrix} = \begin{bmatrix} x_{r,i} + a_i\Delta_{x,i} \\ y_{r,i} + a_i\Delta_{y,i} \end{bmatrix}$$

ovvero, il punto più a destra del segmento i -esimo ortogonale alla centerline spostato sullo stesso per un fattore a . Nella figura 12 ne viene mostrato un esempio. La distanza tra un sample e il suo successivo può essere descritta, quindi, come:

$$\Delta P_{x,i} = \vec{P}_{i+1} - \vec{P}_i = \begin{bmatrix} x_{r,i+1} - x_{r,i} + a_{i+1}\Delta_{x,i+1} - a_i\Delta_{x,i} \\ y_{r,i+1} - y_{r,i} + a_{i+1}\Delta_{y,i+1} - a_i\Delta_{y,i} \end{bmatrix}$$

Le variabili da trovare è il vettore di a da applicare ai samples, dunque:

$$\begin{aligned} \min [a_i \dots a_N] & \sum_{i=1}^N (\Delta P_{x,i})^2 \\ \text{subj. to } a_i & \in [a_{i,\min}, a_{i,\max}] \quad \forall 1 \leq i \leq N \end{aligned}$$

dove il vincolo per a_i è lo stesso descritto nella formula 1.

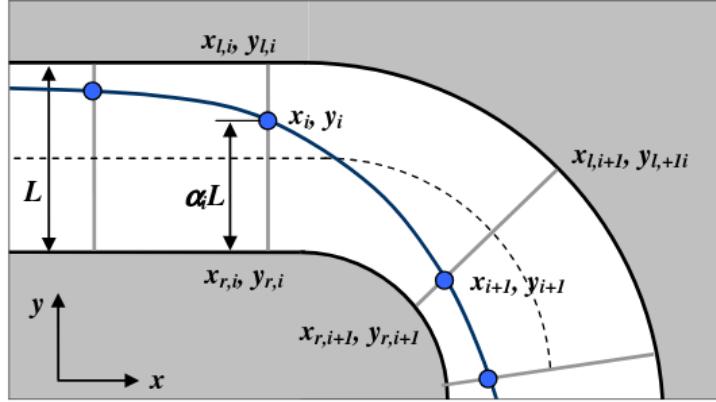


Figura 12: Rappresentazione grafica della descrizione matematica di un sample. [2]

Minima Curvatura La formulazione del modello è identica a differenza, ovviamente, della funzione obiettivo che deve descrivere la curvatura della raceline, il modello, quindi, viene definito come:

$$\begin{aligned} \min [a_i \dots a_N] & \sum_{i=1}^N \kappa_i^2 \\ \text{subj. to } a_i & \in [a_{i,\min}, a_{i,\max}] \quad \forall 1 \leq i \leq N \end{aligned}$$

dove κ_i è la curvatura della funzione in un sample i , definita come:

$$\kappa_i = \frac{x'_i y''_i - y'_i x''_i}{(x'^2_i + y'^2_i)^{\frac{3}{2}}}$$

che elevata al quadrato:

$$\kappa_i^2 = \frac{x'^2_i y''^2_i - 2x'_i x''_i y'_i y''_i + y'^2_i x''^2_i}{(x'^2_i + y'^2_i)^3}$$

3.4 Velocity planning

Come citato precedentemente, il calcolo della velocità per ogni sample può essere calcolato con l'algoritmo *forward-backward*, che ha il vantaggio di essere semplice, veloce e sufficientemente accurato.

L'algoritmo assume che il robot venga guidato ai suoi limiti fisici per ogni punto del percorso, quindi si ha che l'accelerazione longitudinale dipende dai limiti del motore e dall'azione frenante dei freni, mentre l'accelerazione laterale è limitata dalle forze massime trasmissibili degli pneumatici.

Ad alto livello, l'algoritmo si può riassumere in:

1. Si genera una prima stima della velocità basandosi sulla massima accelerazione laterale per la curvatura di ogni sample;
2. Si riducono le stime alla velocità massima consentita per quei sample che la superano;
3. **Forward:** si itera sui samples e, se non si sono superati i limiti fisici del robot, si applica una possibile accelerazione longitudinale, altrimenti
4. **Backward:** si itera sui samples e si applica una possibile decelerazione longitudinale.

In particolare per gli ultimi due punti la velocità al punto successivo viene calcolata come segue:

1. Si calcola l'accelerazione laterale data la curvatura per quel sample:

$$a_{y,i} = v_i^2 \cdot \kappa_i$$

2. Si determina il potenziale rimanente per l'accelerazione longitudinale:

$$a_{x,i} = a_{x,max} \sqrt{1 - \left(\frac{a_{y,i}}{a_{y,max}} \right)^2}$$

3. Si può calcolare quindi la velocità al punto successivo assumendo un'accelerazione costante per la distanza l_i tra un sample e quello successivo:

$$v_{i+1} = \sqrt{v_i^2 + 2 \cdot a_{x,i} \cdot l_i}$$

Capitolo 4

Implementazione

In questo capitolo verranno descritti i passaggi implementati per generare la race-line ottima data una mappa.

Nel corso dello studio di questa tesi, è stato integrato un fork¹ della repository di [3], utilizzandolo come base di partenza. In questa repository sono implementati gli algoritmi analizzati al capitolo 3 (pag. 14); quindi si è condotta un’analisi del codice sorgente, identificando le parti rilevanti per lo scopo e apportando delle modifiche e personalizzazioni per adattarlo al contesto, come per esempio selezionare i tipi di output da produrre. Questo ha permesso di ottimizzare le funzionalità esistenti e di implementare ulteriori miglioramenti, garantendo così che il codice rispondesse alle necessità dello studio.

I passaggi ad alto livello sono i seguenti:

1. ottenere l’immagine di una mappa (per esempio attraverso SLAM);
2. eventualmente ripulirla di imperfezioni e ottenere un circuito chiuso;
3. generare la centerline e calcolare la larghezza del circuito;
4. applicare gli algoritmi di generazione della raceline;

La mappa viene digitalizzata in file immagine che rappresenta una *occupancy grid*, mentre i percorsi, che siano raceline o centerline, sono in formato csv con due colonne rappresentanti la posizione (x, y) di ogni sample, nel caso sia una centerline, e una terza colonna per la velocità per quanto riguarda la raceline ed eventualmente ulteriori colonne per altri valori come l’accelerazione e l’angolo di sterzata.

Una occupancy grid è una immagine in scala di grigi in cui, per ogni pixel, viene rappresentata la *probabilità* che quel pixel sia uno spazio libero attraverso una percentuale di grigio. Dunque in occupancy grid binarie, il nero indica l’ostacolo certo, mentre il bianco lo spazio libero.

¹<https://github.com/CL2-UWaterloo/Raceline-Optimization>

Di seguito si indicano le specifiche della macchina usata durante questa tesi:

- *OS*: Ubuntu 22.04.4 LTS Jammy Jellyfish (kernel 6.8.0-45-generic)
- *Kernel*: 6.8.0-40 generic x86_64
- *CPU*: Intel Core i7-6700k quad core
- *RAM*: 16GB

4.1 Generazione centerline

Affinché l'algoritmo per l'estrazione della linea di riferimento restituisca un risultato corretto l'immagine della mappa deve rappresentare un circuito chiuso e con bordi ben definiti e lisci, come effettivamente sarebbe un circuito di F1. Dunque se si acquisisce una mappa con SLAM come in figura 13 è necessario modificarla come in figura 14. Durante l'implementazione di questa tesi si è deciso di usare le mappe dei circuiti da gara più famosi forniti da F1TENTH stesso [9].

