

UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI



CORSO DI LAUREA IN INFORMATICA

OTTIMIZZAZIONE DEL PATH TRACKING CON MODEL
PREDICTIVE CONTROL PER AUTONOMOUS RACING
IN F1TENTH

Relatore: Matteo Luperto
Correlatore: Michele Antonazzi

Tesi di Laurea di:
Vincenzo Siano
Matr. Nr. 981734

ANNO ACCADEMICO 2023-2024

Indice

Indice	i
1 Introduzione	1
1.1 F1TENTH	3
1.2 ROS 2	5
1.2.1 Nodi	6
1.2.2 Topic	7
1.2.3 Parametri e Launch file	7
1.3 Simulatore: F1TENTH Gym	8
1.4 Progettazione dei veicoli a guida autonoma	10
1.4.1 Perception	11
1.4.2 Planning	13
1.4.3 Control	14
2 Metodi reattivi per il controllo	17
2.1 PID Controller	17
2.1.1 Struttura	18
2.1.2 Tuning	19
2.2 Pure Pursuit	20
2.3 Vantaggi e svantaggi	21
3 Model Predictive Control	22
3.1 Struttura	24
3.1.1 Receding Horizon Control	26
3.2 Modello	28
3.2.1 Kinematic Bicycle Model	29
3.3 Funzione Obiettivo	31
3.4 Vincoli	32
3.4.1 Stati futuri e dinamica	33
3.4.2 Stato e input	33
3.5 Algoritmo	33

3.6	Quadratic Programming	34
3.6.1	Discretizzazione	36
3.6.2	Linearizzazione	36
3.6.3	Conversione	37
3.7	Vantaggi e svantaggi	38
4	Implementazione	39
4.1	Circuiti	39
4.2	Configurazione	40
4.3	Risolutore	42
4.4	Obiettivi	43
4.5	Vincoli	44
4.6	Tuning con le penalità	46
4.6.1	Spa	47
4.6.2	Monza	48
4.7	Visualizzazione	48
5	Analisi dei risultati	50
5.1	Metriche usate	50
5.1.1	Circuiti	52
5.2	Visualizzazione dei risultati	54
5.2.1	Lap time	54
5.2.2	Traiettorie	55
5.2.3	Crosstrack Error	58
5.2.4	Velocità e Angolo di sterzata	60
6	Conclusioni	63
6.1	Risultati ottenuti	63
6.2	Sviluppi futuri	64
Ringraziamenti		66
Bibliografia		68

Capitolo 1

Introduzione

Nell’ultimo decennio il settore della guida autonoma ha fatto grandi progressi grazie all’introduzione di nuovi algoritmi e tecniche avanzate nel campo della robotica e dell’intelligenza artificiale. L’*autonomous driving* rappresenta una sfida tecnologica molto complessa poiché è caratterizzata da diversi ambiti di ricerca, quali percezione, pianificazione e controllo. In aggiunta alle più classiche applicazioni di guida autonoma, come il trasporto pubblico urbano e le automobili con funzionalità di guida assistita o totalmente autonoma, un settore altamente competitivo che sta attirando sempre più attenzione è quello dell’*autonomous racing*. Le auto da corsa possono essere veicoli reali – come una vettura di *Formula 1* – o veicoli a scala ridotta – ad esempio scala 1:10 per *F1TENTH*. Proprio comunità come quella di *F1TENTH* stanno raggiungendo ottimi risultati grazie a delle vere e proprie competizioni, che permettono di far gareggiare ingegneri e ricercatori provenienti da tutto il mondo con veicoli autonomi in condizioni estreme, mettendo alla prova le prestazioni delle proprie soluzioni in ambienti a velocità elevate, come dei circuiti. I partecipanti possono così affrontare le sfide del controllo più estreme in un ambiente controllato, riflettendo però le dinamiche del mondo reale.

A differenza della guida su strada, lo scopo è quello di guidare in sicurezza nel modo più veloce ed efficiente, riducendo i tempi di completamento su piste competitive. In questo contesto, vi sono dunque due sfide cruciali: la pianificazione della traiettoria e sistemi di controllo robusti e accurati, in grado di gestire variazioni improvvise.

Inoltre, il controllo del veicolo ha il compito di ottimizzare il percorso seguito e di minimizzare gli errori di traiettoria.

L'obiettivo del lavoro di tesi è sviluppare e analizzare l'efficacia di *Model Predictive Control (MPC)*, un metodo di controllo avanzato per l'ottimizzazione dell'attività di *path tracking*, applicata in un contesto di *autonomous racing* su veicoli in scala 1:10 per la piattaforma *F1TENTH*. Attraverso l'implementazione dell'algoritmo e la simulazione su diversi tracciati di *Formula 1*, si confrontano le prestazioni di *MPC* con quelle di *Pure Pursuit*, un altro algoritmo di controllo più semplice basato su *metodi reattivi*, secondo diverse metriche individuate.

Il lavoro è stato suddiviso nelle seguenti fasi:

Capitolo 1: Introduzione Si fornisce una panoramica di *F1TENTH* e del contesto dell'*autonomous racing*. In seguito, si introducono i concetti chiave di *ROS*, alla base di tutta l'implementazione del progetto.

Capitolo 2: Metodi reattivi per il controllo Si discutono i metodi di controllo reattivi, come il *PID Controller*, individuando vantaggi e limitazioni di queste soluzioni per il contesto trattato.

Capitolo 3: Model Predictive Control Si descrive l'algoritmo oggetto di questa tesi, trattandone la struttura, l'*orizzonte*, il modello utilizzato e formulando il problema di controllo ottimale (obiettivo e vincoli). Si discute poi di discretizzazione, linearizzazione e dei vantaggi e svantaggi di questo metodo.

Capitolo 4: Implementazione Vengono presentati i circuiti utilizzati per mettere alla prova *MPC* e le sue diverse configurazioni di guida. Dopodiché, si descrive il risolutore utilizzato e si discute il codice degli obiettivi e dei vincoli. Infine, si approfondisce la fase di *tuning* delle matrici di pesi, fondamentali per l'obiettivo.

Capitolo 5: Analisi dei risultati Si analizzano i risultati dei profili di *MPC* su entrambe le piste, confrontandoli tra loro e con il *Pure Pursuit*.

Capitolo 6: Conclusioni Si riassumono i principali risultati ottenuti e si menzionano possibili sviluppi futuri.

1.1 F1TENTH

F1TENTH è una comunità internazionale di ricercatori, ingegneri, sviluppatori e appassionati di sistemi autonomi che ogni anno organizza una serie di gare tra auto autonome da corsa in scala 1:10, nelle quali squadre provenienti da tutto il mondo si riuniscono per competere [1]. Fondata nel 2016 dall'*University of Pennsylvania*, comprende ora diverse università provenienti da tutto il mondo, le quali hanno incluso *F1TENTH* nei programmi dei propri corsi di laurea, portando a un ulteriore sviluppo della comunità globale.



Figura 1: Robot mobile di *F1TENTH* in scala 1:10 [1].

La piattaforma di sviluppo di *F1TENTH* incentiva lo studio e la ricerca nelle discipline della guida autonoma, della robotica e dell'intelligenza artificiale; inoltre, essa permette di andare a sviluppare anche le capacità analitiche necessarie per ragionare su situazioni etiche nella fase di progettazione dei propri veicoli autonomi.

L'obiettivo degli organizzatori consiste quindi nel fornire le basi sulle più recenti tecnologie implementate e testate sulle auto a guida autonoma e, più in generale, sui sistemi mobili autonomi.

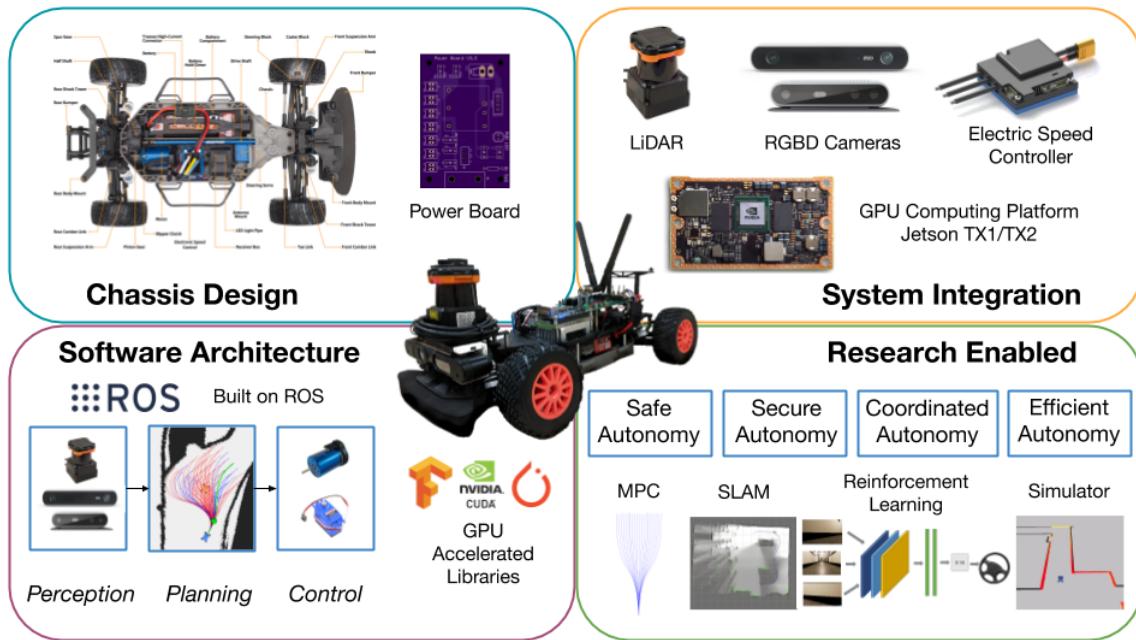


Figura 2: Stack architetturale di *F1TENTH* [2].

In primo luogo, è stato effettuato uno studio preliminare della piattaforma F1TENTH e dei concetti fondamentali della guida autonoma. Essi comprendono le seguenti macro-aree tematiche:

- Infrastruttura di *ROS 2* e il simulatore *F1TENTH Gym*;
 - Dinamica del veicolo e *metodi reattivi* – Comprende la modellazione e la simulazione della dinamica del veicolo, i metodi di navigazione reattiva, come l'algoritmo *PID* applicato al *Wall Following* e l'algoritmo *Follow the Gap* per evitare gli ostacoli.
 - *Mapping* e *Localization* – Concetti come la stima dello stato, la modellazione dell'ambiente, le tecniche basate su filtri per la localizzazione del robot e la tecnica *SLAM*.
 - *Control* e *Planning* – Tematiche fondamentali per la progettazione di veicoli a guida autonoma, come il *path tracking* con *Pure Pursuit* e gli algoritmi di

pianificazione del movimento di un robot, come *RRT*, altri metodi basati su *spline* o *clostoids* e altri tipi di *planner*.

Inoltre, durante questa fase iniziale sono stati implementati, mediante *Python* e le relative librerie per ROS 2, i seguenti algoritmi:

1. *Frenata di emergenza automatica*: si calcola il tempo istantaneo alla collisione (*iTTC*) grazie al messaggio `LaserScan` nel simulatore, al cui interno è presente un array che contiene tutte le misurazioni del LiDAR.
2. *Wall Following*: si è implementato un *controller PID* per far correre l'auto parallelamente alle pareti di un corridoio ad una distanza fissa;
3. *Follow the Gap*: si tratta di un algoritmo *reattivo* per evitare gli ostacoli che permette al robot di percorrere più giri all'interno della mappa utilizzata;
4. *Pure Pursuit*: è stato utilizzato *SLAM* per effettuare il *mapping* e, al contempo, la localizzazione del veicolo. Il *mapping* è il processo di creazione di una rappresentazione dell'ambiente circostante, mentre la *localizzazione* consiste nel determinare la posizione dell'auto rispetto alla mappa. Invece, per ottenere la localizzazione a partire dalla mappa generata, è stato impiegato un *Particle Filter*. È stato così possibile implementare l'algoritmo *Pure Pursuit*, nel quale viene calcolata iterativamente la curvatura dell'arco da seguire. Questo metodo, infatti, permette di seguire una traiettoria composta da più punti (*waypoint*).
5. *Motion Planning*: è stata realizzata una struttura per il controllo delle collisioni del veicolo, detta *occupancy grid*. In seguito, è stato implementato l'algoritmo *RRT** come *local planner* e si è utilizzato *Pure Pursuit* come controller.

1.2 ROS 2

Il lavoro oggetto di questa tesi è sviluppato a partire da *Robot Operating System (ROS)*, che consiste in un insieme di librerie software e strumenti per creare applicazioni robotiche [3]. Nato nel 2007, il progetto ha visto numerosi cambiamenti, tra

cui il passaggio da ROS 1 a ROS 2. Contrariamente a come si potrebbe pensare dal nome, *ROS* non è un sistema operativo, bensì si tratta di un *middleware* basato su un meccanismo di pubblicazione e sottoscrizione anonimo che consente lo scambio di messaggi tra diversi processi. Un *middleware* è un software che si trova a metà tra un sistema operativo e le applicazioni in esecuzione al suo interno e permette la comunicazione e la gestione dei dati per applicazioni distribuite [4].

Si tratta di un progetto *open source* supportato da una grande comunità di ricercatori nell'ambito della robotica, i quali effettuano rilasci per più distribuzioni *ROS*. Alcune di queste sono dotate di supporto a lungo termine, dunque sono più stabili e vengono sottoposte a test approfonditi; rientra in questa categoria *ROS 2 Humble*, quella usata per questo lavoro.

ROS 2 fornisce tutte le funzionalità necessarie per la realizzazione di applicazioni robotiche, permettendo dunque di risparmiare più tempo e risorse per sviluppo vero e proprio, una caratteristica cruciale, soprattutto, in un contesto lavorativo. Nello specifico, sono supporti due linguaggi di programmazione: *Python* e *C++*, in base alle proprie conoscenze e/o esigenze. In questa tesi si è scelto di utilizzare il primo per ogni fase implementativa, come si può notare al Capitolo 4. Nelle prossime sottosezioni si discute invece dei concetti alla base di *ROS 2*.

1.2.1 Nodi

Un *nodo* rappresenta una singola unità logica che svolge una specifica funzione all'interno del processo di esecuzione di un robot. Ogni nodo in *ROS* può inviare e ricevere dati da altri nodi tramite *topic*, servizi e azioni [5]. I nodi possono:

- pubblicare su *topic* specifici per fornire dati ad altri nodi;
- sottoscriversi per ottenere dati da altri nodi;
- agire come un client di servizio per far sì che un altro nodo esegua un calcolo per loro, o come un server di servizio per fornire funzionalità ad altri nodi;
- analogamente al servizio, agire come action client/server, per calcoli di lunga durata.

- fornire *parametri* per modificare il comportamento durante il runtime.

1.2.2 Topic

Un *topic* rappresenta il mezzo di comunicazione tramite cui i nodi si scambiano i *messaggi* e risulta particolarmente indicato per flussi di dati continui, come quelli dei sensori, lo stato del robot e altre informazioni.

I *messaggi* rappresentano il mezzo con cui un nodo invia dati sulla rete ad altri nodi, senza alcuna risposta prevista. Questi sono descritti e definiti nei file `.msg` nella directory `msg/` di un pacchetto *ROS* e sono composti da campi e costanti.

Un *topic* è dunque un sistema di pubblicazione/sottoscrizione *fortemente tipizzato*. Entrando nello specifico, nel sistema dei topic sono presenti produttori di dati (detti “*publishers*”) e consumatori di dati (detti “*subscribers*”). Queste due entità sanno come mettersi in contatto tra loro proprio grazie al concetto stesso di *topic*, identificato da un nome comune per far sì che le entità possano “vedersi”. Quando i dati vengono pubblicati nel *topic* da uno qualsiasi dei *publishers*, tutti i *subscribers* attivi nel sistema riceveranno i dati.

Questa struttura ricorda molto il concetto di *bus* nell’informatica, poiché assomiglia proprio a quello presente nei dispositivi hardware. Un’altra peculiarità di *ROS* è che, di fatto, tutto è *anonimo*: questo implica che, quando un *subscriber* riceve un dato, generalmente non sa quale *publisher* lo abbia originariamente inviato, anche se lo si può scoprire. Il vantaggio di questa architettura è che entrambi i soggetti possono essere scambiati a piacimento, senza influenzare il resto del sistema.

1.2.3 Parametri e Launch file

I *parametri* consistono in una coppia `<chiave:valore>` e sono associati ai singoli nodi, in modo da configurarli all’avvio e durante il runtime, senza quindi dover modificare continuamente il codice. I valori iniziali dei parametri possono essere impostati durante l’esecuzione del nodo tramite argomenti da riga di comando, file `YAML`, o quando si esegue il nodo, con l’uso di *launch file*.

Un sistema sviluppato con *ROS 2* è tipicamente costituito da molti nodi che eseguono processi diversi, eventualmente anche su macchine differenti. In particolare, i grandi progetti di robotica coinvolgono spesso più nodi interconnessi, ognuno dei quali può avere a sua volta numerosi parametri. Sebbene sia possibile eseguire ciascuno di questi nodi separatamente, l'operazione si complica rapidamente.

Il sistema dei *launch file* in *ROS 2* ha lo scopo di automatizzare l'esecuzione di più nodi attraverso l'uso di un unico comando: `ros2 launch <nome_launch_file.py>`. La configurazione del file include quali programmi eseguire e quali argomenti passare; in pratica, questi file – scritti in *Python*, *YAML* o *XML* – consentono di configurare e avviare contemporaneamente più eseguibili contenenti dei nodi di *ROS*. Durante lo sviluppo di questa tesi, essi si sono rivelati fondamentali per l'operazione di “*tuning*” degli algoritmi realizzati, come si può notare nella sottosezione 4.6).

1.3 Simulatore: F1TENTH Gym

Il simulatore utilizzato per svolgere tutto il lavoro di tesi è *F1TENTH Gym*, l'ambiente ufficiale ideato a fini ricerca, in quanto è necessaria una simulazione asincrona e realistica dei veicoli, con la possibilità di avere anche più istanze di veicoli (*agenti*) nello stesso ambiente.

Il motore fisico del simulatore favorisce la presenza di programmi estremamente paralleli e, soprattutto, permette di avere un'esecuzione rapida. Si specifica che *F1TENTH Gym* è il simulatore alla base di *F1TENTH Gym ROS*, un bridge di comunicazione con *ROS 2* per *F1TENTH Gym*, che dunque viene trasformato in una simulazione vera e propria per *ROS 2*, visualizzandola nell'applicativo *RViz*.