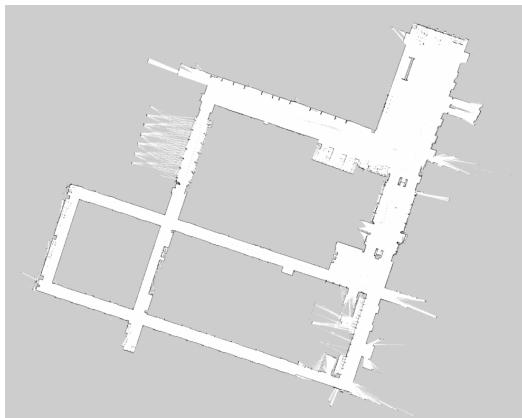


Figura 13: Mappa risultante da SLAM [12]

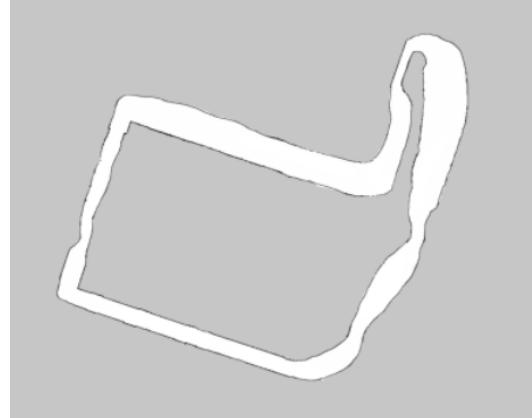


Figura 14: Modifica dell'immagine 13 per creare un circuito chiuso [12]

Operativamente, l'algoritmo usato per la generazione viene dal mondo dell'elaborazione delle immagini e si chiama EDT (Euclidian Distance Transform), che calcola la distanza euclidea per ogni pixel dell'immagine dal background: in questo caso, lo sfondo preso in considerazione sono i punti non esplorati, ovvero quelli esterni ai bordi del circuito. Un esempio di applicazione di EDT si trova all'immagine 15.

Un aspetto da sottolineare è che le immagini fornite da F1TENTH rappresentano una occupancy grid ternaria, in cui il grigio corrisponde alle zone non esplose; per sfruttare l'algoritmo sopra citato è necessario che i pixel grigi vengano convertiti in nero, perché è il valore considerato come background dall'algoritmo EDT. Prendendo come riferimento il circuito di Monza all'immagine 16, il risultato dell'algoritmo si può vedere all'immagine 17.

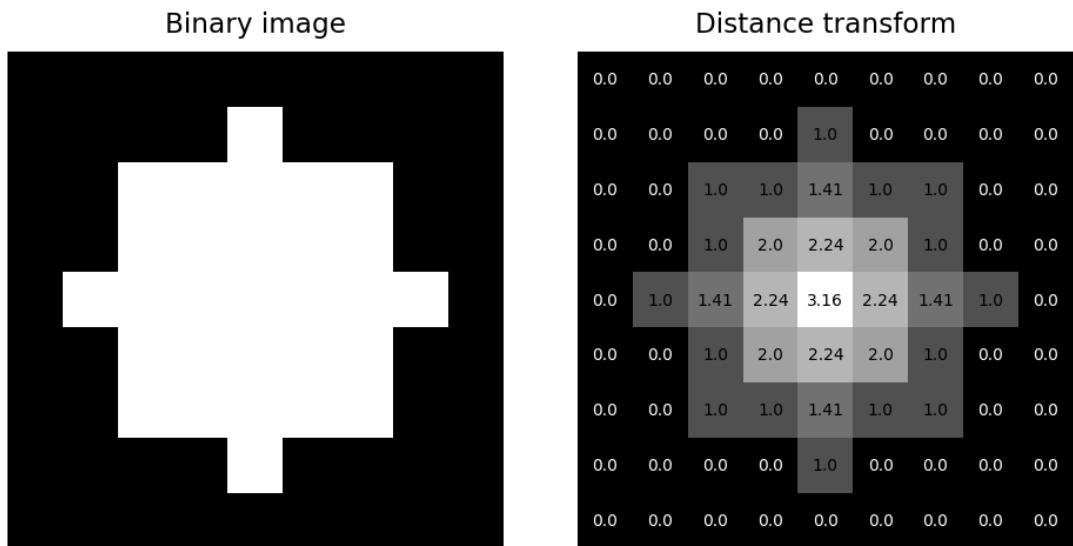


Figura 15: Esempio di applicazione dell'algoritmo EDT su una immagine binaria, l'immagine a destra mostra il risultato indicando la distanza euclidea dal background per ogni pixel [11]

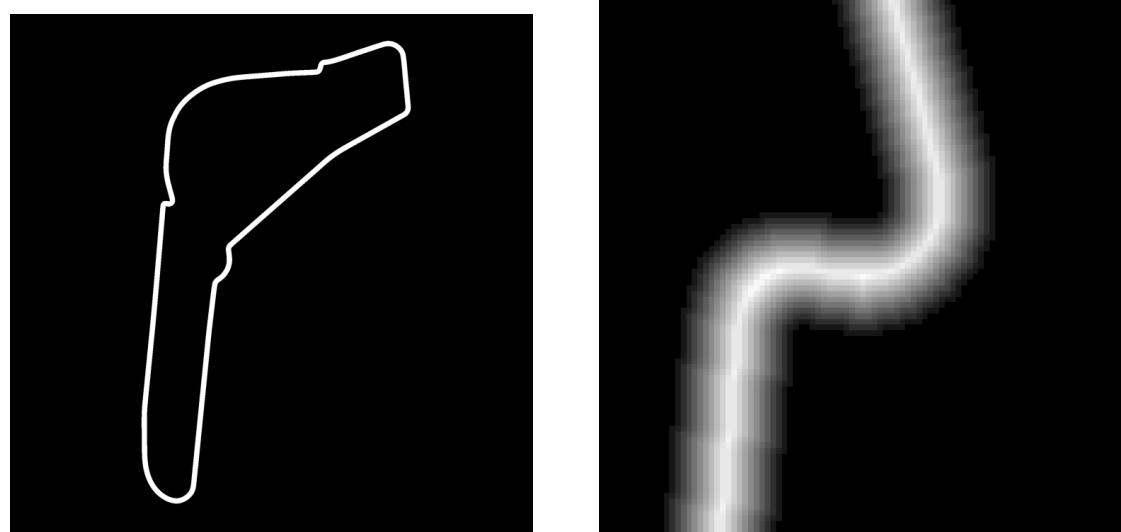


Figura 16: Immagine della occupancy grid binaria del circuito di Monza

Figura 17: Risultato dell'algoritmo EDT mostrato sulla curva 1-2 del circuito di Monza

Il passo successivo è quello di ottenere solo la parte centrale, ovvero quella con i valori di bianco più alti e ridurla a un singolo pixel. In Python, questa operazione, viene eseguita dalla funzione `skeletonize()` del pacchetto `scikit-image`. Seguendo l'esempio con il circuito di Monza, il risultato di questa fase si può osservare all'immagine 18.

A questo punto è necessario campionare il percorso di un pixel trovato: partendo da un punto del percorso, si applica una ricerca DFS (Depth First Search) per trovare i successivi pixel bianchi. Dunque da un pixel del percorso, si cerca il primo pixel bianco in tutte le direzioni che non sia già stato esplorato: seguendo questo processo si ottengono le posizioni dei pixel e la loro distanza dai bordi, si trasformano nel frame della mappa e esportano queste informazioni in un csv contenente le colonne `x`, `y`, `width_left`, `width_right`.



Figura 18: Visualizzazione della centerline di Monza, il punto verde indica il punto di partenza

4.2 Esecuzione dell'ottimizzazione

Ad alto livello, l'ottimizzazione cerca iterativamente di aggiustare la centerline in modo tale da ottenere il risultato migliore, come spiegato nel capitolo 3 pag. 14. Questa fase non richiede solamente di eseguire il programma solutore, ma è necessario configurare i parametri che l'algoritmo usa. Questo compito si chiama *tuning* dei parametri e consiste nel trovare quelli che meglio si adattano nel trovare la soluzione migliore.

Durante lo studio di questa tesi il principale parametro modificato durante le prove è stato lo *stepsize*, ovvero la distanza, in metri, tra un sample e l'altro. In particolare si distinguono tre stepsize diversi:

- **stepsize_prep**: viene usato per l'interpolazione lineare prima dell'approssimazione del percorso con la spline;
- **stepsize_reg**: usato durante l'ottimizzazione per l'interpolazione della spline;
- **stepsize_interp_after_opt**: usato per campionare la spline dopo l'ottimizzazione, è la distanza dei sample esportati nel csv risultante.

Altri parametri fanno riferimento alla dinamica del veicolo, come angolo di sterzata massimo e velocità massima, e grandezze fisiche del veicolo, come massa, lunghezza e larghezza. Sono disponibili anche parametri specifici per il tipo di ottimizzazione, per esempio un valore aggiuntivo alla larghezza del veicolo che comprende un margine di sicurezza dai bordi del circuito. Di seguito si mostrano alcuni dei valori usati che descrivono le proprietà del veicolo indipendentemente dal tipo di ottimizzazione usato:

```
veh_params = {
    "v_max": 15.0,          # [m/s]
    "length": 0.58,         # [m]
    "width": 0.31,          # [m]
    "mass": 3.74,           # [Kg]
    "dragcoeff": 0.075,     # [kg*m^2/m^3]
    "curvlim": 3.0,         # [rad/m]
    "g": 9.81               # [N/Kg]
}
```

Il risultato di questa fase è un file csv contenenti i samples della raceline generata. Di seguito si mostra un esempio ² dei primi 6 samples per il file mintime-28-09-2024_18-12.csv per il circuito di Spa:

```
# s_m: distanza in metri della traiettoria
# x_m, y_m: posizione del sample nel frame della mappa
# psi_rad: orientamento del veicolo rispetto alla mappa
# kappa_radpm: curvatura per quel punto in rad/m
# vx_mps: velocita' in m/s
# ax_mps2: accelerazione in m/s^2

# s_m; x_m; y_m; psi_rad; kappa_radpm; vx_mps; ax_mps2
0.000,11.148,-65.899,-0.370,0.032,10.236,-1.072
0.149,11.202,-65.759,-0.365,0.032,10.220,-1.070
0.299,11.256,-65.618,-0.360,0.032,10.204,-1.068
0.449,11.308,-65.477,-0.355,0.032,10.189,-1.067
0.599,11.360,-65.336,-0.350,0.032,10.173,-1.065
0.749,11.412,-65.194,-0.345,0.032,10.157,-1.064
```

A fini di una migliore gestione, si è deciso di salvare il file di configurazione dei parametri e l'output prodotto dallo script solutore e organizzarli in directory con data e ora dell'esecuzione; in questo modo è stata facilitata la fase di tuning e la successiva analisi dei tracciati.

²per motivi di spazio si è scelto di mostrare solo 3 cifre decimali, durante lo studio si è usata una precisione fino a 7 cifre dopo il punto

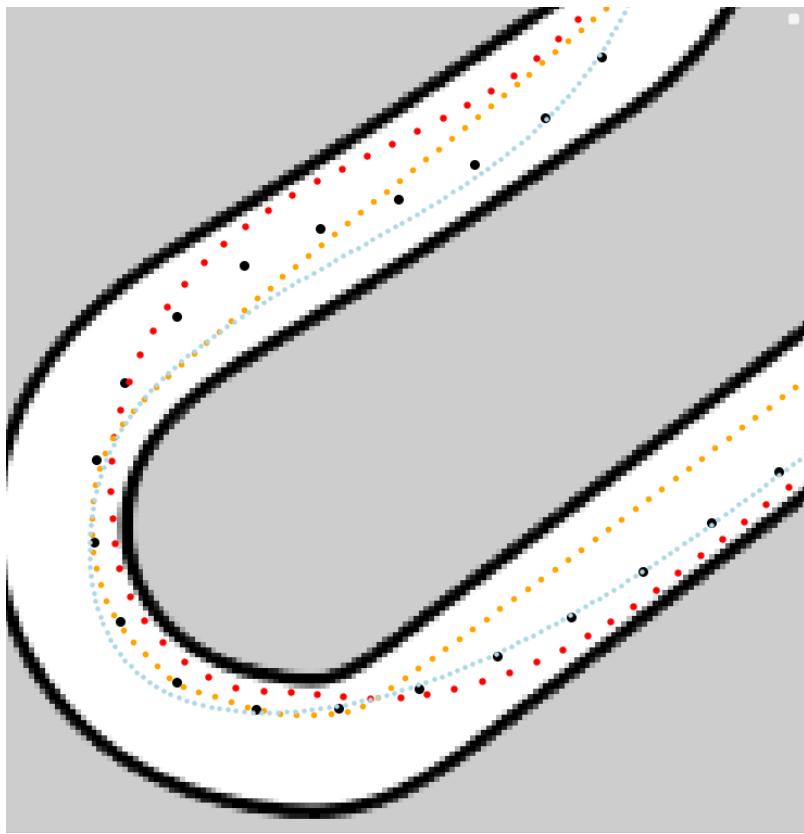


Figura 19: Quattro traiettorie con stepsize diversi: nera a 1.5m, rossa a 0.5m, arancione a 0.3m e azzurra a 0.15m

4.2.1 Tuning dei parametri

È facile immaginare che una minore distanza tra i sample durante l'ottimizzazione porti un risultato più preciso a costo di aumentare il tempo di esecuzione, come è emerso nel capitolo [successivo](#) di analisi (pag. 42). In figura 19 si visualizza una comparazione tra raceline con stepsize diversi.

Come ci si potrebbe aspettare, il parametro che controlla principalmente il tempo di esecuzione è lo stepsize durante l'ottimizzazione, ovvero `stepsize_reg`, per questo è stato necessario scegliere un valore tale che bilanci bene la qualità del tracciato prodotto e il tempo di esecuzione. Infatti, per evitare che l'ottimizzazione impieghi troppo tempo, il numero di iterazioni è limitato a 2000: se entro questo limite non è stata trovata una soluzione viene ritornato un errore.

In linea generale, si è riscontrato che:

- **stepsize_prep**: per quanto riguarda il percorso più breve e la curvatura minima, impatta sulla qualità del risultato, diversamente dal tempo minimo; non influisce significativamente sul tempo di esecuzione. Valori in $[0.1, 0.5)$ per i primi due, $[0.5, 1]$ per mintime;
- **stepsize_reg**: come accennato precedentemente, incide sul tempo di esecuzione e in modo apprezzabile per mintime; è significativo per la qualità del percorso. Valori > 0.3 per shortest path e mincurv, mentre per mintime sono necessari valori > 1.0 per evitare di sfornare le 2000 iterazioni;
- **stepsize_interp_after_opt**: non incide in modo sostanziale sul tempo e per questo può essere valorizzato in base alle necessità dell'algoritmo che userà il percorso.

A valori alti degli stepsize (es: > 1.5), può capitare che la raceline generata "tagli" il percorso per curve molto rapide, come nella chicane di Spa. Questo è capitato principalmente con l'algoritmo del percorso più breve, come mostrato alla figura 20

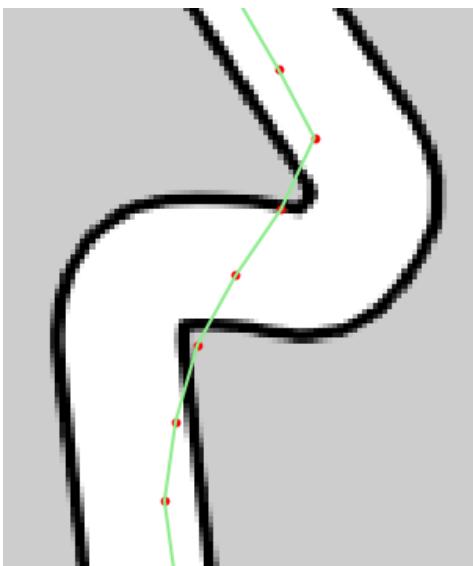


Figura 20: Generazione errata del percorso per via di stepsize troppo ampi

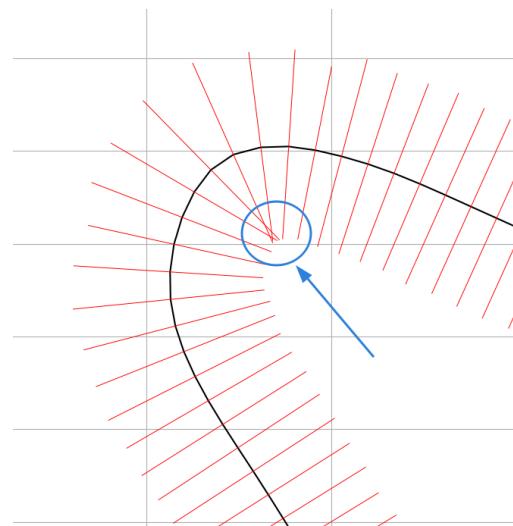


Figura 21: Linee ortogonali alla centerline incrociate

Altri parametri modificati durante le sperimentazioni sono qualità inerenti alle spline, principalmente lo smoothing factor. Quest'ultimo, se troppo basso e in concomitanza con valori di stepsize bassi, può generare delle linee ortogonali che si incrociano tra loro e che quindi non possono produrre un percorso corretto.