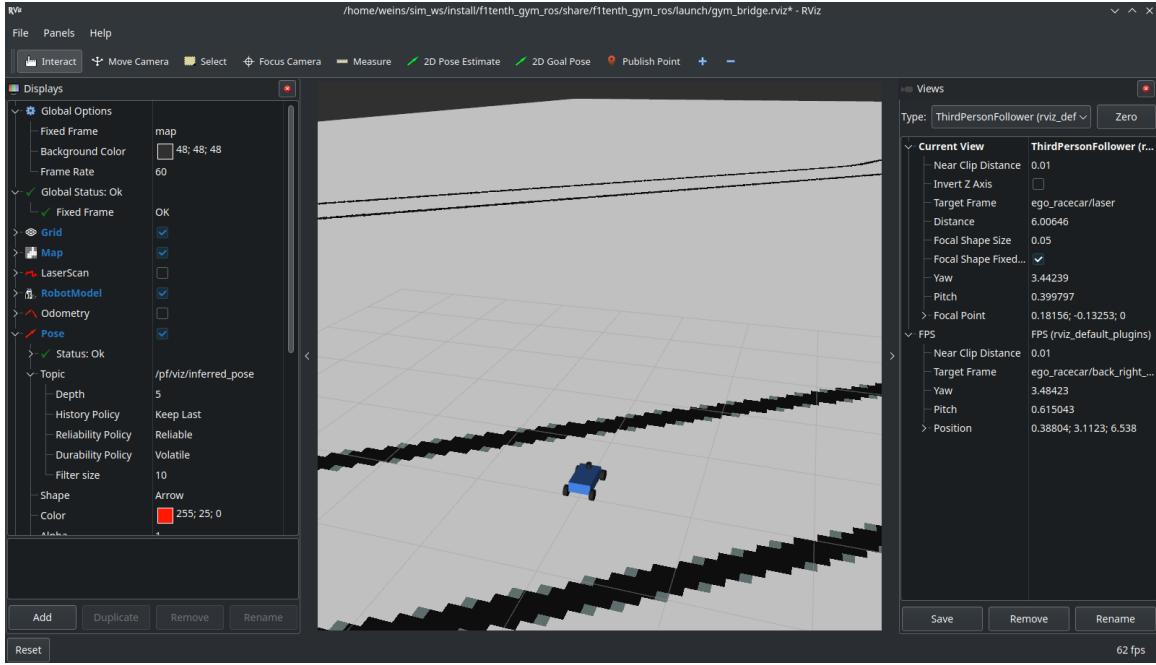


Figura 3: *F1TENTH Gym ROS* con visualizzazione di *RViz*.

Si specifica che per questo lavoro si è sempre usato un unico agente. Per questa casistica, i *topic* pubblicati dalla simulazione sono:

1. `/scan` – La scansione laser dell’agente;
2. `/ego_racecar/odom` – L’*odometria* dell’agente rappresenta l’uso dei dati provenienti dai sensori di movimento per stimare il cambiamento nel tempo della posizione, dell’orientamento e della velocità.
3. `/map` – La mappa dell’ambiente.

Mentre quelli sottoscritti dalla simulazione comprendono:

1. `/drive` – Il comando di guida tramite messaggi `AckermannDriveStamped`.
2. `/initialpose` – Permette di reimpostare la posizione dell’agente tramite lo strumento `2D Pose Estimate` di *RViz*.

Anche il nodo `teleop`, relativo al controllo dell’agente con la tastiera, viene incluso come parte della dipendenza del simulatore.

1.4 Progettazione dei veicoli a guida autonoma

La progettazione di veicoli a guida autonoma viene suddivisa in tre macro-aree che vengono eseguite una dopo l'altra in modo circolare.

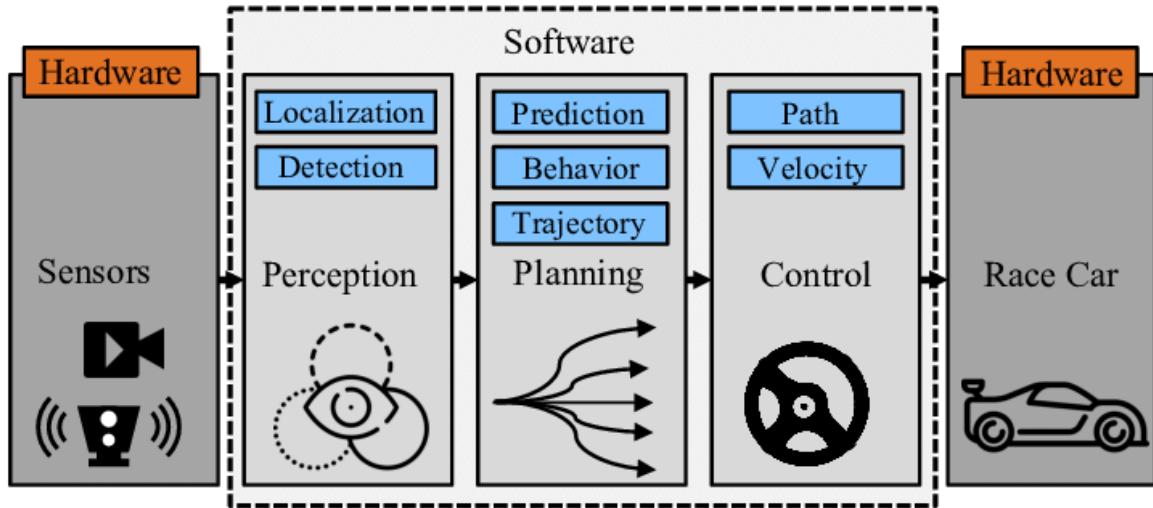


Figura 4: Pipeline della guida autonoma [6].

Ogni area produce degli output utili che vengono passati in input alla successiva [6]. Dunque, in ordine di esecuzione, il problema è suddiviso in:

1. *Perception* – A partire dai sensori presenti nel robot, costruisce un modello dell'ambiente che lo circonda e lo localizza al suo interno.
2. *Planning* – Grazie all'output della fase precedente, è in grado di determinare il suo stato ed esplorare quelli successivi.
3. *Control* – Detto anche *Actuator*, si occupa di generare una sequenza di input di sterzata e accelerazione per portare il veicolo allo stato successivo, che è stato generato come output della fase precedente.

Questo ciclo prende anche il nome di *Sense-Plan-Act* e, generalmente, per garantire una buona performance, lo si vuole completare tra le 20 e 50 volte al secondo.

1.4.1 Perception

La parte di *percezione* comprende la raccolta dei dati dei sensori e la successiva elaborazione per avere un *modello* dell’ambiente, proprio come gli occhi di un guidatore umano. Di seguito gli obiettivi e le attività svolte da questo modulo applicativo:

- *Mapping* – Si vuole rendere il robot abbastanza intelligente da percepire l’ambiente circostante, così da rilevare ostacoli, altri robot mobili e tutto ciò che lo circonda.
- *Localization* – Si vuole capire la posizione del robot all’interno dell’ambiente. Fondamentalmente, si hanno due tipi di strumenti per ottenere tale conoscenza: le informazioni sull’odometria, fornite dall’*Inertial Measurement Unit (IMU)*, e le informazioni sulle osservazioni, fornite dal *LiDAR*.
- *Vision* – Si possono sfruttare metodi classici di visione, come *detection* con *OpenCV*, *Visual SLAM*, o metodi basati sul *Deep Learning*, come *Object Detection*. Tuttavia, si specifica che questa tematica non è stata trattata in questo lavoro di tesi.

Quando si parla del movimento di un robot mobile bisogna innanzitutto dire che è sempre presente l’incertezza nelle osservazioni; per rappresentare esplicitamente l’incertezza si usa la probabilità, che è alla base dei metodi più comuni per eseguire la localizzazione [7], che sono:

1. *Bayes Filter*: viene solitamente utilizzato esclusivamente per i casi discreti, cioè con valore finito per lo stato. L’idea alla base è quella di stimare una densità di probabilità nello spazio degli stati condizionata dai dati (percettivi e odometrici).
2. *Kalman Filter*: algoritmo ricorsivo che utilizza una combinazione di predizioni e misurazioni per stimare lo stato di un sistema nel tempo. Utilizza un metodo basato sui campioni per rappresentare una determinata distribuzione.
3. *Particle Filter*: chiamato anche *Adaptive Monte Carlo*, si basa su un campionamento pesato e casuale. Si tratta di un *Bayes Filter* ricorsivo, simile al *Kalman*

Filter, ma ne “rilassa” alcune assunzioni. Ogni particella è un’ipotesi di posizione e, invece di campionare ogni volta, ciascun campione viene propagato con il modello di movimento per ottenere i campioni nell’iterazione successiva.

Per il problema di localizzazione, inizialmente si potrebbe pensare di usare l’odometria attraverso la tecnica del *dead reckoning*. Si inizia cioè da una posizione nota, integrando le misurazioni del controllo e del movimento per stimare la posizione corrente. Disporre di strumenti o attrezzi migliori sarà senza dubbio di aiuto, tuttavia, esiste una limitazione fondamentale all’uso della sola odometria: si tratta di un approccio di stima *open loop*, dunque non esiste un meccanismo di *feedback* che ci consenta di correggere gli errori nella misurazione. Pertanto, in caso di rumore nelle misurazioni, col passare del tempo si accumulerà l’errore. Un altro problema correlato è lo slittamento delle ruote, detto *odometry drift*.

La soluzione a queste problematiche è data da *Adaptive Monte Carlo Localization (AMCL)*, un algoritmo di localizzazione basato sul *Particle Filter* [8].

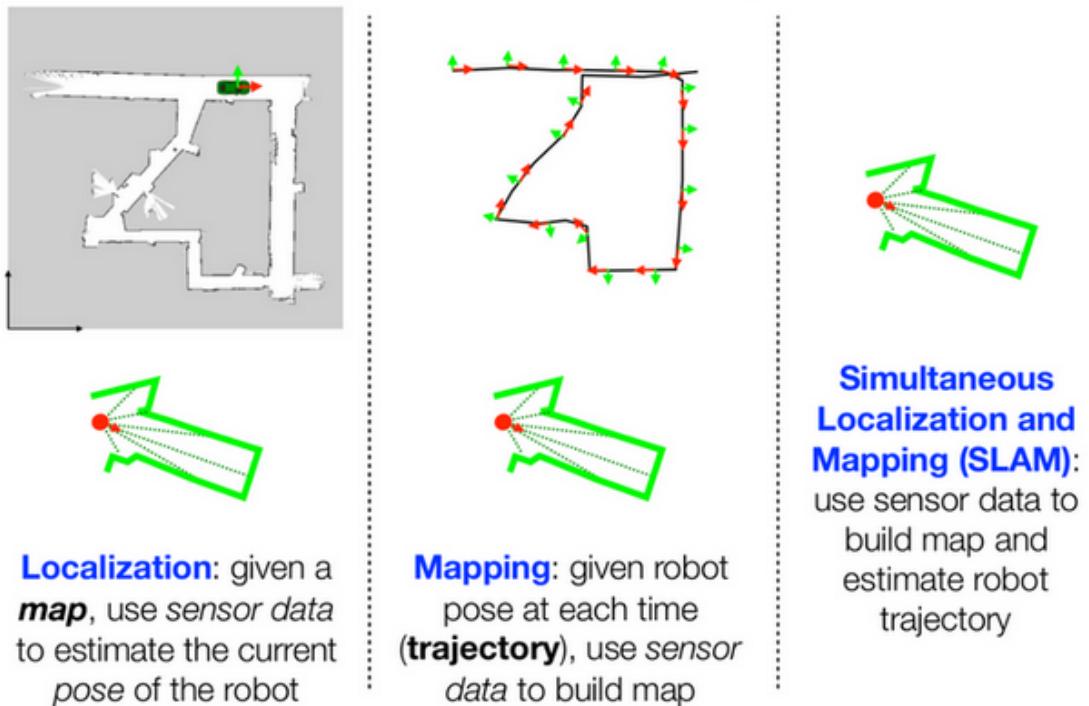


Figura 5: Differenza tra localization, mapping e SLAM [9].

Problema – Per determinare la posizione del robot è necessaria una mappa dell’ambiente, ma si ha prima bisogno della posizione del robot per costruire la mappa stessa.

Soluzione – *Simultaneous Localization and Mapping (SLAM)* è una tecnica che consente ai robot di eseguire simultaneamente la *localizzazione* e il *mapping* di un ambiente sconosciuto. Questo processo continua per diversi istanti di tempo, aggiornando la mappa a ogni passo [9].

1.4.2 Planning

La *pianificazione* consiste nel trovare un percorso ottimo grazie al quale il robot possa spostarsi progressivamente dal punto di partenza a quello di “*goal*”, evitando eventuali ostacoli presenti nel cammino.

Generalmente, si suddivide quest’area di sviluppo in una gerarchia di tecniche di *planning* [10]:

- *Global Planner* – Si tratta di una vista d’insieme del problema per determinare il percorso ottimo;
- *Local Planner* – È una vista che riguarda solo la porzione dell’ambiente vicina al veicolo, sfruttata per individuare le traiettorie possibili per l’auto;
- *Behavioral Planner* – Consiste in una pianificazione delle strategie da adottare in diverse situazioni, selezionando la migliore secondo una specifica funzione di costo.

Ad alto livello, le tecniche di pianificazione di un percorso si dividono in due categorie:

1. *basate sul campionamento*: *Probabilistic Road Maps (PRM)* e *Rapidly-Exploring Random Tree (RRT*)* [11];
2. *basate sulla ricerca*: *A** e *A* Lattice Planning*.

1.4.3 Control

Il modulo di *controllo* si occupa dei segnali da inviare agli *attuatori* delle ruote (es. angolo di sterzata, freni, ecc.) e del motore (velocità, accelerazione) per portare il veicolo al suo stato successivo.

In linea generale, le domande a cui si vuole rispondere sono [12]:

- Come seguire un percorso prestabilito?
- Come correggere gli errori di attuazione?
- Come guidare il veicolo il più velocemente possibile?

Si tratta di un campo di studi così vasto che viene racchiuso nella disciplina dell'*ingegneria del controllo* [13], che comprende la modellazione di una vasta gamma di sistemi dinamici e la progettazione di controllori che garantiscano che questi sistemi rispettino il comportamento atteso. In particolare, se ne distinguono due tipologie differenti:

1. *Controller Open Loop*: sono adatti per sistemi senza vincoli di dinamica o stabilità, poiché non catturano lo stato del sistema. Inoltre, attuano direttamente la risposta desiderata, senza la continua osservazione dello stato del sistema;
2. *Controller Closed Loop Feedback*: rispetto ai precedenti, sono indicati per sistemi dinamici e calcolano continuamente un valore di errore $e(t)$, come differenza tra la risposta di output desiderata, detta *setpoint*, e una variabile di processo misurata che rappresenta l'output reale.

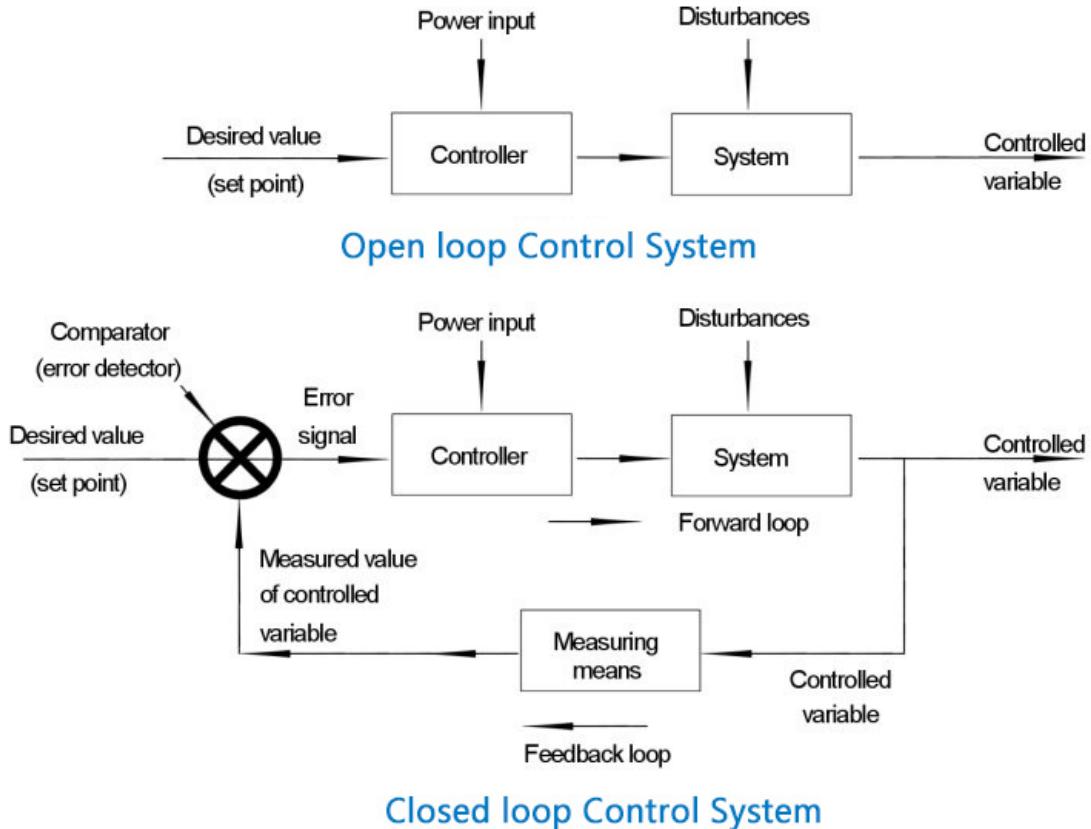


Figura 6: Confronto tra i controller *Open Loop* e *Closed Loop Feedback* [14].

L’obiettivo principale di un controller è quello di applicare automaticamente una correzione precisa e reattiva a una certa funzione di controllo, ad esempio, per mantenere la velocità desiderata, con un ritardo e un superamento minimi, andando ad aumentare gradualmente la potenza del motore. L’evoluzione del controllo dei veicoli autonomi può essere suddivisa in tre principali fasi di sviluppo [15]:

1. Ispirati dalla comunità della robotica, i primi algoritmi di controllo sono stati ispirati da concetti geometrici come la pianificazione di un movimento circolare o l’allineamento del volante verso un percorso *target*. Questi approcci evidenziano buone prestazioni per velocità medio-basse.
2. All’aumentare delle velocità e delle accelerazioni dei veicoli a guida autonoma,

all'interno della comunità di ricerca si sono diffusi più metodi di analisi approfonditi, provenienti dalla teoria del controllo e dai sistemi dinamici. Permettono di progettare *controllori state-feedback* e di considerarne più effetti dettagliati, come la dinamica dell'imbardata del veicolo e la dinamica dell'attuatore dello sterzo.

3. Si è in seguito sviluppata una specifica branca del controllo, detta *controllo ottimale*, che vuole controllare un sistema in modo da minimizzare o massimizzare un determinato obiettivo. È particolarmente indicata per applicazioni complesse: infatti, fa parte di questa classe di problemi il *Model Predictive Control*, il tema principale studiato in questa tesi, che verrà discusso nel capitolo 3.