Un esempio di questo caso è visibile all'immagine 21. Durante le sperimentazioni questo valore è stato fissato a 80. Di seguito si mostrano gli stepsize usati per generare il percorso a figura 20.

```
stepsize_opts={  
    "stepsize_prep": 1.0,  
    "stepsize_reg": 1.5,  
    "stepsize_interp_after_opt": 1.5  
}
```

Mentre i successivi sono i parametri della spline e stepsize utilizzati per la figura 21.

```
stepsize_opts={  
    "stepsize_prep": 0.1,  
    "stepsize_reg": 0.5,  
    "stepsize_interp_after_opt": 0.3  
}  
  
reg_smooth_opts={  
    "k_reg": 3,    # grado della spline  
    "s_reg": 1.0  # smoothing factor [1.0, 100]  
}
```

Capitolo 5

Analisi dei risultati

In questo capitolo verranno comparate le tre strategie usate e descritte in questa tesi. Verranno presentate delle metriche su cui l'ottimizzazione lavora e altre di contorno sulla qualità del percorso generato corredate da grafici e rappresentazioni grafiche del tracciato. Sono state prese in considerazione due mappe: il circuito di Spa-Francorchamps in Belgio e il circuito di Monza.

I dati usati in questo capitolo sono stati estrapolati solamente dai file di output dell'ottimizzazione e analizzati con l'uso di Jupyter Notebook.

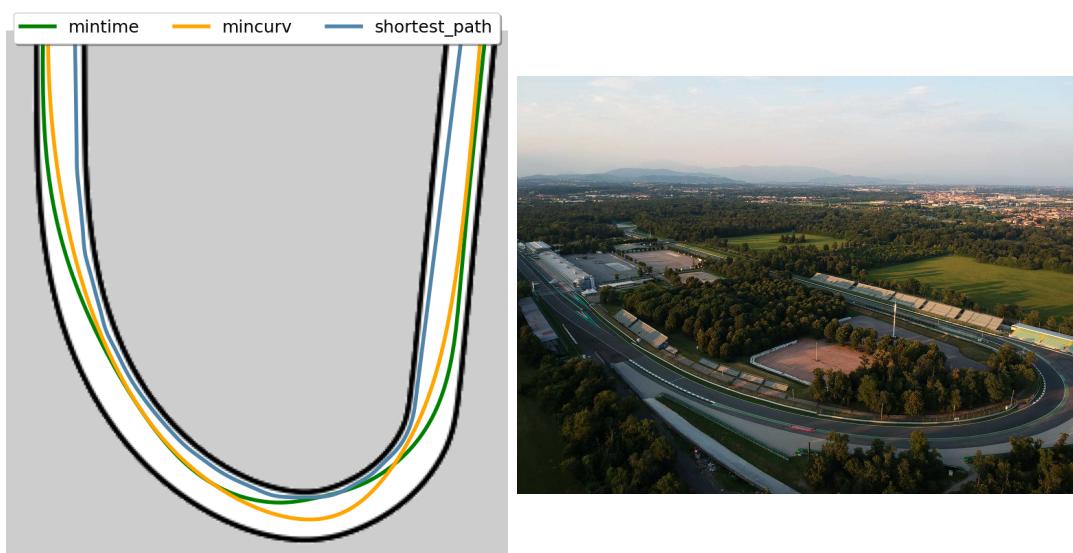


Figura 22: Raceline generate per la curva parabolica di Monza

5.1 Metriche

Le metriche usate in questo studio sono le seguenti:

- il tempo di percorrenza (lap time);
- lunghezza del tracciato prodotto;
- velocità;
- accelerazione;
- curvatura;
- tempo di esecuzione.

Le metriche scelte permettono di confrontare le tre strategie considerando quale criterio ottimizzano: il lap time per la strategia del percorso col tempo minimo, la lunghezza del tracciato per quella del percorso più corto e, infine, la curvatura per la strategia della curvatura minima. Ulteriori metriche di contesto sono la velocità, l'accelerazione e il tempo di esecuzione dei tre algoritmi.

Per ogni mappa e per ogni strategia vengono confrontati lap time, curvatura mediana e la lunghezza del tracciato risultante. Si è scelta la curvatura mediana perché quella media risultava sempre molto vicino a zero e quindi poco significativa per effettuare un confronto. Egual discorso per l'accelerazione, che tende a bilanciarsi tra accelerazioni e decelerazioni; a questa vengono anche aggiunte la deviazione standard, l'accelerazione massima e minima. Viene poi indicata la velocità media, la sua deviazione standard, la mediana e la minima, dato che la massima è stata impostata a 15 m/s .

Di seguito si mostrano le misurazioni per i due circuiti di Monza e Spa.

5.2 Spa

Tabella 1: Circuito di Spa

	Lap time [s]	Curvatura mediana [rad/m]	Lunghezza [m]
mintime	76.25	0.10	545.41
mincurv	49.22	0.07	546.87
shortest path	55.12	0.09	532.98

Osservanod la tabella 1, per quanto riguarda la curvatura e la lunghezza del circuito, le strategie di mincurv e shortest path effettivamente producono rispettivamente il risultato minore tra le strategie. Pare inaspettato il risultato per il tempo minimo, che è il più alto tra tutte le strategie, questo tuttavia è giustificato dalla migliore modellazione della dinamica del veicolo da parte dell'algoritmo: in questo senso, sebbene mincurv e shortest path abbiano un lap time minore, questo è anche meno affidabile rispetto al mintime. Il lap time viene calcolato sommando il tempo di percorrenza tra un sample e il successivo assumendo che la velocità rimanga costante tra questi due: dunque un tracciato con una velocità media più alta risulterà con un lap time minore; come si può notare dalla tabella 2, la velocità media della strategia mintime è decisamente inferiore rispetto alle altre due.

A conferma della più completa modellazione di mintime sono state le simulazioni con MPC sul simulatore per le quali, a fronte del percorso di mincurv, il robot raggiungeva una velocità troppo elevata e non riusciva a seguire bene il tracciato, uscendo spesso fuori dal circuito; mentre per quanto riguarda il tracciato di mintime il controller riusciva a seguire meglio il percorso.

Tabella 2: Velocità in m/s per Spa

	Media	Mediana	Minima	Dev. std
mintime	8.33	8.44	2.3	2.93
mincurv	12.05	12.82	3.47	2.88
shortest path	10.79	11.29	1.63	2.97

Come accennato al paragrafo precedente, la velocità media e mediana più alta appartiene alla strategia della curvatura minima, il che rimane in linea con la formulazione del problema, ovvero mantenere la velocità più alta in curva. Lo stesso è visivamente immediato osservando la figura 23, dove mincurv tende ad avere colori più verso il verde nelle curve rispetto alle altre due strategie.

La deviazione standard delle tre strategie rimane pressoché simile: solo il percorso più breve ne riporta una più alta che è congrua anche con la variabilità dell'accelerazione nella tabella 3.

Come è possibile notare dal grafico 24, solo mintime non ha mai toccato la massima velocità impostata, raggiungendo un picco di $14.7 m/s$ e solo per pochissimo tempo, mentre le altre due l'hanno tenuta per più tempo, mincurv tra tutte.

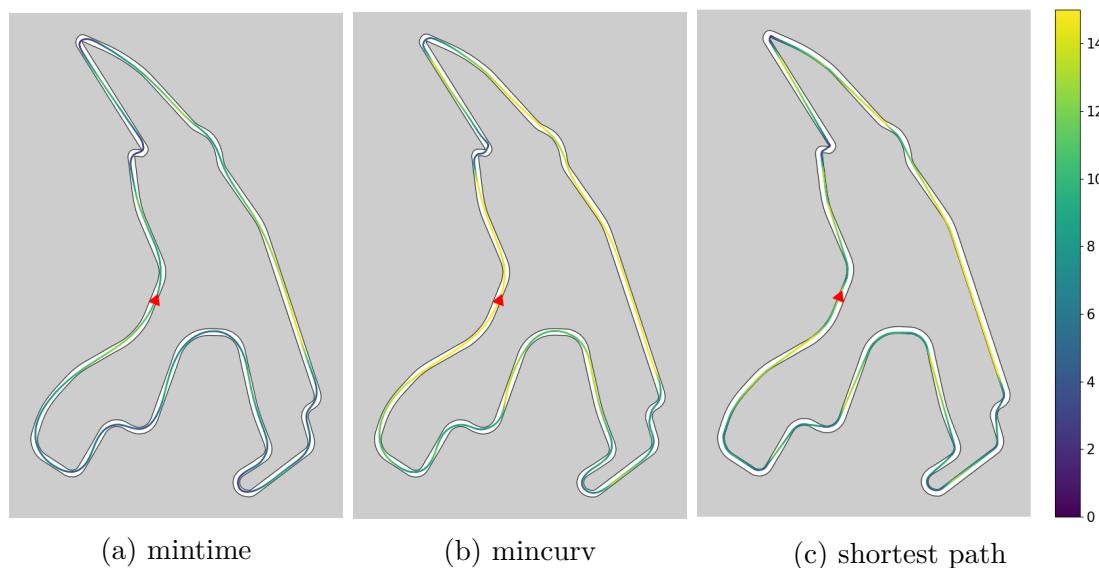


Figura 23: Confronto della velocità per le tre strategie

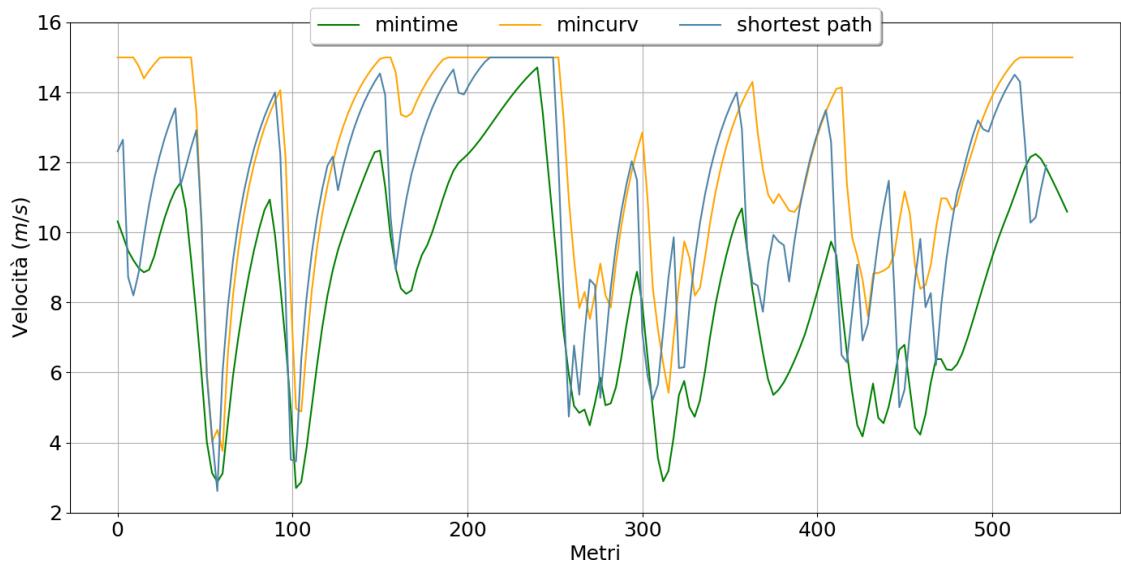


Figura 24: Profilo della velocità in Spa

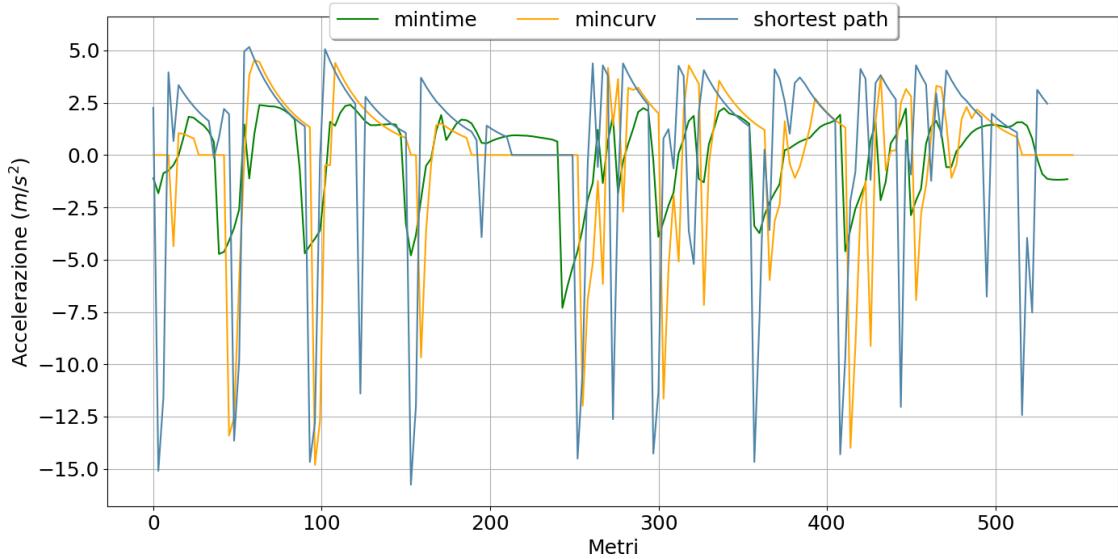


Figura 25: Profilo dell'accelerazione per Spa

Nell'accelerazione si osservano le maggiori differenze: si nota come la strategia del percorso minimo sia molto aggressiva nel cambio di accelerazione, infatti ha la deviazione standard maggiore tra tutte. È da sottolineare che i valori di massimo e minimo sono spesso picchi che vengono mantenuti solo per qualche frazione di metro – come mostrato in figura 25 – e quindi poco significativi nell'insieme; per questo, oltre a quanto scritto precedentemente, si è scelto di usare la mediana. In ogni caso, possono dare un'idea dei valori assunti, e quindi indicare quanto estremo è il range della strategia del percorso minimo rispetto al tempo minimo.

Qualitativamente, dall'immagine 26, si può notare come l'algoritmo per mintime produca una serie di accelerazioni e decelerazioni più graduale rispetto alle altre due, soprattutto con shortest path. Questo repentino e significativo cambio di accelerazione da parte di quest'ultimo, e di mincurv in misura minore come mostra la figura 25, produce dei tracciati difficili da applicare nella realtà e quindi di minor qualità rispetto a mintime.

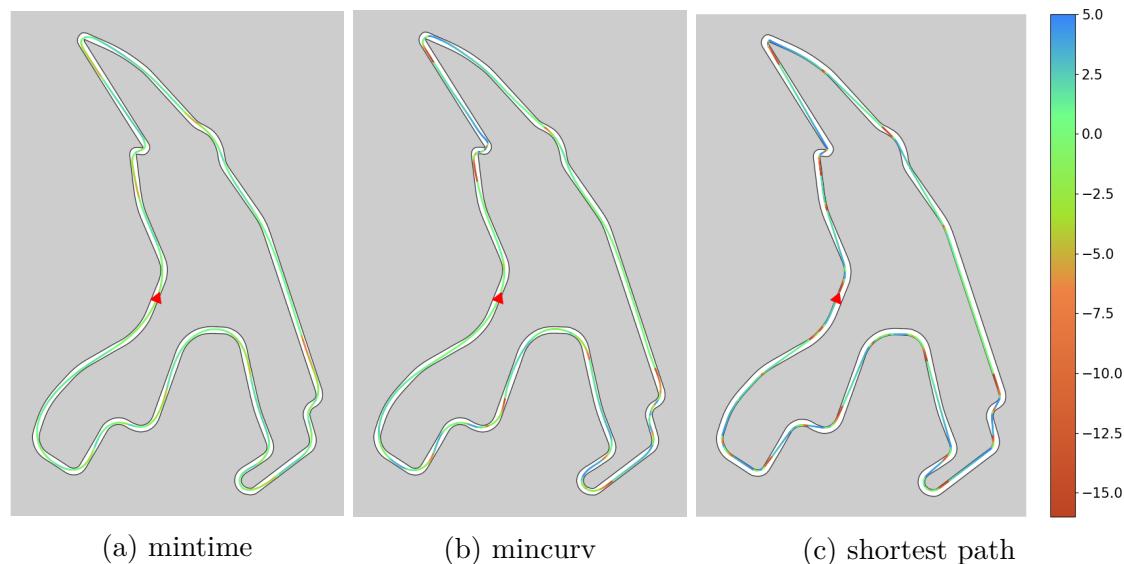


Figura 26: Confronto dell'accelerazione per le tre strategie

Tabella 3: Accelerazione in m/s^2 per Spa

	Mediana	Massima	Minima	Dev. std
mintime	0.79	2.57	-11.54	2.09
mincurv	0.75	4.98	-15.29	3.05
shortest path	1.83	5.29	-16.27	5.11

5.3 Monza

Tabella 4: Circuito di Monza

	Lap time [s]	Curvatura mediana [rad/m]	Lunghezza [m]
mintime	50.72	-0.07	441.37
mincurv	33.74	-0.03	439.57
shortest path	41.37	-0.12	434.1

Anche in questo caso valgono le stesse considerazioni descritte per il circuito di Spa sopra analizzato: gli algoritmi per mincurv e shortest path producono il risultato minore per le rispettive metriche, mentre sono poco affidabili per il lap time, di cui mintime offre un risultato più realistico per via della sua formulazione. Una nota da sottolineare è che l'algoritmo per mincurv è riuscito a minimizzare maggiormente la curvatura rispetto alla mappa precedente; in questo senso minimizzare si intende tendere a zero, dunque è necessario considerare i valori assoluti per la colonna della curvatura alla tabella 4.