Gli algoritmi di controllo trattati nel contesto di *F1TENTH* comprendono *PID Controller* e *Pure Pursuit*, discussi in dettaglio nel Capitolo 2. Invece, a partire dal Capitolo 3, si approfondisce *Model Predictive Control (MPC)*.

Capitolo 2

Metodi reattivi per il controllo

Questo capitolo si occupa di descrivere la navigazione con *metodi reattivi*, in quanto rappresenta una strategia largamente utilizzata per il controllo di un agente autonomo. Essa suggerisce di prendere tutte le decisioni di controllo attraverso un'elaborazione dei dati recenti dei sensori. Infatti, sono algoritmi senza un vero e proprio piano che cercano di guidare il veicolo evitando gli ostacoli o seguendo un percorso prestabilito – spesso generato dal modulo di *planning*, come spiegato nella sottosezione 1.4.2 – fornendo input di sterzata e velocità all'auto per effettuare manovre fluide.

2.1 PID Controller

Una strategia semplice per far navigare l'auto consiste nel seguire una linea (*line follow*), un muro di un corridoio o il cordolo di un circuito (*wall follow*), attraverso un *controllore* che fornisce input di *sterzata* e *accelerazione* all'auto autonoma.

L'algoritmo più diffuso che fornisce questi input è il controllore *PID*, acronimo di *Proportional, Integral e Derivative*. Si tratta di un controller molto popolare e versatile, il cui punto di forza risiede nella sua semplicità e nei risultati che ottiene. Questo tipo di controller viene usato nelle applicazioni che richiedono un controllo di tipo *closed-loop feedback*. Di seguito si descrive come il controllore calcola l'angolo di sterzata e la velocità a cui guidare per poter eseguire una manovra *fluida*.

2.1.1 Struttura

Il controllore acquisisce in ingresso un valore e lo confronta con un valore di riferimento, detto *setpoint*. La differenza tra questi due valori corrisponde al segnale di *errore*, che viene usato per determinare il valore di uscita del controllore.

Il controller *PID* regola l'uscita in base ai seguenti parametri:

1. il valore del segnale di errore (azione proporzionale);
2. la somma degli errori accumulati nel tempo fino all'istante corrente (azione integrale);
3. la velocità di variazione del segnale di errore (azione derivata).

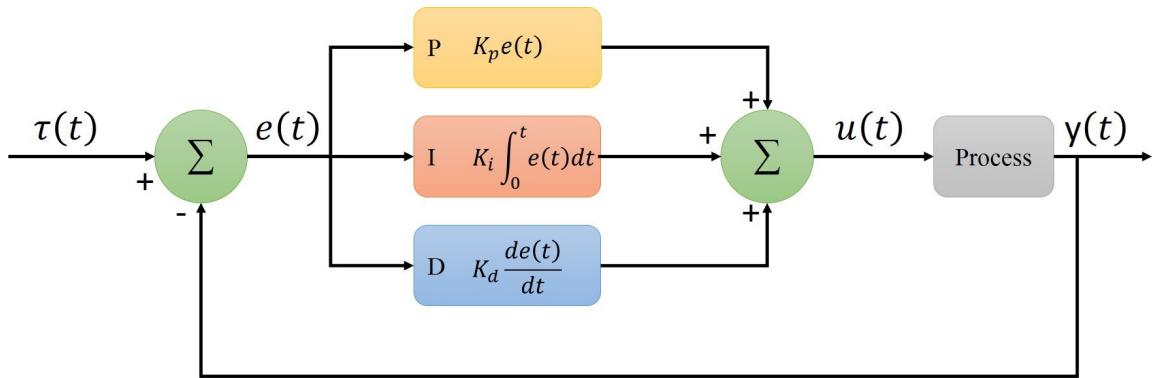


Figura 7: Schema a blocchi del PID controller. Qui $\tau(t)$ è il setpoint, $u(t)$ è l'output di controllo, mentre $y(t)$ è il valore della variabile di processo [16].

Nello specifico, le componenti di controllo del *PID* sono:

- **Proportional Controller** – l'angolo di sterzata dell'auto deve essere proporzionale alla distanza della stessa dal percorso stabilito, detta *crosstrack error*. Maggiore è l'errore, maggiore deve essere la correzione per correggerlo.
- **Derivative Controller** – lo svantaggio principale del **Proportional** è che provoca continue oscillazioni attorno alla linea centrale e aumenta l'*overshoot*, cioè il superamento del *setpoint*. Quello che si desidera, invece, è che l'errore diminuisca nel tempo, riducendo di conseguenza anche l'azione correttiva.

Prevenendo quest'effetto, si applica una correzione anticipata, che deve essere proporzionale alla velocità con cui l'errore si sta riducendo. Una previsione triviale è data dall'attuazione di un **gain** basato sull'errore, che porterà il robot a controsterzare e ad avvicinarsi progressivamente alla linea centrale. Dunque, l'azione derivata prevede il comportamento del sistema e ne migliora la stabilità.

- **Integral Controller** – è possibile includere anche un termine di **gain** integrale, che è proporzionale all'errore accumulato a partire da un tempo di riferimento t . L'azione integrale, dunque, tiene conto del *crosstrack error* accumulato nel tempo, correggendo gli errori che non sono stati risolti nei passi temporali precedenti.

L'output di controllo $u(t)$ di quest'algoritmo, vale a dire ciò che viene controllato, è l'angolo di sterzata δ a cui si vuole che l'auto guidi. Si può così indicare di seguito l'equazione standard di *PID*, dove $e(t)$ è l'errore dalla traiettoria desiderata:

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} + K_i \int_0^t e(t) dt$$

Le *costanti* K_p , K_i e K_d determinano con quale peso contribuisce ciascuna delle tre componenti. Nell'implementazione di PID, l'azione *integrale* non è sempre necessaria. Infatti, a seguito di una grande variazione del *setpoint*, il termine integrale può accumulare un errore superiore al valore massimo della variabile di controllo, portando il sistema in *overshooting*, con conseguente aumento dell'errore. La soluzione a questo problema consiste nel disabilitare l'azione integrale [13].

Ricapitolando, il **Proportional** svolge la maggior parte del lavoro, il **Derivative** agisce sul primo per ridurne l'*overshoot* e l'**Integral**, se presente, riduce l'errore che resta dopo che il primo ha svolto il suo lavoro.

2.1.2 Tuning

Il *PID* è un esempio di metodo di controllo libero da qualsiasi tipo di *modello*: non si modella il veicolo e quindi, per ottimizzare i valori dei **gain** di controllo K_p , K_i e K_d , bisogna fare affidamento sull'esperienza e sull'analisi. Un ingegnere del controllo

deve sapere cosa si può fare, cosa non è sicuro e quali bias sono presenti nel sistema, tuttavia molto dipende anche dalle caratteristiche prestazionali desiderate. Esistono sia approcci manuali che euristiche, come quella di *Ziegler-Nichols*. Nell'ambito di F1TENTH, il metodo più comune è quello manuale, soprattutto in ambiente simulativo, come in questo caso. Con un divario inferiore tra la simulazione e la realtà (*Sim2Real*), il codice sviluppato nel simulatore potrebbe essere utilizzato direttamente sull'auto reale.

2.2 Pure Pursuit

Il metodo del *Pure Pursuit* è uno degli approcci più diffusi nella guida autonoma per il problema affrontato in questa tesi: il *path tracking* per auto autonome da corsa. Al veicolo viene assegnata una sequenza di posizioni da seguire, dette *waypoint*, indicate nel frame di riferimento del veicolo e l'obiettivo dell'algoritmo consiste nel seguirli.

Il controllo del movimento di un veicolo *olonomico* è più semplice: essi possono muoversi in qualsiasi direzione, a prescindere dall'orientamento del veicolo. Tuttavia, in questo contesto si ha un veicolo non oeconomico, cioè l'automobile di *F1TENTH*, che deve riposizionarsi e cambiare direzione per spostarsi in un certo punto. Ciò, infatti, complica il problema del *path tracking*.

Per affrontare questa sfida, *Pure Pursuit* calcola geometricamente la curvatura di un arco che collega la posizione dell'asse posteriore dell'auto a un punto di *goal* sul percorso da seguire. Quest'ultimo è sempre il *waypoint* più vicino al veicolo e ad almeno una data distanza di *lookahead*, che va dalla posizione corrente dell'auto al percorso desiderato. In questo contesto, *Pure Pursuit* rappresenta un controller laterale che ignora le forze dinamiche che agiscono sui veicoli e presuppone che le ruote mantengano la condizione antiscivolo (*no-slip*).

L'algoritmo è un **P Controller** dell'angolo di sterzata δ che agisce sull'errore di tracking, secondo un certo **gain** e *lookahead*. Per determinare l'angolo, si calcola come segue la curvatura dell'arco γ :

$$r = \frac{L^2}{2|y|} \rightarrow \gamma = \frac{1}{r} = \frac{2|y|}{L^2} \quad (1)$$

Il *lookahead* è sempre soggetto a operazioni di *tuning*: con un valore basso si ha un tracciamento più accurato, ma più oscillazioni; invece se più alto, si osservano meno oscillazioni, ottenendo però un peggioramento del tracciamento. Infine, viene attuata la velocità che è indicata per ogni waypoint del percorso.

2.3 Vantaggi e svantaggi

I *vantaggi* dei metodi reattivi per il controllo comprendono:

1. la possibilità di applicarlo a robot con risorse hardware limitate e a basso prezzo;
2. il fatto di poter far navigare un robot in sicurezza in ambienti completamente sconosciuti e contenenti ostacoli non prevedibili.

Invece, tra gli *svantaggi* di questi metodi si ha che:

1. possono occasionalmente far seguire traiettorie *impraticabili* per i robot, dunque non è facile adattarli a sistemi più complessi;
2. i **gain** di controllo devono essere regolati manualmente e per diverso tempo, comportando un'intensa attività di *tuning*;
3. un'eventuale suddivisione del problema in due diversi controllori (*longitudinale* e *laterale*) ignorerebbe l'accoppiamento tra le due dimensioni. Ciò comporterebbe che, per esempio, il controllore *PID*, usato solo come controllore longitudinale, non sappia nulla del controllore laterale (*Pure Pursuit*). In certe situazioni, come in una curva, ciò può generare problemi nella sterzata;
4. non offrono alcuna possibilità di gestione dei vincoli;
5. considerano solo lo stato attuale e, dunque, ignorano le decisioni future. In questo modo, non considerano le conseguenze che possono avere le decisioni attuali sulle successive.

Come anticipato nella sottosezione 1.4.3, *Model Predictive Control* costituisce una soluzione più avanzata che permette di superare i limiti dei metodi reattivi, poiché ha alla base il *controllo ottimale*.

Capitolo 3

Model Predictive Control

In questo capitolo si approfondisce il lavoro principale di questa tesi che, come anticipato in precedenza, è stato lo studio e l'implementazione di un sistema di *path tracking* che permettesse al veicolo di seguire un percorso – composto da una serie di *waypoint* equidistanti – mediante un tracking preciso della traiettoria di riferimento, eventualmente tentando anche di ridurre i tempi di completamento di un giro su specifici circuiti da corsa.

Per raggiungere questo scopo, si è scelto di realizzare *Model Predictive Control (MPC)*, un controller all'avanguardia che, generalmente, viene progettato per automatizzare un sistema di *sensori* e *attuatori*, come sistemi energetici industriali o robotici. Tra l'altro, anche Ayrton Senna, uno dei più grandi piloti di Formula 1 di tutti i tempi, andava ad applicare una strategia simile a *MPC*, attraverso la *tecnica dell'acceleratore*. In pratica, Senna aveva un modello mentale del comportamento del turbocompressore e, in modo predittivo, cercava di massimizzare l'accelerazione in uscita da una curva [17].

Le metodologie basate su *MPC* rappresentano la soluzione principale alla base di diversi controller per veicoli da corsa autonomi che sono stati implementati su veicoli reali [6]. Più precisamente, si tratta di una strategia di controllo che sfrutta un determinato *modello della dinamica dell'ambiente* e, rispettando una serie di *vincoli*, è in grado di controllare un dato processo. Il modello deve essere adeguato al sistema o processo da controllare poiché uno dei vincoli da soddisfare è proprio il modello.

Pertanto, esso viene utilizzato per prevedere il comportamento futuro con l'*obiettivo* di trovare l'input di controllo *ottimale*, che deve essere applicato per un certo *orizzonte temporale* scelto.

Nel contesto di questa tesi e di *F1TENTH*, lo scopo di *MPC* è generare, per N passi futuri, controlli di input validi che guidino il veicolo il più vicino possibile alla traiettoria di riferimento, garantendo al contempo prestazioni stabili anche a velocità sostenute. Come accennato, *MPC* è in grado di gestire i vincoli del sistema, cosa che non è possibile con un metodo reattivo come il *PID*. Quest'ultimo, infatti, è un sistema di tipo *SISO* (*Single Input, Single Output*), in grado di gestire solo un singolo input $e(t)$ e generare un singolo output $u(t)$. Un esempio può essere l'avere come input l'errore dell'angolo di sterzata e come output il calcolo del nuovo angolo di sterzata.

Quindi, tra i due, conviene adottare *MPC*, anche perché le dinamiche dell'ambiente possono cambiare nel tempo e *MPC* offre un modo semplice per stimare tali dinamiche e adeguarsi correttamente. Al contrario, *PID* non è adatto per i sistemi la cui dinamica cambia frequentemente, perché potrebbe essere necessario modificare continuamente i valori dei `gain` di controllo per ottenere il comportamento desiderato, rischiando comunque di imbattersi in un *overshoot*. Inoltre, a differenza di *PID*, *MPC* è dotato di un vettore di *input di controllo* e di un vettore di *stato*, dunque è un sistema di tipo *MIMO* (*Multi-Input, Multi-Output*), che consente cioè a più input di variare l'output del sistema.

Un'alternativa di tipo *MIMO* a *MPC* è *LQR* (*Linear Quadratic Regulator*), una versione senza vincoli e più semplice di *MPC* [17], che però ha i seguenti limiti:

1. non è in grado di gestire dei vincoli;
2. può generare input di controllo impraticabili per il robot, come un angolo di sterzata di $\frac{\pi}{2}$.

MPC risulta quindi essere un controllore che permette di soddisfare:

1. vincoli di *sicurezza*, come velocità, accelerazione e limiti del percorso;
2. vincoli *fisici*, come mantenere una traiettoria possibile per la dinamica dell'auto.

3.1 Struttura

Dopo aver analizzato le problematiche di altri metodi e aver introdotto *MPC*, si vuole comprendere come controllare all’ottimo un sistema. Infatti, il primo passo per impostare *MPC* è definire il problema di controllo ottimale da risolvere, che è caratterizzato dalle seguenti componenti:

1. il *modello* della dinamica del sistema, per prevedere il comportamento futuro.
Esempio: dinamica del veicolo;
2. la *funzione obiettivo* da minimizzare. *Esempio*: *cross-track error* e/o tempo di un giro;
3. i *vincoli* da rispettare all’interno del sistema. *Esempio*: limiti della velocità, della sterzata, ecc.

Pertanto, si tratta di un problema di *ottimizzazione convessa* con vincoli e ciò che si vuole risolvere è la sequenza $[x_0 \dots x_N \quad u_0 \dots u_N]^T$, composta da:

1. azioni di controllo, indicate come u ;
2. stati previsti, indicati come x .

Come anticipato, *MPC* si prefigge di controllare un sistema risolvendo iterativamente un problema di ottimizzazione che sia in grado di prendere in considerazione un modello e dei vincoli fisici. Il controllore ha il compito di minimizzare una certa funzione obiettivo, detta anche di costo, attraverso la simulazione degli stati futuri, a partire da quello attuale e dalle variabili di controllo, secondo uno specifico modello del sistema. Questi stati sono delimitati dai vincoli dati a *MPC* in modo che non vengano raggiunti stati indesiderati e/o non praticabili. Le variabili di controllo possono quindi essere scelte in modo che gli stati generati soddisfino la funzione di costo.

Il punto nel futuro fino al quale simula viene determinato da un *orizzonte* di previsione finito: per un orizzonte più ampio il tracciamento risultante sarebbe certamente più fluido, ma anche più pesante a livello computazionale. Le variabili di controllo

calcolate possono quindi essere utilizzate come attuazione per il sistema controllato in quell'istante di tempo [18].

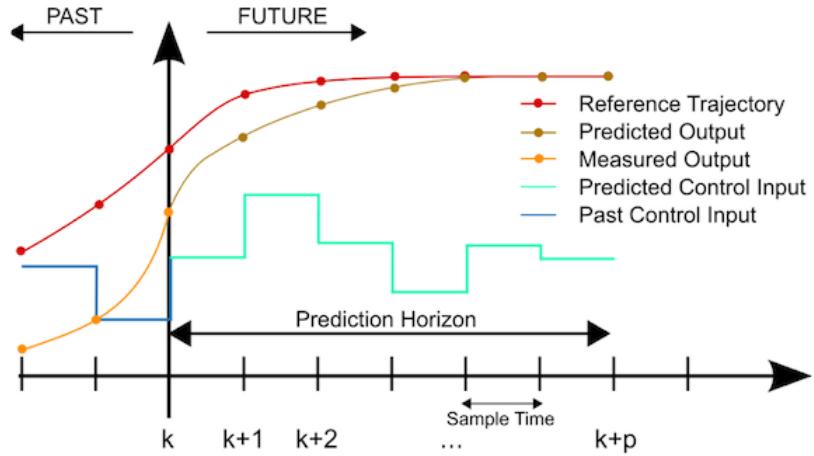


Figura 8: Diagramma che mostra il lavoro svolto da *MPC*, differenziando tra *passato* e *futuro* (con un certo orizzonte) [19].

Nella Fig. 8 si può osservare che la linea azzurra in basso è l'attuazione, un input di controllo discreto, mentre in alto si trovano sia la traiettoria di riferimento che quella futura prevista. Il controllore prenderà in considerazione lo stato rilevato del sistema e ottimizzerà una sequenza di controllo ideale, in base alla funzione obiettivo definita.

Dunque, il problema di ottimizzazione con vincoli alla base di *MPC* si può definire come la minimizzazione del costo di una traiettoria che sia praticabile per la dinamica del sistema e che non violi i vincoli imposti. Si riporta di seguito un *setup* generale

di questo problema:

$$U_t^*(x(t)) := \min_{U_t} \sum_{k=0}^{N-1} q(x_{t+k}, u_{t+k})$$

subject to $x_t = x(t)$	misure
$x_{t+k+1} = Ax_{t+k} + Bu_{t+k}$	modello del sistema
$x_{t+k} \in \mathcal{X}$	vincoli dello stato
$u_{t+k} \in \mathcal{U}$	vincoli degli input
$U_t = \{u_t, u_{t+1}, \dots, u_{t+N-1}\}$	variabili di ottimizzazione

Si specifica che con $k = 0$ si intende il time-step corrente, mentre con $k = N - 1$ si fa riferimento al time-step antecedente alla fine della finestra dell'orizzonte temporale. Si discuterà maggiormente del tema nelle sezioni successive, entrando nello specifico di ciascuna componente del problema di ottimizzazione.

3.1.1 Receding Horizon Control

Alla base del funzionamento di *Model Predictive Control* c'è il concetto di orizzonte finito, definito in letteratura come *Receding Horizon Control*. Questo indica che, dopo che è stata applicata l'attuazione, viene calcolata una nuova serie di variabili di controllo per il nuovo stato del veicolo.

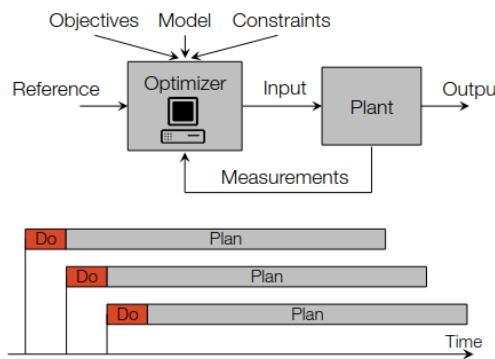


Figura 9: Struttura di *MPC con Receding Horizon Control*, in cui si pianifica per una certa finestra di tempo [17].

Come si può osservare dalla Fig. 9, ciò viene reiterato in modo da aiutare il controllore nella gestione di comportamenti inaspettati. Il vantaggio principale è rappresentato dal fatto che, applicando continuamente gli input di controllo ottimali, il sistema può essere portato verso un riferimento desiderato – come una traiettoria ottima da seguire – che viene raggiunto quando la funzione obiettivo risulta minimizzata.

Per progettare un controllore con queste caratteristiche, il sistema deve essere di tipo *closed loop*, cioè deve includere un meccanismo di feedback in cui le variabili di controllo u vengono calcolate in base all'errore tra gli stati di riferimento x_{ref} e gli stati simulati x . Inoltre, l'orizzonte finito consente di avere un problema di ottimizzazione trattabile computazionalmente, in contrapposizione al controllo dell'orizzonte infinito (con un sistema “*open loop*”), che risulterebbe impraticabile poiché considererebbe tutto l'ambiente – il circuito in questo contesto – e non più solo una parte di esso.

Più precisamente, i passi dell'*MPC* con orizzonte finito sono i seguenti [20]:

1. Ottenere lo stato corrente $x(t)$;
2. Calcolare, per una determinata finestra di pianificazione temporale N , la sequenza di input di controllo ottimale: $U_t^* = \{u_t^*, \dots, u_{t+N-1}^*\}$;
3. Applicare solo il primo input di controllo u_t^* ;
4. Ripetere per ripianificare.

L'intuizione fondamentale è che, invece di applicare l'intera sequenza di input calcolata, si applichi solo il primo input ottimale per poter istantaneamente misurare lo stato risultante del veicolo. In questo modo si riescono a considerare preventivamente gli errori poiché viene introdotto un feedback grazie alla successiva ripianificazione.

È presente un trade-off nella scelta dell'orizzonte di pianificazione N : idealmente, se ne vorrebbe avere uno breve che consideri pochi istanti temporali, risultando anche più leggero; tuttavia, con un orizzonte troppo ridotto la traiettoria pianificata potrebbe non tenere conto del comportamento futuro del sistema, portando il controllore a intraprendere azioni di controllo miopi – ad esempio, mantenere un'accelerazione elevata in prossimità di una curva che non è stata rilevata. D'altro canto, un orizzonte

che guarda troppo in avanti renderebbe impegnativo il raggiungimento di soluzioni precise, sia per ciò che riguarda la funzione di costo che per il rispetto dei vincoli.

3.2 Modello

Per poter formulare un problema di controllo ottimale, è necessario disporre di un *modello della dinamica del sistema*, poiché esso è in grado di prevedere l'evoluzione dello stato del veicolo per una sequenza di input data in ingresso. Questo modello può essere rappresentato dalla seguente equazione:

$$\dot{x}(t) = f(x(t), u(t))$$

dove $x(t) \in \mathbb{R}^n$ è lo stato del sistema, $u(t) \in \mathbb{R}^m$ rappresenta gli input di controllo e $f : \mathbb{R}^n \times \mathbb{R}^m \rightarrow \mathbb{R}^n$ è la funzione di sistema, cioè il modello della dinamica [17]. Non è possibile risolvere l'equazione analiticamente in forma chiusa, pertanto deve essere risolta numericamente. Passando dal continuo al discreto, dunque si ottiene:

$$x_{t+1} = f(x_t, u_t)$$

con $t \in \mathbb{N}$. Si approfondirà il processo di discretizzazione nella sezione 3.6.1.

Nello specifico, il modello della dinamica f prende lo stato x_t – ad esempio, corrispondente alla posizione e alla velocità dell'auto – e gli input di controllo – ad esempio, corrispondenti all'angolo di sterzata e all'accelerazione – e li mappa allo stato dell'istante di tempo successivo. In altre parole, data una sequenza di input e, soprattutto, un modello si può prevedere l'evoluzione dello stato nel futuro.

Il modello della dinamica del sistema appena descritto è necessario affinché il controllore sia in grado di simulare correttamente il cambiamento nello stato del veicolo. Più precisamente, questo modello è caratterizzato da una serie di equazioni differenziali che descrivono questo cambiamento in ciascuna variabile di stato. Esistono diverse semplificazioni del modello, alcune più dettagliate e altre meno, ma tutte comprendono sempre il comportamento generale del veicolo. Questi “rilassamenti” del modello sono utili perché è difficile modellare precisamente un veicolo reale. Oltre

a ciò, al crescere della complessità del modello cresce rapidamente anche quella del controllore.

3.2.1 Kinematic Bicycle Model

La modellazione del comportamento dinamico del veicolo è una parte cruciale nel campo delle auto da corsa a guida autonoma. Questi modelli vengono utilizzati sia negli ambienti di simulazione che in lavori riguardanti la progettazione del controllo della traiettoria, proprio come fatto in questo lavoro di tesi.

Lo stato dell'arte al momento offre molti modelli diversi tra loro, come il *Kinematic Bicycle Model*, *Dynamic Bicycle Model*, il *Double Track Model* o il *Full Vehicle Model*. Più il modello della dinamica del veicolo è complesso, più aumentano i parametri necessari per la sua realizzazione [6].

Il modello scelto per questa tesi è il primo, anche detto *Kinematic Single-Track Model* e *Front Wheel Steering Model* in letteratura. Esso può essere definito come un modello semplificato che cattura molto bene il movimento del veicolo. Nel concreto, il modello “rilassa” il concetto di veicolo attraverso l'unione delle due ruote anteriori in un'unica ruota; stesso discorso per quelle posteriori.

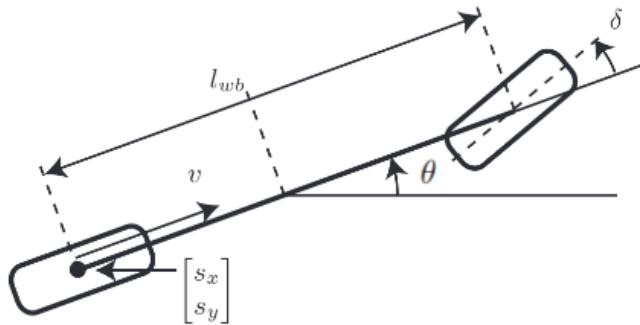


Figura 10: *Kinematic Bicycle Model* con punto di riferimento al centro dell'asse posteriore [21].

Come visibile in Fig. 10, questa configurazione a due ruote permette di paragonarlo proprio a una *bicicletta*; il modello infatti copre solo il movimento su un piano, ignorando di conseguenza i movimenti di rollio e beccheggio. Inoltre, può differire

leggermente a seconda che le variabili di input siano la velocità e la sterzata, o l'accelerazione e la sterzata (come in questo caso). L'orientamento della ruota anteriore può essere controllato rispetto alla direzione del veicolo, mentre non è possibile lo stesso ragionamento per la ruota posteriore; dunque, di fatto, si considera e si segue solo la ruota anteriore [21].

Per formalizzare il “*Kinematic Bicycle Model*”, bisogna innanzitutto definire lo spazio degli stati come $x = [s_x, s_y, \theta, v]$, dove s_x e s_y sono le coordinate del veicolo poste al centro dell'asse posteriore, θ è l'orientamento del veicolo e v la velocità; dopodiché, si definisce il vettore degli input come $u = [a, \delta]$, dove a rappresenta l'accelerazione e δ l'angolo di sterzata delle ruote anteriori. Le equazioni differenziali che descrivono il cambiamento di stato per il modello sono:

$$\dot{s}_x = v \cos(\theta), \quad \dot{s}_y = v \sin(\theta), \quad \dot{v} = a, \quad \dot{\theta} = \frac{v \tan(\delta)}{l_{wb}}$$

Dove l_{wb} indica la lunghezza del passo (*wheelbase*) del veicolo da una ruota all'altra e corrisponde a $l_{\text{front}} + l_{\text{rear}}$. Le equazioni differenziali indicate sopra si possono esprimere in forma sintetica come:

$$\dot{x} = f(x, u) = A'x + B'u$$

Dove A' e B' sono matrici di sistema continue e, come anticipato, il sistema verrà poi discretizzato (sezione 3.6.1) e linearizzato (sezione 3.6.2).

Il modello cinematico ignora l'effetto dello slittamento degli pneumatici e quindi non riflette in certi casi la dinamica effettiva [22]. In genere, è particolarmente indicato in condizioni di guida normali, tuttavia un modello più complesso, come quello dinamico, non necessariamente migliorerebbe le prestazioni poiché potrebbe richiedere maggior tempo per la risoluzione. Tuttavia, in alcuni casi il modello cinematico può risultare adeguato anche in condizioni di guida più estreme; in caso contrario, il modello necessita di una buona rapidità per riuscire a compensare i segnali di controllo non ottimali.

3.3 Funzione Obiettivo

Il compito della *funzione obiettivo* è quello di assegnare un *costo* J a una traiettoria data, in modo da valutare la bontà di quest'ultima per lo scopo prefissato. Considerando T come il numero di passi dell'orizzonte temporale, tale costo può essere espresso come:

$$\begin{aligned} J = & \sum_{t=0}^T c_t(x_t, u_t) = \\ & Q_f (x_T - x_{T,ref})^2 + \\ & Q \sum_{t=0}^{T-1} (x_t - x_{t,ref})^2 + \\ & R \sum_{t=0}^T u_t^2 + R_d \sum_{t=0}^{T-1} (u_{t+1} - u_t)^2 \end{aligned}$$

L'obiettivo consiste nel minimizzare il costo J . Si noti che J è quadratica ed è divisa in due parti [17]:

1. *Costo dell'errore di stato*, pesato dalla matrice di peso Q e Q_f ;
2. *Costo dell'attuazione*, pesato dalle matrici di peso R e R_d .

Nello specifico, la funzione di costo penalizza sia la differenza tra gli stati calcolati e gli stati di riferimento, sia i grandi input di controllo, affinché lo stato desiderato venga raggiunto con un input di controllo minimo. Si vogliono infatti minimizzare tre sotto-obiettivi:

1. La deviazione del veicolo dalla traiettoria di riferimento: $\forall t \in T, x_t - x_{t,ref}$. La deviazione dello stato finale è pesata da Q_f , mentre le altre da Q .
2. Il fattore di influenza degli input di controllo, pesato da R .
3. La differenza tra un input di controllo e il successivo, pesata da R_d .

Le matrici di pesi R , R_d e Q giocano un ruolo fondamentale nel determinare il comportamento di *MPC*. Esse infatti influenzano il modo in cui l'algoritmo penalizza,

rispettivamente, l’attuazione degli input di controllo, i cambiamenti negli input di controllo e la deviazione dalla traiettoria di riferimento. In *MPC*, la matrice Q è sempre semi-definita positiva; ciò garantisce che il costo associato agli errori di stato non sia mai negativo, il che è importante per mantenere la convessità del problema di ottimizzazione e per garantire che il costo totale non sia appunto negativo. Invece, la matrice R è sempre definita positiva, garantendo quindi che ci sia sempre un costo associato all’applicazione degli input di controllo, evitando così soluzioni banali senza penalità in cui gli input di controllo potrebbero essere troppo grandi o piccoli.

Più precisamente, valori più alti per R indicano che l’algoritmo sarà più riluttante a utilizzare input di controllo elevati; la matrice R_d può aiutare a ottenere un controllo più fluido; infine, le matrici Q e Q_f riducono la deviazione dalla traiettoria di riferimento da seguire.

3.4 Vincoli

Come già anticipato a inizio capitolo, uno dei vantaggi più significativi di *MPC* è proprio la possibilità di gestire i *vincoli* del sistema trattato. Ciò permette di riflettere in modo più preciso la realtà rispetto a un metodo reattivo come il *PID Controller*, trattato nel Capitolo 2. I vincoli del problema di ottimizzazione sono i seguenti:

- lo *stato iniziale* in programma per l’orizzonte corrente deve corrispondere allo stato corrente del veicolo, cioè $x_0 = x(0)$. Quindi, lo stato iniziale x_0 è impostato come lo stato corrente del veicolo, mentre il tempo iniziale corrisponde al tempo nello stato di riferimento più vicino allo stato corrente del veicolo.
- gli *stati futuri* del veicolo devono seguire il modello della dinamica del veicolo *linearizzato*.
- gli *input generati* non devono superare i limiti fisici del veicolo.

3.4.1 Stati futuri e dinamica

Un modello che prevede stati x futuri può mutare nel tempo. Alcuni risolutori moderni possono gestire dinamiche non lineari – ad esempio *Casadi*, che non è stato però oggetto di questo lavoro – tuttavia spesso si cerca di linearizzare la dinamica del sistema per effettuare il calcolo di *MPC* in modo più veloce. La dinamica linearizzata è solitamente nella seguente forma:

$$x_{t+1} = Ax_t + Bu_t \quad \forall x, u \in [x_0 \dots x_N \quad u_0 \dots u_N]^T$$

Questo vincolo su due stati consecutivi, vale a dire x_t e x_{t+1} , garantisce che la sequenza degli stati possa essere esclusivamente una *traiettoria realistica* che rispetta la dinamica del sistema.

3.4.2 Stato e input

I vincoli per stato e input possono essere formulati semplicemente come un limite inferiore e superiore su una delle variabili di stato o di controllo. Alcuni esempi di vincoli di disegualanza sono dati da:

- *Limiti dell'attuatore* – in riferimento all'angolo di sterzata, alla velocità, ecc.
- *Confini del tracciato* – può essere interpretato come una regione delimitata da un insieme di linee.

Dunque, i vincoli del sistema codificano i domini degli stati $x_t \in \mathcal{X}$ e degli input $u_t \in \mathcal{U}$ consentiti. Ciascun vincolo di disegualanza può essere scritto come $Ax \leq b$, dove per ogni riga di A , b corrisponde a un vincolo applicato.

3.5 Algoritmo

Si mostra di seguito lo pseudocodice del funzionamento generale dell'algoritmo del *Model Predictive Control* che, ricevendo in ingresso la funzione di costo J , il modello

della dinamica del sistema f , l'orizzonte degli eventi T e l'input di controllo iniziale \hat{u}_0 , risolve all'ottimo il problema.

Algorithm 1 MPC: Model Predictive Control

Input: Obiettivo J , modello della dinamica f , orizzonte temporale T , input di controllo iniziale \hat{u}_0 .

```

1:  $u_t \leftarrow \hat{u}_0$ 
2: while True do
3:    $x_0 \leftarrow \text{GET\_CURRENT\_STATE}()$ 
4:    $\triangleright$  Il risolutore viene settato in warm-start, così da sfruttare la soluzione
      dell'iterazione precedente.  $\triangleleft$ 
5:    $u_t \leftarrow \text{SOLVE\_LINEAR\_MPC}(J, f, x_0, T, u_t)$ 
6:    $u \leftarrow \text{GET\_FIRST}(u_t)$ 
7:    $\text{EXECUTE\_CONTROL}(u)$ 
```

Prima di farlo però, come si può vedere nell'algoritmo 1, l'input di controllo viene inizializzato con quello iniziale. In seguito, dentro il ciclo viene rilevato lo stato corrente x_0 , che viene poi passato in input al risolutore, insieme agli altri componenti principali del problema di ottimizzazione.

Una volta fatto ciò, si ottiene la sequenza di controlli ottimale. Si noti però come l'algoritmo sfrutta e attua solo il primo di questi input, che corrisponde al primo passo temporale della finestra dell'orizzonte data.

Dato che i controlli tra un passo temporale e il successivo variano di poco, si sfrutta il concetto di *warm start*: si riutilizza l'input di controllo dell'iterazione precedente in modo da trovare la soluzione del passo successivo in modo più rapido. In questo modo l'intero algoritmo trarrà beneficio da questa caratteristica, richiedendo tempi di calcolo inferiori.

3.6 Quadratic Programming

Il problema di ottimizzazione che caratterizza *MPC* non ha soluzioni analitiche. Per poterlo risolvere, infatti, bisogna formulare un problema di *Quadratic Programming (QP)*. Si tratta di un processo per risolvere problemi di ottimizzazione matematica che coinvolgono *funzioni quadratiche*. L'obiettivo di questo metodo è quello di trovare

un vettore z che minimizzi (o massimizzi) una funzione quadratica soggetta a *vincoli lineari* sulle variabili. Infatti, devono essere mantenute alcune condizioni di frontiera (*boundary*) durante l'ottimizzazione, come *lower bound* e *upper bound*. In realtà, il termine “*programmazione*” in questo contesto si riferisce solo a una procedura formale usata per risolvere problemi matematici.

Avendo $c, z \in \mathbb{R}^n$, $A_{m \times n}$, $Q_{m \times n}$, una forma standard comune di *programmazione quadratica* è la seguente:

$$\begin{aligned} \min \quad & f(z) = \frac{1}{2} z^T H z + c^T z \\ \text{subject to} \quad & l_b \leq A_c z \leq u_b \end{aligned}$$

Un'altra caratteristica di *QP* è che è *convessa*, cioè, esiste un *unico ottimo globale*. Si ricorda che una funzione a valori reali è detta *convessa* se il segmento di linea che congiunge due punti qualsiasi del suo grafico giace sopra il grafico stesso (o coincide con una sua parte). In altri termini, se si traccia una linea tra due punti qualsiasi, la funzione sarà sotto la linea. Inoltre, un problema di questo tipo può essere risolto efficientemente in tempo reale (*online*). Sono disponibili molti solutori di *QP*, tra i quali:

- *OSQP*, CVXGen, QuadProg, ecc.
- *Casadi* per l'ottimizzazione non convessa.
- MPT3 per la progettazione e l'analisi lineare di *MPC*.