Tabella 5: Velocità in m/s per Monza

	Media	Mediana	Minima	Dev. std
mintime	9.84	9.82	3.21	3.09
mincurv	13.41	14.56	7.88	2.04
shortest path	11.34	11.52	4.67	2.83

Anche per la velocità valgono alcune considerazioni fatte per Spa: la strategia di mintime è più conservativa rispetto alle altre, ma in questo caso ha la deviazione standard più alta tra tutte. In questo caso, anche mintime ha raggiunto la velocità massima indicata, tuttavia l'ha raggiunta solo una volta e comunque per pochi metri, come è possibile alla figura 28; mentre, anche in questo caso, mincurv è al primo posto per aver mantenuto per più tempo la velocità massima, complice anche la topologia della mappa che offre maggiori possibilità grazie ai suoi rettilinei: qualitativamente si nota alla figura 27 come per micurv venga indicato il colore giallo anche sulla *Curva Grande* ad indicare una velocità prossima a quella massima, come non avviene per le altre due strategie.

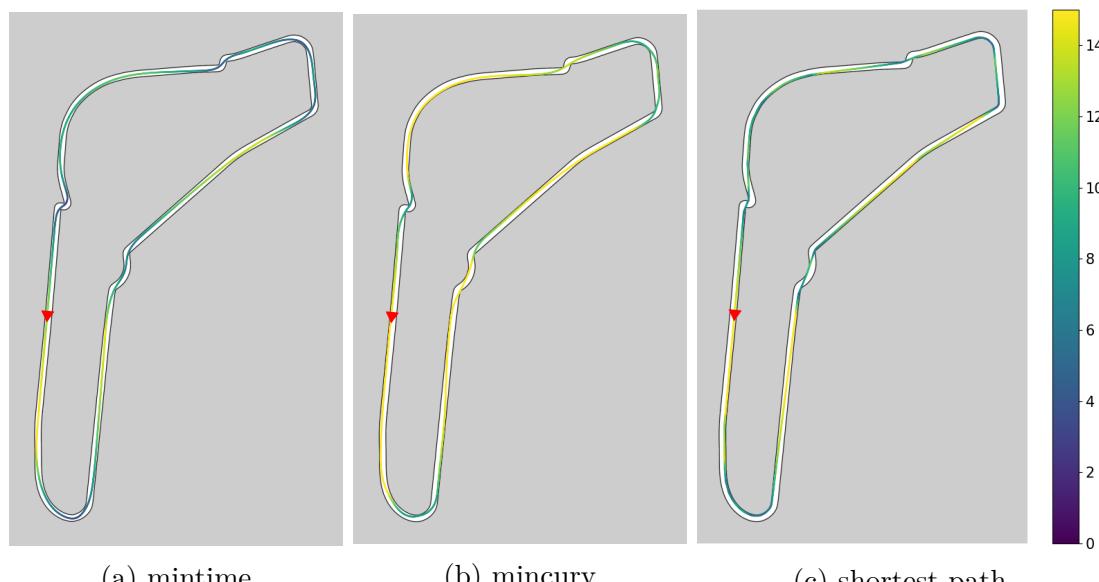


Figura 27: Confronto della velocità per le tre strategie

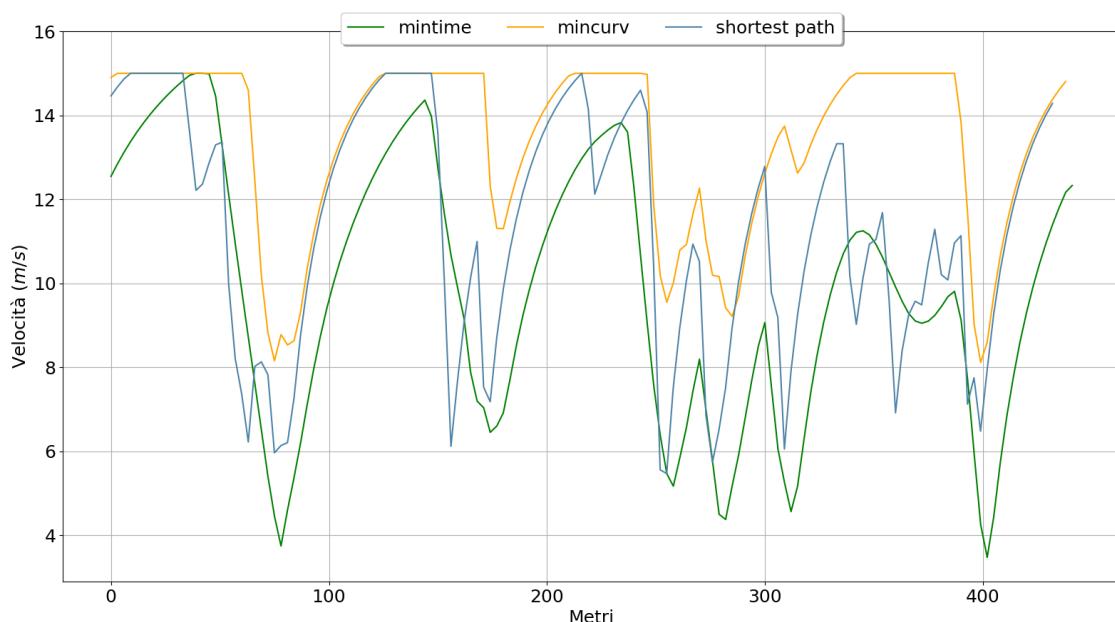


Figura 28: Profilo della velocità per Monza

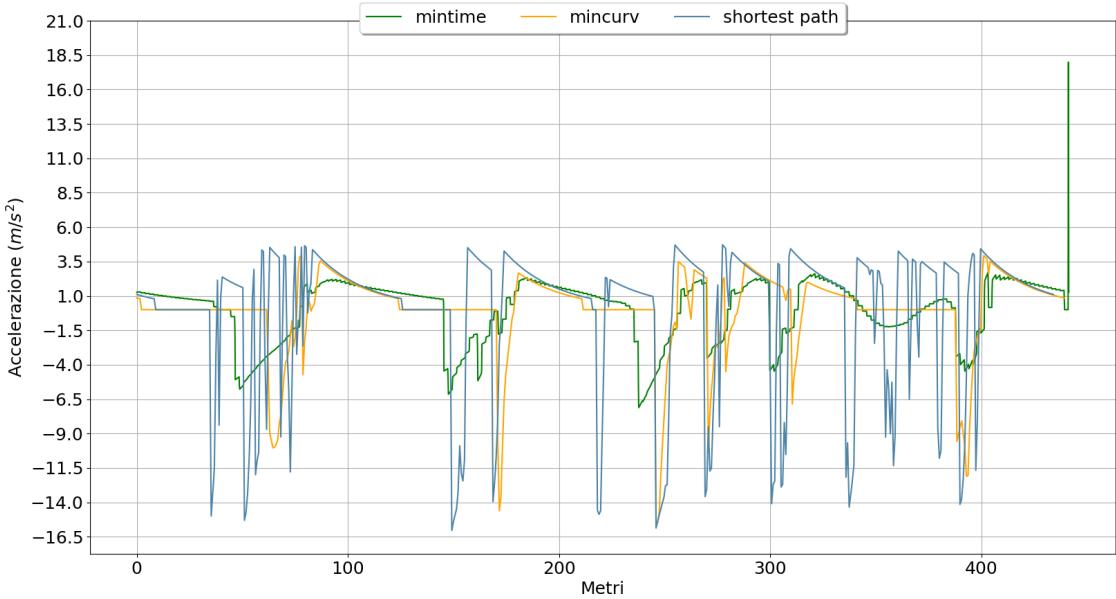


Figura 29: Profilo dell'accelerazione per Monza

A differenza di Spa, la variabilità della strategia del percorso più breve risulta in linea con le altre strategie e la sua mediana risulta simile alla rispettiva di Spa, come risulta dalla tabella 6. Spicca il valore fuori scala dell'accelerazione massima della strategia di mintime: analizzando il dataset e osservando il grafico alla figura 29 si nota immediatamente che si tratta di un outlier generato, tra l'altro, solo per un singolo sample e quindi non veramente applicabile dal robot reale o simulato. In ultima analisi, si tratta di un errore relativo a questa esecuzione in particolare.