Il risolutore più consigliato da F1TENTH è *OSQP* (*Operator Splitting Quadratic Program*) [23].

Manca ancora un aspetto importante per poter formulare il problema come uno di *QP*: bisogna prima *discretizzare* il sistema dinamico e *linearizzarlo* attorno a un certo punto.

3.6.1 Discretizzazione

Per effettuare quest'operazione viene utilizzata – ma non implementato – il metodo “*Forward Euler*” poiché è il più semplice. Dovrebbero funzionare anche altri metodi, come *RK4/6*. Per la discretizzazione viene utilizzato il tempo di campionamento dt , che può essere scelto come parametro da ottimizzare. Si può quindi esprimere l'equazione del sistema come:

$$x_{t+1} = x_t + f(x_t, u_t) dt \quad (2)$$

3.6.2 Linearizzazione

La maggior parte dei sistemi fisici sono non lineari, il che comporta che le loro equazioni sono più complesse da risolvere. Invece, i sistemi lineari sono più semplici, sono generalmente ben compresi e possono essere analizzati più agevolmente. I sistemi non lineari possono essere ben approssimati da un sistema lineare in un piccolo intorno di un punto presente nello spazio degli *stati*. L'idea è quella che se il sistema viene controllato per rimanere vicino a un punto operativo – in una condizione di lavoro stabile – si può poi sostituire il sistema non lineare con uno linearizzato attorno a quel punto.

Per la *linearizzazione* si sfrutta l'espansione di Taylor del primo ordine della funzione a due variabili in (2) attorno a certi \bar{x} e \bar{u} , che porta ad avere:

$$\begin{aligned} x_{t+1} &= x_t + (f(\bar{x}_t, \bar{u}_t) + f'_z(\bar{x}_t, \bar{u}_t)(x_t - \bar{x}_t) + f'_u(\bar{x}_t, \bar{u}_t)(u_t - \bar{u}_t))dt \\ &\rightarrow Ax_t + Bu_t + C \end{aligned}$$

Nel contesto di *MPC*, i sistemi lineari consentono di tradurre la dinamica come vincoli lineari secondo la formulazione di *MPC*, dunque essi permettono di riscrivere *MPC* come un *QP* standard. Il problema di ottimizzazione risultante è così di rapida risoluzione.

3.6.3 Conversione

Per passare dalla forma generale del problema di *MPC* alla formulazione in programmazione quadratica si effettua principalmente un lavoro di gestione di indici e di matrici. Per quanto riguarda la funzione di costo $J = \frac{1}{2}z^T Hz + c^T z$, vale:

$$H = \text{diag}(Q, Q, \dots, Q_N, R, \dots, R)$$

$$c = \begin{bmatrix} -Qx_r \\ \vdots \\ -Q_Nx_r \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad z = \begin{bmatrix} x_0 \\ \vdots \\ x_{N+1} \\ u_0 \\ \vdots \\ u_N \end{bmatrix}$$

H è una matrice diagonale in cui ci sono $N - 1$ matrici di penalità Q e N matrici R uguali, dove ognuna di esse fa riferimento a un *timestep* preciso dell'orizzonte di MPC. Si distingue solo Q_N perché l'ultima potrebbe avere pesi diversi. Invece c è un vettore che considera gli stati di riferimento – dove $-Qx_r$ è estratto da $(x_k - x_{ref})^T Q (x_k - x_{ref})$.

Bisogna poi trasformare i vincoli della dinamica e quelli di stato e input nella forma compatta $l_b \leq A_c z \leq u_b$. Per la matrice A_c , diversamente, si ha:

$$l_b = \begin{bmatrix} -x_0 \\ 0 \\ \vdots \\ 0 \\ x_{\min} \\ \vdots \\ x_{\min} \\ u_{\min} \\ \vdots \\ u_{\min} \end{bmatrix} \quad A_c = \left[\begin{array}{ccccccc} -I & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ A & -I & 0 & \dots & 0 & B & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ I & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & 0 & I \end{array} \right] \quad u_b = \begin{bmatrix} -x_0 \\ 0 \\ \vdots \\ 0 \\ x_{\max} \\ \vdots \\ x_{\max} \\ u_{\max} \\ \vdots \\ u_{\max} \end{bmatrix}$$

In riferimento alle righe di A_c , applicando la forma compatta si ha:

Riga 1: $-x_0 \leq -Ix_0 \leq -x_0 \rightarrow x(0) = x_0$	stato iniziale
Riga 2: $x_1 = Ax_0 + Bx_0$	aggiornamento dello stato
Riga 3: $x_{min} \leq x \leq x_{max}$	limiti del stato
Riga 4: $u_{min} \leq u \leq u_{max}$	limiti del controllo

3.7 Vantaggi e svantaggi

Per concludere, si raggruppano di seguito i punti di forza di *MPC*:

1. È un controller ad alte prestazioni che gestisce sistematicamente i vincoli.
2. Considera preventivamente le decisioni future.
3. Ha una formulazione flessibile che può incorporare sotto-obiettivi aggiuntivi.
4. Può essere formulato anche per la dinamica dei sistemi non lineari.
5. È in grado di gestire anche dinamiche variabili nel tempo.

La sfida principale quando si implementa *MPC* è il carico computazionale causato dalla risoluzione in tempo reale del problema di ottimizzazione, ripetuta per ogni passo temporale. I principali fattori che influenzano il tempo di calcolo sono il numero di *stati*, il numero di *input* e l'orizzonte di previsione [24]. Ci sono poi anche altre sfide da affrontare per raggiungere buone performance, poiché l'algoritmo non riesce sempre a garantire la stabilità, la praticabilità e la robustezza.

Capitolo 4

Implementazione

Questo capitolo tratta in modo approfondito i dettagli implementativi che hanno caratterizzato lo sviluppo di *Model Predictive Control*, come la scelta dei circuiti di Formula 1 che sono stati utilizzati per testare il sistema e per effettuare successivamente le analisi, il risolutore utilizzato, la realizzazione del problema di ottimizzazione attraverso il linguaggio *Python*, la fase di *tuning* e altri aspetti tecnici.

4.1 Circuiti

Per mettere alla prova l'algoritmo di *Model Predictive Control* sono stati scelti due circuiti di Formula 1 in scala 1:10, ovvero *Spa-Francorchamps* e *Monza*. In realtà, prima di far correre il veicolo autonomo su un classico circuito, lo si è testato su una mappa rettangolare più semplice, corrispondente a una versione modificata di “*Levine Hallway 2nd Floor*”, la sede del Dipartimento di Informatica della *University of Pennsylvania*, la cui versione originale viene fornita di default all'interno del simulatore. I circuiti usati per questa tesi sono visibili nella Fig. 11 e sono stati scaricati dal GitHub di *F1TENTH* [25], in quanto offre diverse mappe di circuiti di F1 in scala 1:10 che possono essere utilizzati all'interno del simulatore.

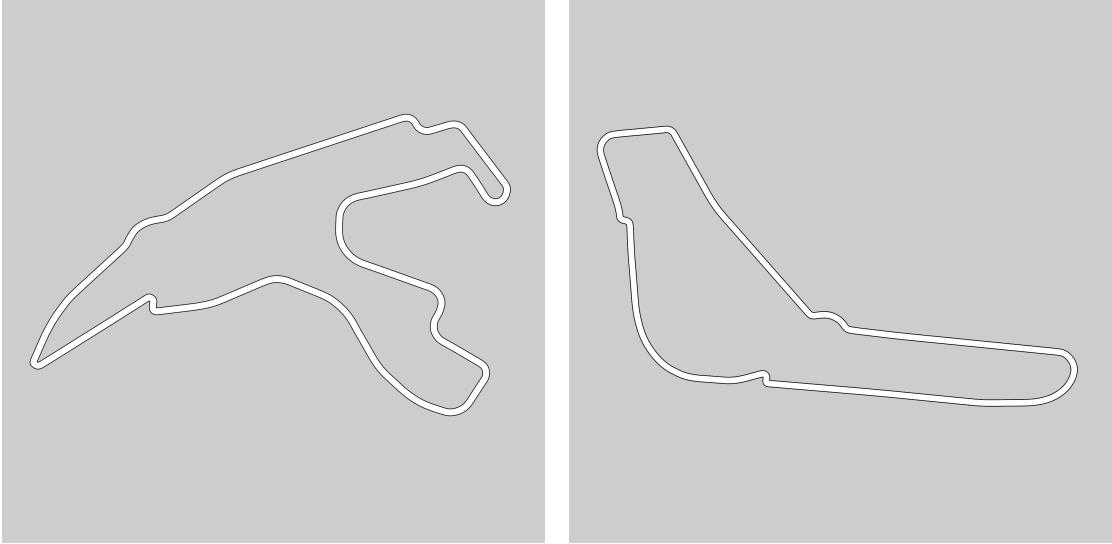


Figura 11: Mappe usate per testare *MPC* [25].

Ognuna di queste mappe ha comportato un intenso lavoro di adattamento delle funzionalità e, in particolare, dei valori di costanti e pesi che caratterizzano il *Model Predictive Control*. Questa parte dello sviluppo verrà trattata dal punto di vista pratico e implementativo nella sezione 4.6.

4.2 Configurazione

Prima di poter risolvere il problema di *Model Predictive Control*, è stato necessario organizzare i parametri di configurazione che lo definiscono. Essi sono stati riuniti in un'apposita classe, utilizzata per definire le caratteristiche ben definite del modello, le proprietà fisiche del veicolo, i pesi delle matrici di costo e altri valori necessari per l'ottimizzazione. Nello specifico, questi parametri comprendono:

1. Dimensioni del modello.
 - **NXK:** Dimensione del vettore di stato del veicolo. Gli stati includono posizione \mathbf{x} , y , velocità \mathbf{v} e orientamento θ .
 - **NU:** Dimensione del vettore di input di controllo. Gli input includono solo la sterzata δ , interpretata anche come velocità, e l'accelerazione \mathbf{a} .

- \mathbf{TK} : Orizzonte temporale finito per la previsione dell'algoritmo.
 - \mathbf{DTK} : Passo temporale per la previsione, espresso in secondi.
 - \mathbf{dlk} : Passo di distanza per la previsione, sempre di 0.03 metri.
2. Matrici di penalità, anticipate nel Capitolo 3. I valori utilizzati per queste matrici saranno approfonditi nella sezione 4.6.
- \mathbf{Rk} : Matrice di costo per gli input di controllo. Penalizza l'uso degli input di controllo, come l'accelerazione e la velocità di sterzata.
 - \mathbf{Rdk} : Matrice di costo per la differenza degli input di controllo. Penalizza i cambiamenti negli input di controllo tra i passi temporali.
 - \mathbf{Qk} : Matrice di costo per l'errore di stato. Penalizza la deviazione dello stato del veicolo – in ordine \mathbf{x} , \mathbf{y} , \mathbf{v} e θ – dalla traiettoria di riferimento.
 - \mathbf{Qfk} : Come \mathbf{Qk} , ma per lo stato finale del veicolo.
3. Caratteristiche del veicolo in simulazione.
- **LENGTH**: Lunghezza del veicolo, di 0.58 metri.
 - **WIDTH**: Larghezza del veicolo, di 0.31 metri.
 - **WB**: Passo del veicolo, di 0.33 metri.
 - **MIN_STEER**: Angolo di sterzata minimo, che vale -0.4189 radianti.
 - **MAX_STEER**: Angolo di sterzata massimo, che vale 0.4189 radianti.
 - **MAX_DSTEER**: Velocità massima di sterzata, in radianti/secondo.
 - **MAX_SPEED**: Velocità massima del veicolo, di 15 m/s.
 - **MIN_SPEED**: Velocità minima del veicolo, di 0 m/s (per la fase di avvio).
 - **MAX_ACCEL**: Accelerazione massima del veicolo, generalmente di 3 m/s².

Questa configurazione viene utilizzata per inizializzare e per risolvere il problema di ottimizzazione di *MPC*. I parametri definiti influenzano direttamente il comportamento del veicolo in simulazione, il modo in cui viene seguita la traiettoria di riferimento e la risposta del sistema agli input di controllo.

4.3 Risolutore

Una volta completata la fase di configurazione, si può definire e impostare il problema di ottimizzazione. Ciò è stato effettuato attraverso l'utilizzo di **CVXPY**, selezionando **OSQP** come risolutore.

CVXPY è un linguaggio di modellazione open-source per problemi di *ottimizzazione convessa* che viene distribuito come pacchetto di *Python*. Esso consente di esprimere un certo problema in un modo naturale che segue la matematica, piuttosto che nella forma standard restrittiva tipicamente richiesta dai risolutori, poiché esso trasforma direttamente il problema in tale forma. È contraddistinto dai seguenti concetti:

- la classe immutabile **Problem** per definire il problema (con obiettivo e vincoli);
- la classe **Variable**, che può rappresentare scalari, vettori o matrici;
- la classe **Parameter**, che rappresenta espressioni costanti il cui valore potrebbe essere specificato dopo la creazione del problema;
- il metodo **solve**, che accetta argomenti facoltativi che consentono di modificare il modo in cui **CVXPY** risolve il problema. Come effetto collaterale, valorizza gli attributi **status** e **value** sull'oggetto relativo al problema.

Questi componenti vengono usati per la *creazione* del problema di ottimizzazione quadratica – che verrà risolto a ogni iterazione per determinare gli input di controllo ottimali – al fine di inizializzare:

1. le variabili di stato e controllo, rispettivamente per gli orizzonti $TK+1$ e TK ;
2. i parametri per lo stato iniziale e per la traiettoria di riferimento da seguire, con quest'ultimo per l'orizzonte $TK+1$;
3. le matrici di costo a blocchi diagonali, ripetute per ogni passo temporale TK .

4.4 Obiettivi

La prima implementazione cruciale è stata la formulazione e la creazione del problema di controllo ottimale con orizzonte finito. Dunque, è stata definita la *funzione obiettivo*, suddivisa in tre parti distinte, poi sommate tra loro. I dettagli teorici di queste parti sono stati trattati ampiamente nella sezione 3.3; invece, di seguito si riportano delle porzioni di codice per ognuna di esse.

Dettagli implementativi

- `cvxpy.quad_form`: calcola la forma quadratica di un vettore o di una matrice;
- `cvxpy.vec`: converte una matrice in un vettore colonna, concatenando le colonne della matrice;
- `cvxpy.diff`: calcola la differenza tra le colonne consecutive di una matrice, quindi, in questo caso, viene fatta tra input di controllo in passi temporali consecutivi.

Parte 1 La prima parte dell’obiettivo richiede di gestire l’influenza degli input di controllo u , da moltiplicare dunque con la matrice di penalità R .

```

1 # Inizializza R come matrice di costo a blocco diagonale
2 # R = [R, R, ..., R] (NU*T, NU*T)
3 R_bl = block_diag(tuple([self.config.Rk] * self.config.TK))
4 obj += quad_form(vec(self.uk), R_bl)
```

La funzione `cvxpy.quad_form` calcola la forma quadratica ($u_k^T R u_k$) per ogni k e somma i risultati per tutti i passi temporali T , indicati anche come N . Questa operazione corrisponde alla seguente formula matematica:

$$\text{obj}_1 = \sum_{k=0}^{N-1} u_k^T R u_k$$

Parte 2 La seconda parte richiede di gestire la deviazione del veicolo dalla traiettoria di riferimento, pesata dalla matrice di penalità Q , includendo il passo temporale finale, che viene pesato però da Q_N (cioè Q_f). L'obiettivo viene espresso come:

$$\text{obj}_2 = (x_N - x_{ref}^N)^T Q_N (x_N - x_{ref}^N) + \sum_{k=0}^{N-1} (x_k - x_{ref}^k)^T Q_k (x_k - x_{ref}^k)$$

```

1 # Inizializza Q come matrice di costo a blocco diagonale
2 # Q = [Q, Q, ..., Qf] (NX*T, NX*T)
3 Q_b1 = [self.config.Qk] * (self.config.TK)
4 Q_b1.append(self.config.Qfk)
5 Q_b1 = block_diag(tuple(Q_b1))
6 obj += quad_form(vec(self.xk - self.ref_traj_k), Q_b1)

```

Parte 3 La terza parte richiede di gestire la differenza tra un input di controllo e il successivo, pesata dalla matrice di penalità R_d . L'obiettivo in forma quadratica risultante è:

$$\text{obj}_3 = \sum_{k=0}^{N-2} (u_{k+1} - u_k)^T R_d (u_{k+1} - u_k)$$

```

1 # Inizializza Rd come matrice di costo a blocco diagonale
2 # Rd = [Rd, ..., Rd] (NU*(T-1), NU*(T-1))
3 Rd_b1 = block_diag(tuple([self.config.Rdk] * (self.config.TK-1)))
4 obj += quad_form(vec(diff(self.uk, axis=1)), Rd_b1)

```

Questa parte dell'obiettivo dunque calcola la forma quadratica delle differenze negli input di controllo, penalizzando così grandi cambiamenti negli input di controllo.