Qualitativamente dalla figura 30, anche in questo circuito si nota come mintime sia più graduale, soprattutto rispetto a shortest path, dove si può osservare più volte repentini sbalzi di accelerazione, che sono impraticabili per un veicolo.

Tabella 6: Accelerazione in m/s^2 per Monza

	Mediana	Massima	Minima	Dev. std
mintime	0.80	17.98	-7.11	2.22
mincurv	0.00	4.00	-15.48	2.86
shortest path	1.63	4.78	-16.31	3.13

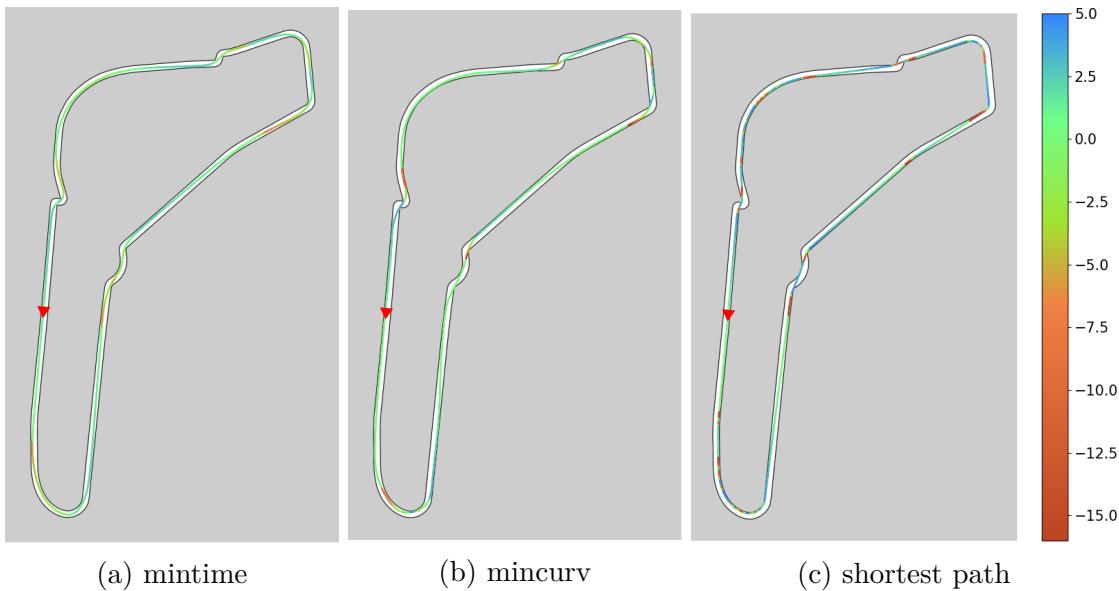


Figura 30: Confronto dell’accelerazione per le tre strategie

5.4 Tempo di Esecuzione

Un aspetto interessante che è stato analizzato è il tempo di esecuzione per ogni algoritmo. In generale si è sperimentato un tempo tra pochi secondi e alcuni minuti dal lancio del solutore fino all’esportazione in csv del risultato; tuttavia questo tempo racchiude anche controlli sul percorso generato ed eventuali input dell’utente, quindi non è un buon indicatore del singolo processo di risoluzione. Dunque si è fatto riferimento alla sola risoluzione del problema matematico e non a processi precedenti o successivi.

Di seguito si mostra una tabella che indica il numero di secondi del tempo di risoluzione di ogni strategia per 6 valori di `stepsize_reg` diversi, mentre `stepsize_prep` è stato fissato a 0.75 e `stepsize_interp_after_opt` a 0.15.

Tabella 7: Tempo di esecuzione in s per ogni stepsize

Stepsize	0.75	1.00	1.5	2.0	2.5	3.0
mintime	fallito	fallito	526.88	266.29	257.09	138.59
mincurv	0.798	0.372	0.167	0.094	0.072	0.048
shortest path	0.361	0.313	0.086	0.041	0.022	0.013

Sono evidenti i fallimenti della strategia del percorso più breve a i primi due valori dello stepsize, la motivazione è stata data al capitolo 4 al paragrafo 4.2.1, pag. 30. Si nota, in ogni caso, un'importante differenza di scala tra il tempo minimo e le altre due strategie, questo è dato principalmente dalla complessità e una più completa modellazione del veicolo dell'algoritmo e che quindi ha bisogno di una maggiore computazione, come descritto al capitolo 3, pag. 14.

Come anche accennato al paragrafo sopra citato, si può notare che l'aumentare degli stepsize spesso dimezzi la velocità nel trovare una soluzione, o in generale la diminuisce, dato appunto dal numero minore di sample da considerare. In figura 31 si può osservare come l'andamento di tutte le tre strategie possa essere approssimato grossomodo ad una funzione quadratica inversa.

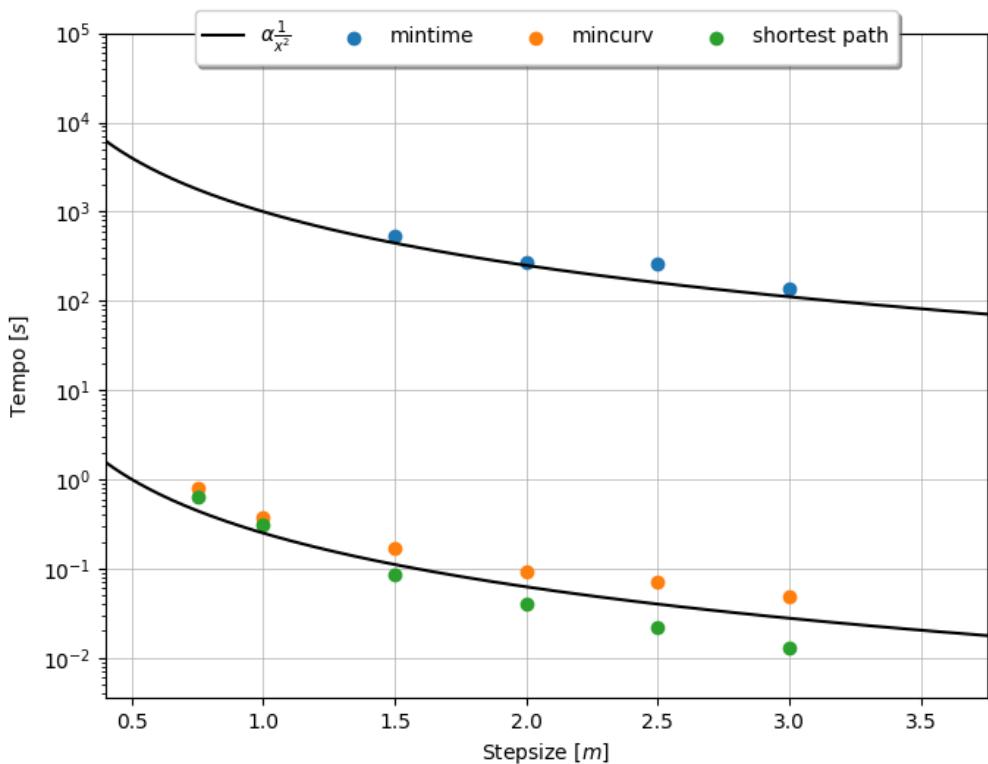


Figura 31: Grafico dell'andamento del tempo di esecuzione secondo diversi valori di `stepsize_reg`

Capitolo 6

Conclusioni

Di seguito si discute dei risultati ottenuti durante l'analisi ed eventuali sviluppi futuri.

6.1 Risultati ottenuti

Lo studio di questa tesi si è concentrato sulle soluzioni trovate in letteratura per l'ottimizzazione del *percorso globale* data la mappa di un circuito. Nello specifico, sono state analizzate *tre strategie* di ottimizzazione: il percorso più breve, il percorso con la curvatura minima e il percorso con il lap time minore. Le tre strategie vengono modellate secondo un problema di controllo ottimale.

Dalle analisi svolte e da sperimentazioni svolte con il simulatore è emerso che la formulazione delle prime due strategie risulta troppo semplice per poter essere applicata anche in un simulatore: nel caso della curvatura minima, sebbene sia l'obiettivo della strategia, la velocità mediana è troppo alta – nel caso in cui la massima sia impostata in egual modo per tutte le tre strategie – mentre per il percorso con lunghezza minima la traiettoria risultante è troppo vicina ai bordi e perciò difficile da far seguire ad un controller. I risultati migliori sono stati ottenuti dalla strategia del tempo minimo, principalmente perché integra i coefficienti d'attrito tra gli pneumatici e la strada, dunque risulta un modello più realistico, soprattutto nel caso come questo, dove si spinge il veicol ai suoi limiti.