4.5 Vincoli

Una volta scritto l'obiettivo, sono stati definiti i *vincoli* di *MPC*. La parte di preparazione delle matrici del modello della dinamica (A , B e C) è stata già data dalla comunità di F1TENTH: le matrici in questione descrivono l'evoluzione dello stato

del veicolo nel tempo in base agli input di controllo; inoltre, vengono convertite in parametri di CVXPY, al fine di poter essere utilizzate nei vincoli della dinamica del veicolo. Infatti, a questa fase è seguita l'implementazione effettiva dei vincoli, i quali si possono suddividere in tre parti principali:

Vincoli dinamici Questi vincoli rappresentano la dinamica del veicolo. La formula corrisponde a:

$$x_{k+1} = A_k x_k + B_k u_k + C_k$$

```

1 flatten_prev_xk = vec(self.xk[:, :-1])
2 flatten_next_xk = vec(self.xk[:, 1:])
3 c1 = flatten_next_xk == \
4     self.Ak_ @ flatten_prev_xk + \
5     self.Bk_ @ vec(self.uk) + \
6     self.Ck_
7 constraints.append(c1)

```

Vincolo sulla sterzata Questo vincolo limita la variazione della velocità di sterzata per evitare cambiamenti troppo rapidi. La formula corrisponde a:

$$-\delta_{\max} \leq \delta_{k+1} - \delta_k \leq \delta_{\max}$$

```

1 dsteering = cvxpy.diff(self.uk[1, :])
2 c2_lower = -self.config.MAX_DSTEER * self.config.DTK <= dsteering
3 c2_upper = dsteering <= self.config.MAX_DSTEER * self.config.DTK
4 constraints.append(c2_lower)
5 constraints.append(c2_upper)

```

Vincoli su stati e input Questi vincoli impongono limiti sugli stati e sugli input del veicolo. È compreso anche quello sullo stato iniziale, la cui formula corrisponde a: $x_0 = x_{0k}$.

```

1 # Stato iniziale
2 c3 = self.xk[:, 0] == self.x0k
3 constraints.append(c3)
4 # Stato
5 speed = self.xk[2, :]
6 c4_lower = self.config.MIN_SPEED <= speed
7 c4_upper = speed <= self.config.MAX_SPEED
8 constraints.append(c4_lower)
9 constraints.append(c4_upper)
10 # Input di controllo
11 steering = self.uk[1, :]
12 c5_lower = self.config.MIN_STEER <= steering
13 c5_upper = steering <= self.config.MAX_STEER
14 constraints.append(c5_lower)
15 constraints.append(c5_upper)
16 acc = self.uk[0, :]
17 c6 = acc <= self.config.MAX_ACCEL
18 constraints.append(c6)

```

4.6 Tuning con le penalità

Il *tuning* delle matrici di penalità è un passaggio cruciale per ottimizzare le prestazioni di *MPC*; infatti, le matrici influenzano direttamente il comportamento del veicolo, poiché bilanciano il *trade-off* tra la minimizzazione dell'*errore di stato* e la limitazione dell'effetto degli *input di controllo*. La scelta dei pesi è stata effettuata in simulazione mediante un processo iterativo di test empirici. Ciò è stato fatto al fine di valutare come le variazioni nei pesi influenzassero le prestazioni del veicolo, così da trovare anche i valori più critici per regolare di conseguenza. Come anticipato, i pesi sono stati scelti per mantenere un equilibrio tra la reattività del veicolo e la stabilità del controllo. A partire dai pesi determinati in questa fase, si costruiscono le matrici diagonali a blocchi *R_block*, *Rd_block* e *Q_block*, descritte nella sezione 4.4.

Per la configurazione e la gestione del nodo *ROS* di *MPC* sono stati usati dei *launch file*, ciascuno per ogni profilo di guida determinato per quest'algoritmo. Ogni

profilo di guida, detto anche configurazione, è configurato con parametri specifici per adattarsi alle diverse esigenze operative. Essi consistono in:

- **Safe** – Rappresenta un profilo adatto per una guida più sicura e stabile, in quanto parte a velocità basse per poi salire con accelerazione elevata. Considerando un orizzonte temporale non molto alto, applica penalità elevate sull'errore di stato per garantire che il veicolo segua accuratamente la traiettoria di riferimento.
- **Fast** – Questo profilo è bilanciato tra velocità e stabilità, con valori di velocità massima di sterzata inferiore. Inoltre, a seconda del circuito, ha penalità sugli input di controllo leggermente superiori rispetto al profilo **Safe**, in modo da favorire una risposta più rapida del sistema.
- **High Performance** – Si tratta di un profilo configurato per una guida estremamente reattiva e rapida. Esso presenta un orizzonte temporale più ampio e penalità generalmente inferiori sull'errore di stato e sugli input. Ciò permette al veicolo di rispondere rapidamente ai cambiamenti nella traiettoria.

4.6.1 Spa

Nella Tabella 1 sono indicati i valori scelti per ciascuna configurazione relativa al circuito di *Spa*. I valori presenti riguardano i parametri del modello, quelli del veicolo e i pesi delle matrici di penalità.

Parametro	Safe	Fast	High Performance
TK	5	5	7
DTK	0.03 s	0.03 s	0.03 s
MAX_SPEED	15 m/s	15 m/s	15 m/s
MAX_ACCEL	10 m/s ²	3 m/s ²	3 m/s ²
MAX_DSTEER	180°/s	90°/s	45°/s
Rk	[0.001, 90]	[0.001, 90]	[0.001, 110]
Rdk	[0.1, 500]	[0.1, 600]	[0.001, 110]
Qk	[60, 60, 30, 60]	[70, 70, 30, 70]	[70, 70, 5.5, 60]
Qfk	[60, 60, 30, 60]	[70, 70, 30, 70]	[70, 70, 5.5, 60]

Tabella 1: Confronto dei profili di MPC per il circuito di *Spa*.

4.6.2 Monza

Similmente, per il circuito di *Monza* nella Tabella 2 sono riportati i valori scelti per ciascuna configurazione.

Parametro	Safe	Fast	High Performance
TK	5	5	7
DTK	0.03 s	0.03 s	0.03 s
MAX_SPEED	15 m/s	15 m/s	15 m/s
MAX_ACCEL	10 m/s ²	3 m/s ²	3 m/s ²
MAX_DSTEER	180°/s	45°/s	45°/s
Rk	[0.001, 90]	[0.005, 80]	[0.01, 100]
Rdk	[0.1, 750]	[0.1, 750]	[0.01, 100]
Qk	[200, 200, 20, 200]	[200, 200, 20, 200]	[50, 50, 5.5, 50]
Qfk	[200, 200, 20, 200]	[200, 200, 20, 200]	[50, 50, 5.5, 50]

Tabella 2: Confronto dei profili di MPC per il circuito di Monza.

4.7 Visualizzazione

L'algoritmo *MPC* viene visualizzato su *RViz* attraverso la pubblicazione di *Marker* appartenenti a *ROS*, osservabili nella Fig. 12.

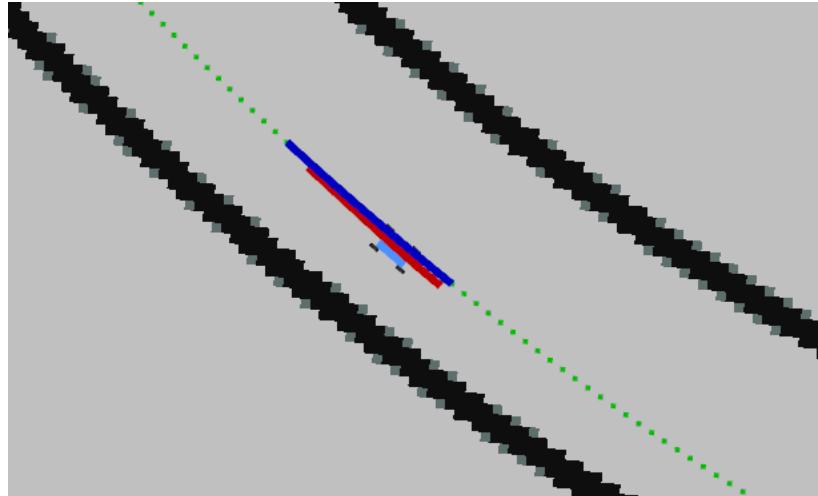


Figura 12: Visualizzazione dei ROS topic utilizzati per *MPC*.

Per visualizzare l'algoritmo si creano e, infine, si pubblicano i marker che rappresentano:

- i *waypoints*, che rappresentano i punti di riferimento che il veicolo deve seguire; per mostrarli si utilizza un marker verde di tipo POINTS.
- la *traiettoria di riferimento*, ovvero la traiettoria ideale che il veicolo dovrebbe seguire, mostrata mediante un marker blu di tipo LINE_STRIP.
- il *percorso predetto*, cioè la traiettoria che il veicolo seguirà in base agli input di controllo attuali, mostrata con un marker rosso di tipo LINE_STRIP.

Tuttavia, si specifica che per visualizzare i marker pubblicati è necessario aggiungere delle nuove schermate di tipologia **Marker** su **RViz**, per ciascuna delle tre visualizzazioni. Inoltre, bisogna configurare correttamente i **topic** su cui si pubblicano i marker definiti, ad esempio, per i waypoints è stato scelto il nome “`/visualization_waypoints`”.

Capitolo 5

Analisi dei risultati

In questo capitolo si presentano e analizzano i risultati ottenuti con l'algoritmo *Model Predictive Control*. In particolare, sono state individuate delle metriche specifiche per misurare i risultati dell'algoritmo, i quali sono stati poi confrontati tra i diversi metodi di controllo, soprattutto per quanto riguarda i profili di guida di *MPC* nelle due piste testate. Successivamente, i risultati ottenuti sono stati analizzati e rappresentati graficamente attraverso dei notebook *Jupyter*, in modo da visualizzare le peculiarità emerse dal lavoro svolto.

5.1 Metriche usate

L'analisi dei risultati – e il resto del lavoro – è stata effettuata su Ubuntu 22.04 (kernel 6.8.0-45-generic), con un sistema avente come CPU un i5-12600K @ 4.9GHz e 32 GB di RAM DDR4 @ 3200 MHz. Gli esperimenti condotti comprendono principalmente dei test sui tracciati di *Spa* e *Monza* per ciascun profilo di guida di *MPC*. Ognuno di essi è stato confrontato con *Pure Pursuit*, che è stato sfruttato come metodo *baseline*. Inoltre, sono stati effettuati confronti tra i tre profilo di *MPC*: in questo caso, si è preso il profilo *High Performance* come riferimento.

Ciascuna configurazione del sistema è stata testata al fine di ottenere una valutazione complessiva delle diverse situazioni operative in cui l'algoritmo si è trovato ad agire.

Le metriche individuate per l'analisi delle performance del controllore comprendono [26]:

- *Lap time*, cioè la misurazione del tempo sul giro, che viene misurato da quando l'auto supera il punto di partenza fino a quando lo raggiunge nuovamente.
- *Errore di tracking*, detto anche *Crosstrack Error*, cioè la distanza tra la linea teorica e quella che viene realmente seguita dal veicolo in una specifica posizione nella mappa. In più, si va a misurare la *deviazione massima* d_{\max} dalla linea di riferimento.

$$\text{Errore di Tracking} = d = \sqrt{(x_{\text{actual}} - x_{\text{ref}})^2 + (y_{\text{actual}} - y_{\text{ref}})^2}$$

- *Deviazione standard campionaria* dell'errore di tracking.

$$\sigma_d = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (d_i - \mu)^2}$$

- *Root Mean Squared Error (RMSE)*, per valutare l'accuratezza del modello nel seguire la traiettoria di riferimento. Infatti, misura la distanza tra il punto più vicino della traiettoria di riferimento da seguire e la posizione effettiva del veicolo. Viene scelto perché, essendo sensibile a valori grandi, esso penalizza maggiormente le deviazioni maggiori.

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n d_i^2}$$

- *Consumo energetico*, misurato per ogni posizione dell'auto durante un giro.

$$P = mav = 3.74 \text{Kg av}$$

Ognuna di queste metriche è stata misurata e calcolata per i circuiti usati, ovvero *Spa* e *Monza*, come detto in precedenza.

5.1.1 Circuiti

Dopo aver definito le metriche, per ciascun circuito sono stati raccolti i risultati ottenuti nelle tre configurazioni di *MPC*. Inoltre, come anticipato, è stato utilizzato *Pure Pursuit* come *baseline* [27] per confrontarlo con gli altri metodi di *MPC*.

Circuito di Spa					
Metodo	Lap Time [s]	d_{\max} [m]	RMSE [m]	σ_d [m]	Energia [W]
<i>Pure Pursuit</i>	85.1	0.638	0.211	0.043	6.352
Safe MPC	78.0	0.389	0.111	0.057	0.217
Fast MPC	75.2	0.380	0.115	0.06	0.547
HP MPC	68.5	0.518	0.088	0.044	2.748
Circuito di Monza					
Metodo	Lap Time [s]	d_{\max} [m]	RMSE [m]	σ_d [m]	Energia [W]
<i>Pure Pursuit</i>	57.4	0.679	0.21	0.045	6.944
Safe MPC	52.4	0.244	0.087	0.047	0.841
Fast MPC	50.6	0.299	0.092	0.051	0.706
HP MPC	49.6	0.261	0.083	0.044	1.156

Tabella 3: Confronto dei metodi di *MPC* con *Pure Pursuit* come baseline.

Dai risultati ottenuti e indicati nella Tabella 3, emergono diversi punti chiave da considerare per le due piste:

1. *Lap Time* – I diversi profili di *MPC* superano nettamente *Pure Pursuit*, con *High Performance MPC* che risulta essere il più veloce in entrambi i circuiti. Questi risultati suggeriscono che l'ottimizzazione svolta con *MPC* permette di migliorare significativamente anche la velocità media nel circuito, come si può notare nella Tabella 4.
2. *Deviazione massima dalla traiettoria ideale* – *Safe MPC* è il profilo con la deviazione massima inferiore su *Monza*, mentre su *Spa* è invece *Fast MPC*. Si osserva come *Pure Pursuit* si discosta maggiormente dalla traiettoria ideale in entrambi i circuiti.
3. *RMSE* – *HP MPC* gode di valori di *RMSE* inferiori per entrambe le piste. Esso quindi garantisce una maggior precisione nel seguire la traiettoria. Infatti, si rileva che *Pure Pursuit* ha un errore superiore rispetto a *HP MPC* di circa il

153% per *Monza* e di circa il 140% per *Spa*. Inoltre, il profilo *Safe* ha valori inferiori rispetto al *Fast*.

4. *Deviazione standard dell'errore di tracking – Fast MPC* tende ad avere più variabilità nella traiettoria seguita, seguito in ordine dai profili *Safe* e *HP*; d'altro canto, *Pure Pursuit* ha una variabilità della traiettoria ridotta, molto simile a quella di *HP MPC*.
5. *Consumo energetico – HP MPC* tende a consumare in media più energia per poter ottenere prestazioni migliori, tuttavia *Pure Pursuit* consuma molto di più, senza però offrire le stesse performance e gli stessi vantaggi di *MPC*.

Infine, la Tabella 4 riporta i dati relativi alle velocità ottenute con ciascun metodo di controllo. Queste vengono confrontate coi valori teorici segnati nel dataset per ciascun waypoint. Dai risultati nei profili di *MPC* si rileva che, se per *Monza* non si osserva un superamento delle prestazioni rispetto alle velocità teoriche generate dal *planner*, per *Spa* si ottiene sempre un miglioramento, eccetto che per *Safe MPC*.

Velocità per il circuito di Spa			
Metodo	Media [m/s]	Underperf. [%]	Overperf. [%]
<i>Pure Pursuit</i>	7.56	93.49	6.51
Safe MPC	8.07	64.22	35.78
Fast MPC	8.34	45.67	54.33
HP MPC	8.6	38.41	61.59
Velocità per il circuito di Monza			
<i>Pure Pursuit</i>	8.82	96.49	3.51
Safe MPC	9.25	66.55	33.45
Fast MPC	9.72	56.79	43.21
HP MPC	9.77	54.18	45.82

Tabella 4: Confronto dei rendimenti rispetto alla velocità di riferimento per i circuiti di *Spa* e *Monza*.

Ricapitolando, le configurazioni di *MPC*, rispetto a *Pure Pursuit*, non solo migliorano la precisione nel tracking della traiettoria e i tempi di percorrenza del circuito, ma evidenziano anche un'evidente efficienza energetica superiore, considerando anche

la scala ridotta del veicolo. In particolare, il profilo *High Performance MPC* emerge come il metodo più *robusto*, con prestazioni considerevoli e consumi energetici discreti.

5.2 Visualizzazione dei risultati

In questa sezione vengono presentati graficamente i risultati relativi ad altri aspetti del controllo del veicolo applicato al *path tracking*. I risultati sono visualizzati sia per i tre profili di *MPC*, che vengono confrontati tra loro, sia in relazione al *Pure Pursuit*. Inoltre, in alcuni grafici essi vengono mostrati anche rispetto ai valori teorici della *raceline*, che viene presa come riferimento per i dati teorici.

5.2.1 Lap time

I tempi di percorrenza su un giro (*lap time*) sono stati misurati a partire dal secondo, così da consentire al veicolo di stabilizzarsi dopo la partenza. Si è scelto di non misurare oltre anche perché, dopo il secondo giro, le variazioni nei tempi registrati risultano trascurabili.

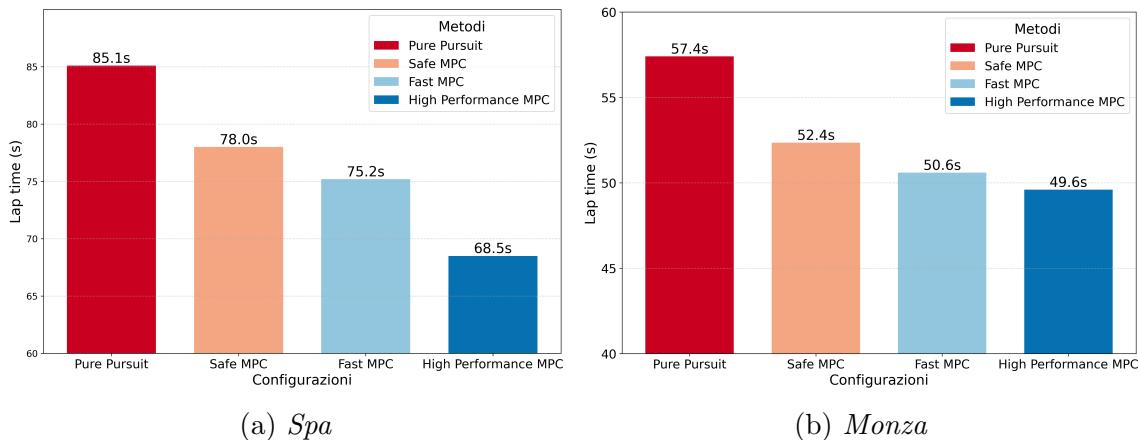


Figura 13: Lap time per ogni metodo di controllo misurato.

Si possono osservare in Fig. 13 i tempi per *Pure Pursuit* e per ciascuna configurazione di *Model Predictive Control*. Essi rispecchiano le aspettative: il metodo col *lap time* inferiore risulta essere *High Performance MPC*, seguito dai metodi *Fast* e

Safe. Invece, *Pure Pursuit* richiede più tempo per completare un giro e, provando a farlo correre ripetutamente, tende ad andare a sbattere con più frequenza per via della semplicità di questo metodo di controllo *reattivo* e *geometrico*.