6.2 Sviluppi futuri

Un possibile miglioramento è sicuramente quello di raffinare meglio i parametri degli algoritmi, in modo tale che producano traiettorie praticabili: dunque diminuire la velocità massima per la strategia mincurv, o comunque trovare una relazione tra quella massima delle altre due e quella di mincurv, e aumentare il distanziamento

tra il bordo del circuito e la traiettoria così da permettere il passaggio del veicolo. Inoltre, si potrebbe ulteriormente aggiornare la modellazione del problema di QP sottostante inserendo nuovi vincoli.

Un ulteriore approccio alla risoluzione del problema è quello di implementare e analizzare algoritmi di ottimizzazione basati su *strategie evolutive*, come il già citato CMA-ES (cap. 3 pag. 14). [20] In generale, è possibile esplorare anche altri algoritmi di ottimizzazione matematica.

Un problema comune in questi contesti è quello che viene chiamato *Sim2Real*, ovvero quello di *trasportare* il lavoro svolto dal simulatore, o comunque da un modello teorico, alla realtà. Il modello è una semplificazione della realtà, e in alcuni casi questo può portare ad un comportamento errato se portato direttamente in hardware. Il passaggio da simulatore a realtà spesso è seguito da un ulteriore fase di tuning dei parametri degli algoritmi così da adattarli al meglio alla complessità della realtà. Un possibile sviluppo futuro, dunque, potrebbe essere quello di costruire il robot e implementare il lavoro svolto in simulazione, testarlo ed adattarlo.

Ringraziamenti

Seppur banale, come prima cosa vorrei ringraziare **i miei genitori** che mi hanno permesso di intraprendere questo percorso. Li ringrazio per avermi dato la libertà di seguire le mie passioni e, nei momenti giusti, costringermi a valutare anche altre opzioni: senza *mia mamma* non avrei conosciuto l'informatica.

Sebbene non abbiamo avuto un ruolo rilevante durante il percorso universitario in senso stretto, vorrei ringraziare anche i miei secondi genitori, **i nonni materni**, con cui sono cresciuto: grazie *nonna* e grazie *nonno* per tutte le cose che avete fatto per me in tutti questi anni.

Durante questa esperienza ho incontrato i miei attuali amici e altri mi hanno accompagnato fin da prima; vorrei ringraziarli tutti per il supporto che mi hanno dato durante questi anni, tra esami, appunti e ore passate su Discord a studiare, chiaccherare e giocare assieme. Con alcuni di loro ho fatto anche nuove esperienze tra vacanze e gite e conosciuto altra gente: insomma, chi di più chi di meno, mi hanno aiutato a crescere e maturare.

Mi accompagnano fin dalle superiori **Vincenzo Siano** (*WinCE* detto vince) e **Cristian Pozzi** (*criii/pohtzie* detto pozzi), senza di loro sarebbe stato più difficile iniziare l'università; ma abbiamo fin da subito conosciuto – non in ordine di importanza – **Federico Marcelli Fabiani** (*fede/fefè*), **Davide Papasodaro** (*papass/dadæ*), **Gabriele Giorgio** (*gabri*), **Marco Morandi** (*marco/mora*), **Gabriele Gilberti** (*gigi*), **Federico Coscia** (*fedoscia*).

È stato un percorso travagliato e pieno di ostacoli, tra esami, scadenze e (alcuni) professori; ho cercato di aiutare a mia volta coloro che l'hanno fatto con me, ciò nonostante arriveremo tutti al nostro traguardo finale (chi prima, chi dopo).

È stato divertente, ragazzi.

Bibliografia

- [1] Johannes Betz et al. «Autonomous vehicles on the edge: A survey on autonomous vehicle racing». In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 458–488.
- [2] Francesco Braghin et al. «Race driver model». In: *Computers & Structures* 86.13-14 (2008), pp. 1503–1516.
- [3] Fabian Christ et al. «Time-optimal trajectory planning for a race car considering variable tyre-road friction coefficients». In: *Vehicle system dynamics* 59.4 (2021), pp. 588–612.
- [4] *Differenze tra ROS e ROS2*. URL:
<https://web.archive.org/web/20240825200348/https://design.ros2.org/articles/changes.html>.
- [5] *DrivingFast website*. URL:
<https://web.archive.org/web/20240816085814/https://drivingfast.net/racing-line/#chapter-0>.
- [6] *F1TENTH about page*. URL: <https://f1tenth.org/about.html>.
- [7] *F1TENTH Gym Repository*. URL:
https://github.com/f1tenth/f1tenth_gym_ros.
- [8] *F1Tenth overtaking with MPC*. URL:
https://www.youtube.com/watch?v=tD-BrSw_9QA.
- [9] *F1TENTH racetracks*. URL:
https://github.com/f1tenth/f1tenth_racetracks.
- [10] *F1TENTH website*. URL: <https://f1tenth.org>.
- [11] Steven Gong. *Image Transform*. URL:
<https://web.archive.org/web/20231207164444/https://stevengong.co/notes/Image-Transform>.
- [12] Steven Gong. *Raceline Optimisation*. URL:
<https://web.archive.org/web/20231207164444/https://stevengong.co/notes/Raceline-Optimization>.

- [13] Thomas Lipp e Stephen Boyd. «Minimum-time speed optimisation over a fixed path». In: *International Journal of Control* 87.6 (2014), pp. 1297–1311.
- [14] Steven Macenski et al. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. DOI: [10.1126/scirobotics.abm6074](https://doi.org/10.1126/scirobotics.abm6074). URL: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>.
- [15] Jacob Olausson e Jacob Larsson. *Optimal control and race line planning for an autonomous race car*. 2021.
- [16] University of Pennsylvania. *Course Overview*. Con comm. di Rahul Mangharam. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [17] University of Pennsylvania. *Lezione 01 di F1TENTH. Introduction to ROS2*. Con comm. di Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [18] University of Pennsylvania. *Lezione 04 di F1TENTH. PID for Wall-following*. Con comm. di Rahul Mangharam. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [19] University of Pennsylvania. *Lezione 10 di F1TENTH. Pure Pursuit*. Con comm. di Hongrui Zheng Rahul Mangharam. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [20] University of Pennsylvania. *Lezione 22 di F1TENTH. Raceline Optimization*. Con comm. di Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [21] University of Pennsylvania. *Local Planner: RRT, Spline Based Planner*. Con comm. di Rahul Mangharam. URL: https://docs.google.com/presentation/d/1W9_nQ-mgeIkZSo40hkxwzSX2LjKG13JTg72nzjMfnkg.
- [22] *PID controller*. URL: https://en.wikipedia.org/wiki/Proportional%20integral%20derivative_controller.
- [23] *ROS2 basic concepts - Actions*. URL: <https://web.archive.org/web/20240520092821/https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html>.
- [24] *ROS2 basic concepts - Launch files*. URL: <https://web.archive.org/web/20240809063614/http://docs.ros.org/en/humble/Concepts/Basic/About-Parameters.html>.

- [25] *ROS2 basic concepts - Messages.* URL:
<https://web.archive.org/web/20241002162417/https://docs.ros.org/en/iron/Concepts/Basic/About-Interfaces.html#messages>.
- [26] *ROS2 basic concepts - Nodes.* URL:
<https://web.archive.org/web/20240218141117/https://docs.ros.org/en/iron/concepts/basic/about-nodes.html>.
- [27] *ROS2 basic concepts - Parameters.* URL:
<https://web.archive.org/web/20241001174044/http://docs.ros.org/en/humble/Concepts/Basic/About-Launch.html>.
- [28] *ROS2 basic concepts - Services.* URL:
<https://web.archive.org/web/20241003081427/https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html>.
- [29] *ROS2 basic concepts - Topics.* URL:
<https://web.archive.org/web/20241002162221/https://docs.ros.org/en/iron/Concepts/Basic/About-Topics.html>.
- [30] *The ROS Ecosystem.* URL:
<https://web.archive.org/web/20241001145206/https://www.ros.org/blog/ecosystem/>.
- [31] Rainer Trauth, Phillip Karle e Markus Lienkamp. *Autonomous Driving Software Engineering - Lecture 06: Planning I - Global Planning*. Giu. 2021.
- [32] *Understanding ROS2 nodes.* URL:
<https://web.archive.org/web/20240528112616/https://docs.ros.org/en/humble/tutorials/beginner-cli-tools/understanding-ros2-nodes/understanding-ros2-nodes.html>.
- [33] *Understanding ROS2 topics.* URL:
<https://web.archive.org/web/20240518194349/https://docs.ros.org/en/iron/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html>.