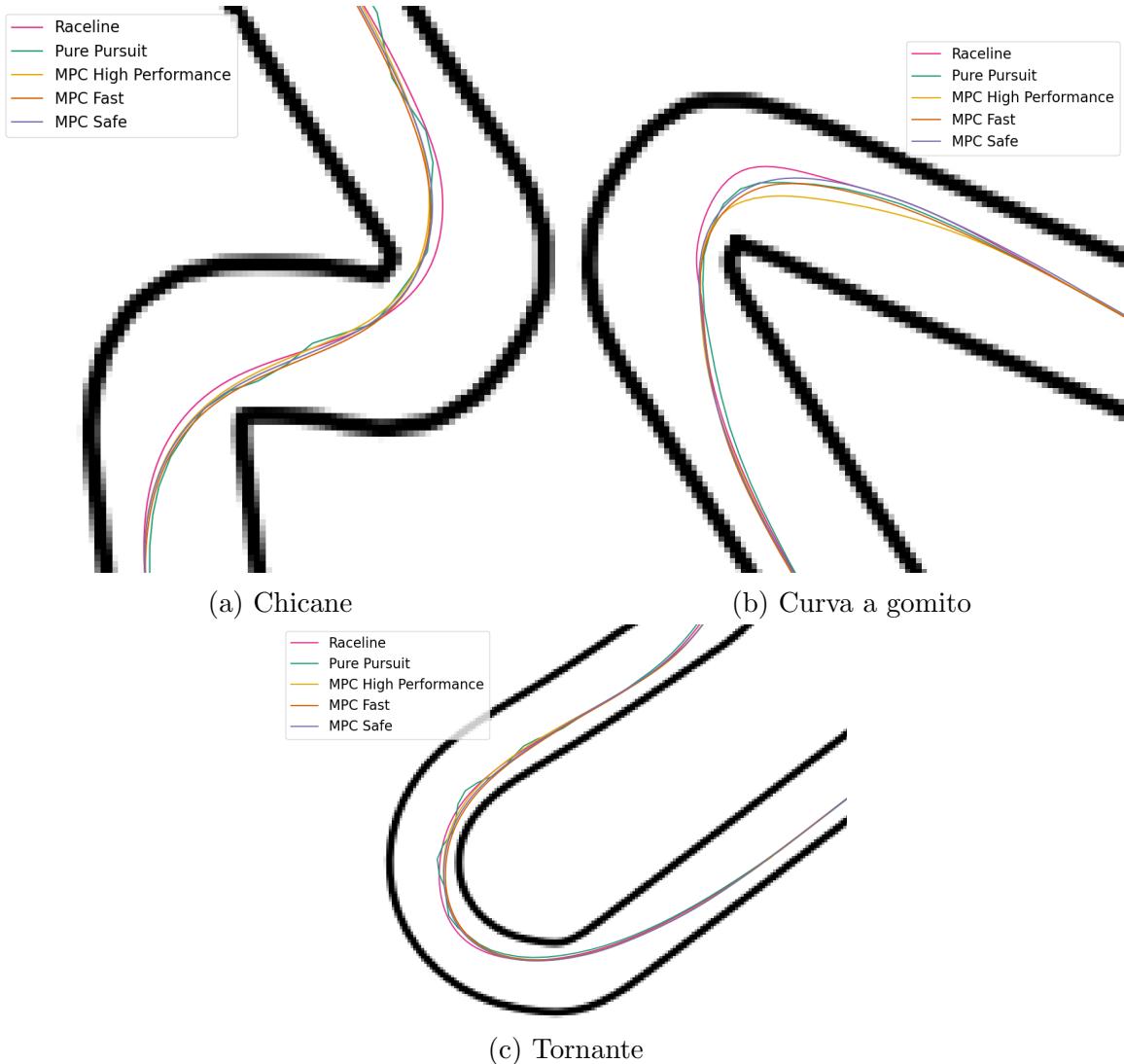
5.2.2 Traiettorie

Per visualizzare le traiettorie, essendo molto ravvicinate tra loro, si utilizzano degli *zoom* su delle zone specifiche del tracciato, ovvero un *rettilineo* e alcune tipologie di curve che sono molto comuni nelle piste di *Formula 1*:

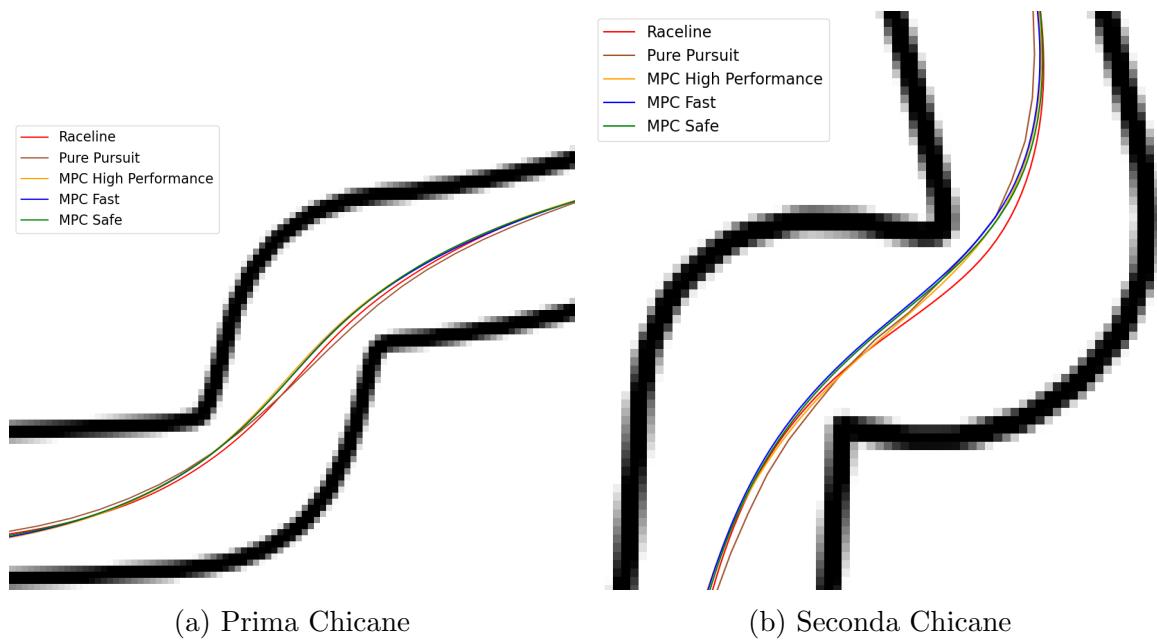
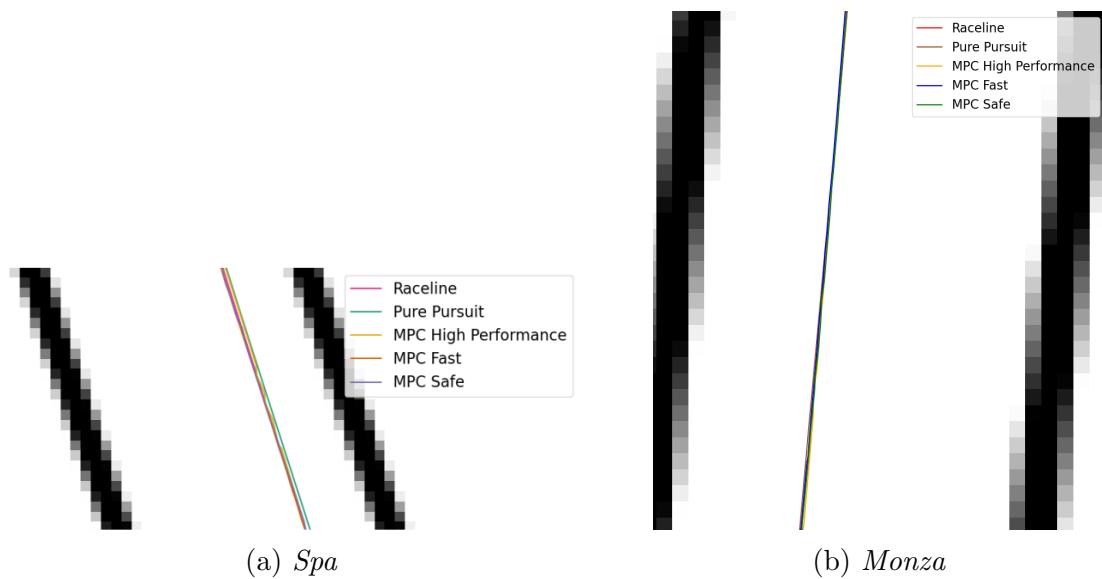
1. *Chicane* – Si tratta di una curva seguita da un’altra curva nella direzione opposta che viene introdotta in un tratto rettilineo di una pista per rallentare la velocità dei veicoli. Può essere utilizzata per testare quanta velocità il veicolo riesce a portare verso l’*apex* (centro curva) – il punto più interno seguito in una curva – in entrambe le curve, e con quanto anticipo ciò consente al sistema di accelerare in uscita dalla curva.
2. *Tornante* – È una svolta di 180 gradi che permette di verificare il modo in cui il sistema raggiunge l’*apex* e, soprattutto, come entra ed esce dalle curve.

Il circuito di *Spa* presenta delle curve *chicane* e un *tornante*. Come visibile dalla Fig. 14, le traiettorie relative ai profili di *MPC* sono più uniformi, poiché l’algoritmo ottimizza sempre “*online*”, e tendono a tagliare prima nelle curve rispetto al riferimento della *raceline*. Le differenze tra i profili di *MPC* sono dovute prettamente ai diversi valori dell’orizzonte temporale e delle matrici di pesi, concetti già discussi nel Capitolo 3, da un punto di vista teorico, e nel Capitolo 4 a livello implementativo.

A differenza di *MPC*, *Pure Pursuit* ha un andamento più variabile, poiché tende a correggere l’angolo di sterzata con elevata frequenza, per seguire il più possibile i waypoints della *raceline* a cui fa riferimento. Quanto descritto lo si può notare specialmente nella curva a gomito della Fig. 14b, nella quale *HP MPC* taglia molto in anticipo rispetto a tutti gli altri.

Figura 14: Diversi tipi di curve su *Spa*.

Si può notare in Fig. 15 un comportamento simile anche per il circuito di Monza, composto prettamente da chicane e da rettilinei. Questi ultimi sono visualizzati per entrambe le piste nella Fig. 16, in modo da evidenziare la vicinanza di tutte le traiettorie, che risultano di fatto sovrapposte.

Figura 15: Curve su *Monza*.Figura 16: Rettilinei su *Monza* e *Spa*.

5.2.3 Crosstrack Error

Nella sezione 5.1, relativa alle metriche, è stato introdotto il *Crosstrack Error*, detto anche errore di tracking. Tuttavia, questo dato è stato usato esclusivamente per mostrare la *deviazione massima* e, soprattutto, per calcolare l'*RMSE*, in modo da avere una misura accurata che indica di quanto si discosta in media l'errore di tracking dallo zero, cioè il caso con valore teorico e valore corrente identici. Si vuole però dare anche un'idea visiva dell'evoluzione dell'errore di tracking, sempre attraverso la realizzazione di un grafico. Nella Fig. 17 viene mostrato, per entrambe le piste, l'errore di tracking generato dal *Pure Pursuit* che, come confermato dall'*RMSE*, ha valori mediamente superiori. Pertanto, questo metodo è stato utilizzato come *baseline* anche per la scala cromatica in ogni grafico inerente a questa metrica. Si segnala che il triangolo indica la posizione di partenza del giro.

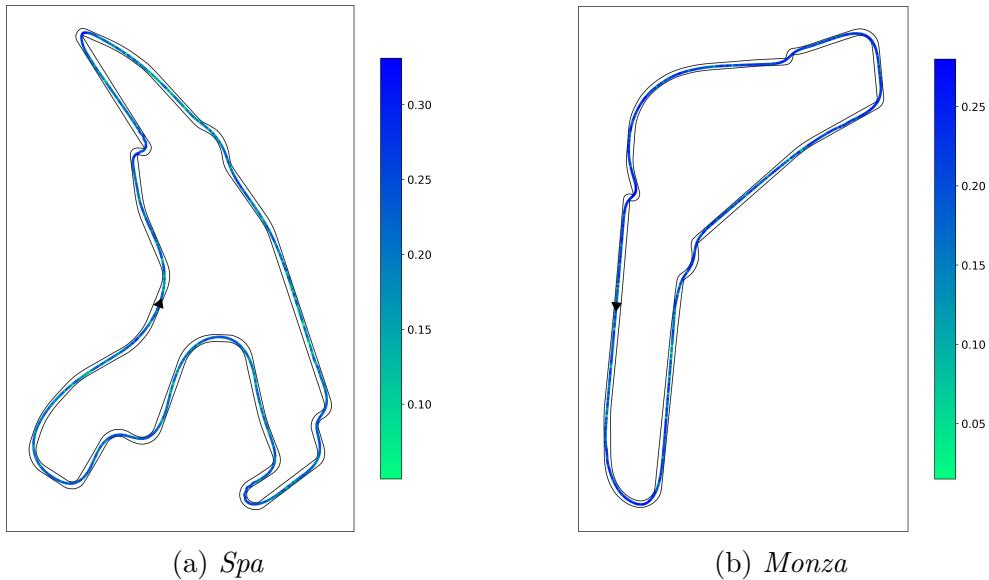


Figura 17: Crosstrack Error per *Pure Pursuit*, usato come baseline.

Infatti, nella Fig. 18 si noti come tutti i profili di *MPC* abbiano un colore più chiaro per la quasi totalità del percorso; in particolare, il profilo *High Performance* è quello coi valori inferiori, seguito da *Safe* – confermando di riflesso il valore basso registrato per l'*RMSE* – e da *Fast*. Inoltre, soprattutto per *High Performance*, si può

notare dal grafico il colore più marcato nella parte della *curva a gomito* – già vista nella Fig. 14b – poiché la “taglia” prima di ogni altro profilo.

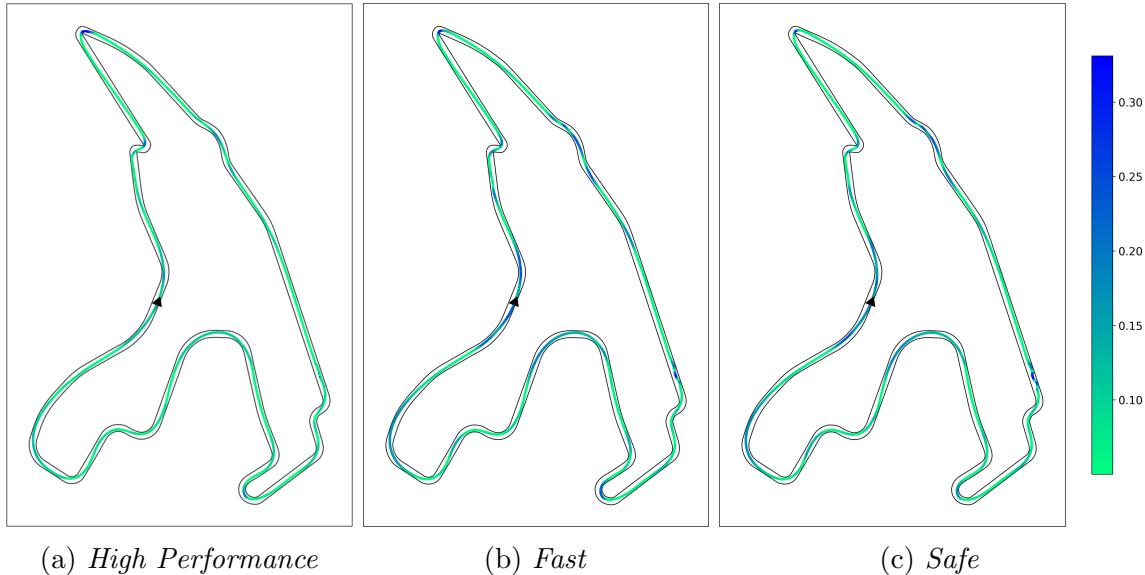


Figura 18: Crosstrack Error per *Spa*.

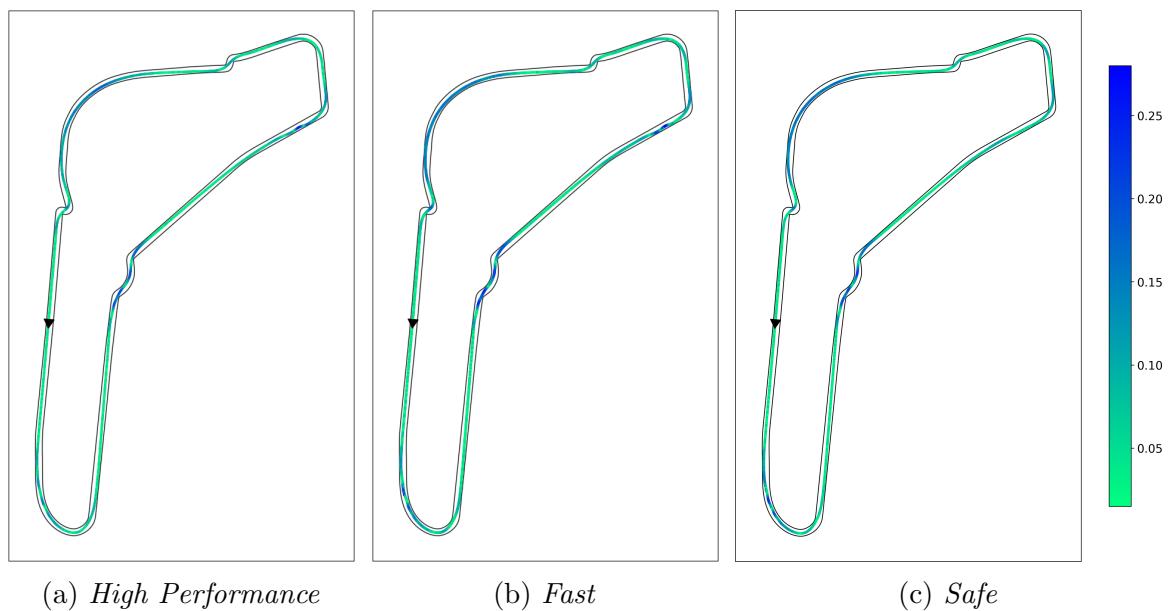


Figura 19: Crosstrack Error per *Monza*.

La situazione è pressoché identica anche per i grafici della pista di *Monza*, visualizzati nella Fig. 19. Unendo i risultati riepilogati nella sezione 5.1, con questi più quelli delle traiettorie mostrate in precedenza, si ottiene un quadro completo relativo alle discrepanze tra i valori reali e quelli ideali, specialmente per quanto riguarda la deviazione dalla linea di riferimento, che è ciò si vuole minimizzare con *MPC*.

5.2.4 Velocità e Angolo di sterzata

Si mostrano di seguito i grafici relativi al confronto della *velocità* e dell'*angolo di sterzata* per ogni zona dei due circuiti. Si precisa che viene mostrato un valore ogni 10 metri per migliorare la resa grafica, così da notare maggiormente le differenze.

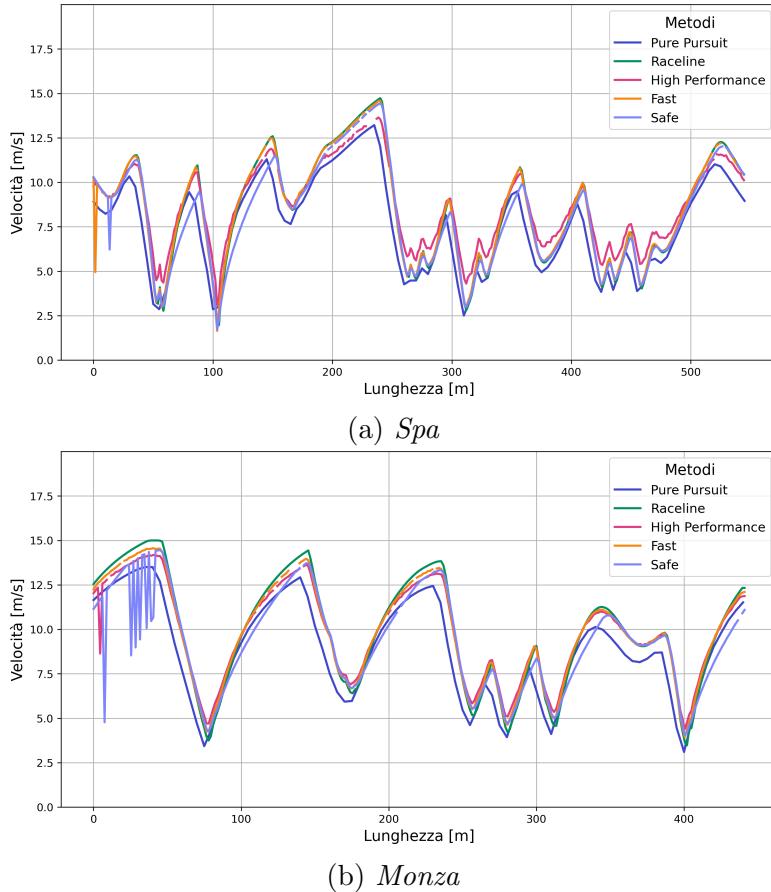


Figura 20: Confronto delle velocità su *Spa* e *Monza*.

Per quanto riguarda la velocità, si può osservare in Fig. 20 come i valori in *Pure Pursuit* restino sempre inferiori agli altri metodi: ciò accade poiché ha un profilo di velocità corrispondente al 90% di quello indicato nel dataset della *raceline*. Invece, per tutti i profili di *MPC*, nei primi 50 metri circa della pista di Monza, si osserva una notevole variazione nella velocità applicata dall'algoritmo. Ciò è dovuto a un iniziale tempo di assestamento, in particolare per il profilo *Safe MPC* dato che parte più lentamente rispetto agli altri e con un'accelerazione di 10 m/s^2 . Al contrario, gli altri profili mostrano un comportamento più attenuato, poiché iniziano con velocità più elevate e un'accelerazione di 3 m/s^2 . Invece, si può osservare nella Fig. 21 quanto appena descritto da un punto di vista qualitativo, dove è stata riportata solo la parte iniziale del giro di *Monza*. Si precisa che la scala cromatica è basata sui valori di velocità della *raceline ottimale*. In più, si osservino le variazioni cromatiche dei punti nello *scatter plot*.

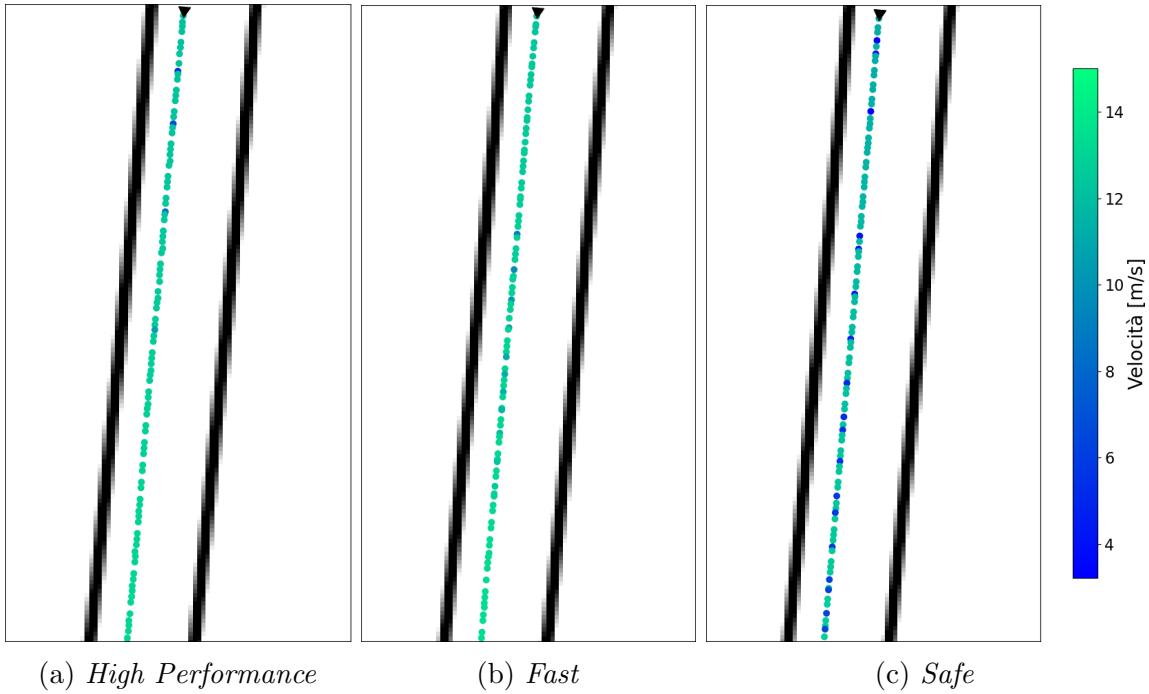


Figura 21: Confronto tra i metodi di *MPC* su *Monza*.

Dopo questa fase iniziale, per la pista di *Monza* le velocità dei tre metodi sono molto simili a quella di riferimento data dalla *raceline ottima*. Invece, per quella

di *Spa*, il profilo *High Performance* tende a superare la velocità di riferimento dalla seconda metà del tracciato.

Viene visualizzato graficamente anche l'angolo di sterzata δ attraverso il confronto dei valori applicati dal *Pure Pursuit* con quelli delle tre configurazioni di *MPC*.

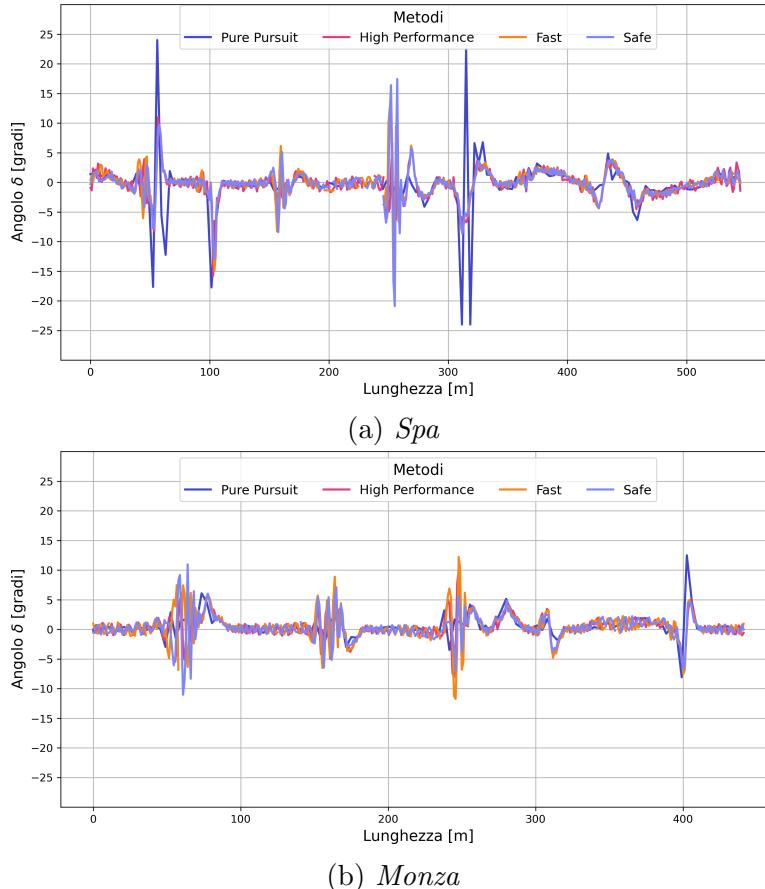


Figura 22: Confronto degli angoli di sterzata applicati su *Spa* e *Monza*.

Come mostrato in Fig. 22, il *Pure Pursuit* presenta angoli di sterzata massimi più elevati in specifici punti di *Spa* rispetto a *Monza*, mentre altrove è generalmente simile a *MPC*. Infatti, i profili *MPC* mostrano angoli analoghi tra loro su *Spa*, con aumenti in certe zone per *Safe* e *Fast*, ma anche su *Monza* si ha un comportamento simile. Dunque, in generale, i profili di *MPC* hanno applicato angoli assimilabili a quelli di *Pure Pursuit*, a eccezione di certe zone di *Spa*.

Capitolo 6

Conclusioni

In quest’ultimo capitolo si traggono le conclusioni sui risultati delle analisi svolte e sui possibili sviluppi futuri di questo lavoro.

6.1 Risultati ottenuti

Si è arrivati a realizzare un sistema di controllo avanzato che permette di ottimizzare l’attività del *path tracking* attraverso l’implementazione in *Python* dell’algoritmo *Model Predictive Control*. Si è dunque in grado di seguire la linea di riferimento, ovvero la raceline generata dal *planner*, con una traiettoria ottimale calcolata in tempo reale lungo l’intero percorso. Ciò permette di minimizzare la deviazione dalla linea di riferimento, che è stata misurata mediante il *Crosstrack Error* e, inoltre, nelle diverse configurazioni si è osservata spesso una riduzione dei tempi di percorrenza sul giro.

Sulla base dei risultati presentati nel Capitolo 5, le diverse configurazioni di *MPC* risultano superiori a un metodo di controllo geometrico come il *Pure Pursuit*, appartenente alla categoria dei metodi reattivi per il controllo di veicoli. Si tratta di un risultato previsto, che risulta coerente con la formulazione di *MPC*, ovvero come un problema di ottimizzazione con vincoli che è caratterizzato da una struttura non banale che guarda al “*futuro*”.

D’altro canto, il *Pure Pursuit* è un metodo di *path tracking geometrico*, poiché

calcola l'angolo di sterzata da applicare alle ruote per raggiungere il waypoint successivo nella linea di riferimento. Questo metodo, a differenza di *MPC*, presenta un andamento più irregolare; infatti, porta a seguire una traiettoria che, osservando i risultati del *Crosstrack Error*, si discosta di molto dalla linea teorica, con valori medi superiori tra il 140% e il 153% rispetto al profilo *High Performance MPC*. Si è anche rilevato che *Pure Pursuit* consuma molta più energia di ogni metodo di *MPC*. Inoltre, come qualsiasi altro metodo di controllo reattivo, esso non considera in alcun modo la dinamica del sistema, pertanto può produrre archi impraticabili; contrariamente, *MPC* incorpora un modello della dinamica, come il *Kinematic Bicycle Model*, discusso nella sezione 3.2.1.

Infine, dalle diverse configurazioni di *MPC* emergono risultati differenti tra loro: la configurazione *High Performance* risulta essere la migliore in un contesto di guida autonoma con un singolo agente che corre a velocità estreme, prossime ai limiti fisici del veicolo. Nello specifico, per questo profilo si ottiene che, per entrambi i circuiti, si hanno dei *lap time* e degli *RMSE* inferiori, con un buon compromesso per ciò che riguarda il *consumo energetico*, il quale risulta però minore per i metodi *Safe* e *Fast*. Ciò non è casuale, infatti *High Performance* presenta prestazioni superiori anche in termini di velocità su entrambe le piste testate. Invece, per quanto riguarda l'angolo di sterzata applicato, non si rilevano particolari miglioramenti per i profili di *MPC*.

6.2 Sviluppi futuri

Gli sviluppi futuri per questo progetto possono muoversi verso nuove prospettive, sia per poterlo applicare per attività più complesse, sia per migliorare la soluzione di *MPC* realizzata.

Sim2Real La prima evoluzione consiste nel passaggio dalla simulazione alla realtà. Ciò implicherebbe sfide non banali, come l'adattamento di diversi valori di configurazione e di certe strategie decisionali a livello implementativo, oltre alla costruzione del veicolo. In un ambiente reale vi è incertezza: il modello utilizzato nella simulazione è solo un'approssimazione e, in più, l'attuazione su un veicolo reale non è più

solo prodotta da un software, ma è in larga parte meccanica. Sarà dunque cruciale effettuare un'attenta attività di *tuning* per ottimizzare le prestazioni.

Modelli più complessi Si possono adottare modelli più aderenti alla realtà, come il *Dynamic Bicycle Model*, che considerano dinamiche non lineari e fenomeni aerodinamici tipici nell'*autonomous racing* a velocità elevate. I modelli non lineari, infatti, potrebbero migliorare la precisione del controllo, al costo però di tempi di risoluzione più lunghi e di possibili valori inferiori per la velocità. Si avrebbe così un problema non convesso, che andrebbe risolto con un risolutore non lineare come *Casadi*.

Competizione multi-agente Un'altra direzione di ricerca potrebbe essere data da un contesto di competizione con due (o più) veicoli. Questo tipo di lavoro richiederebbe nuove strategie per gestire il comportamento competitivo, legate alla *teoria dei giochi*. Ciò implicherebbe, ad esempio, lo sviluppo di tecniche per ottimizzare i sorpassi, la difesa della traiettoria e la gestione delle collisioni.

Reti Neurali e MPC Data-Driven Si potrebbero integrare anche delle reti neurali [28, 29] per migliorare ulteriormente il processo di *path tracking*. Nello specifico, si potrebbero utilizzare tecniche di *Imitation Learning (IL)* e *Reinforcement Learning (RL)* al fine di apprendere comportamenti ottimali dai dati registrati nei giri precedenti. I lavori più recenti sui controller nell'*autonomous racing* si sono concentrati proprio sullo sviluppo di componenti interni basati sull'apprendimento, come soluzioni di *Learning MPC* [30, 31] che applicano proprio queste idee. In particolare, si potrebbe esplorare un approccio ibrido in cui il *controller* sfrutta tecniche basate su modelli classici e le combina coi dati raccolti da esperienze passate per aggiornare e migliorare le prestazioni del sistema in tempo reale.

Ringraziamenti

Mi trovo a scrivere queste parole dopo un percorso molto tormentato, pieno di ostacoli e delusioni. Queste non riguardano esclusivamente l'esperienza universitaria, ma sono dovute in larga parte a ciò che ho passato al di fuori di essa in questi quattro anni. Tutto questo ha causato stress, tristezza e ritardi, specialmente nel primo semestre del secondo anno, che è stato, senza alcun dubbio, il periodo più duro di questi anni.

Innanzitutto, vorrei però ringraziare il mio relatore, il **Prof. Matteo Luperto**, che ha permesso di farmi lavorare a un progetto di tesi molto stimolante su tematiche attuali e non banali, ed è sempre stato disponibile per rispondere ai miei dubbi con ricevimenti e messaggi.

Voglio dedicare le prossime righe alle persone più importanti della mia vita: senza di voi sarebbe stato tutto più complicato.

Ringrazio di cuore **Rossella**, la mia ragazza. Ci siamo conosciuti proprio durante quel periodaccio al secondo anno: sei arrivata nel momento del bisogno, e ti sarò sempre immensamente grato per questo.

Grazie per tutto quello che hai fatto per me in questi anni: il tuo sostegno e il tuo amore sono stati fondamentali per uscire da quel tunnel pieno di negatività. Mi hai aiutato a dare sempre il massimo, senza lasciarmi abbattere.

Grazie per aver percorso, innumerevoli volte, chilometri su chilometri per vedermi, anche quando si riusciva solo per poco tempo. Ogni attimo passato con te è stato linfa vitale; sei sempre riuscita a farmi distrarre dallo studio quando ne avevo bisogno. Infatti, voglio anche ringraziarti per avermi sopportato in quei periodi di maggior stress; per avermi consolato quando un esame non andava bene; per aver letto ogni messaggio di sfogo; per aver ascoltato ogni audio di almeno due minuti o un intero monologo in videochiamata riguardante ciò che stavo studiando o che restava ancora da fare. Grazie per aver provato a capire ciò che ho studiato in

questi anni. Non l'hai fatto perché ti sentivi obbligata, ma perché volevi darmi, da buona *perfezionista*, dei consigli consapevoli.

Ross, grazie per tutto ciò che sei. **Sei speciale.**

Voglio poi ringraziare i miei **genitori**. Mi risulta difficile descrivere in poche parole ciò che provo per voi, solo io sono a conoscenza dei *sacrifici* che avete fatto per me in questi 23 anni. Grazie per avermi permesso di arrivare a questo momento e scusatemi per tutte le volte che sono stato intrattabile durante le sessioni.

Inoltre, voglio ringraziare profondamente le persone che ho conosciuto in questo percorso di studi: ci siamo aiutati a vicenda su Discord per la preparazione di ogni esame, abbiamo passato insieme tanti momenti stupendi, ma anche altri negativi. Infatti, si sono venuti a creare dei legami forti, e sono sicuro che senza il vostro aiuto sarebbe stato tutto più noioso e intricato. Siete stati importanti e desidero ringraziarvi uno ad uno: **Emanuele Manca**¹ (*Ema*), **Davide Papasodaro** (*Papas*), **Gabriele Giorgio** (*Gabri*), **Federico Marcelli Fabiani** (*Fede*), **Marco Morandi** (il fu *Jimmy*), **Ivan (il) Selvaggio**, **Gabriele Gilberti** (*Gigi*), **Cristian Pozzi** (*Criii Pòtzieh*), **Federico Coscia** (*Fedoscia*). Ultimo, ma non per importanza, **Navjot Kumar** (*Nav*), il mio vecchio amico d'infanzia ed ex compagno delle superiori – assieme a **Ema** e **Cri** – col quale ho condiviso anche il primo anno. Voglio ringraziare anche **Mattia Oldani** che, pur non facendo parte del mio gruppo più stretto, è stato d'aiuto durante l'ultimo anno. Noi non dimenticheremo quei parziali di Fisica.

Infine, voglio ringraziare tutti i miei **amici** più cari di Vercelli, in particolar modo **Mattia Manzoni** (**B**attia!), il mio miglior amico, fresco fresco di certificazione per personal trainer. Vi ringrazio tutti perché nei weekend di questi ultimi anni mi avete aiutato a rilassarmi e distrarmi, facendomi passare delle belle serate in compagnia. Mi dispiace per tutti quei “Non ci sono, mi devo svegliare presto domani”, “Non riesco, devo ripassare per l'esame” e così via.

Ne è valsa la pena.

¹<https://youtu.be/3-4u6BVLHcI> ← Qui potete vedere me ed Ema dopo aver dato l'ultimo esame della triennale.

Bibliografia

- [1] F1TENTH. *Sito ufficiale*. URL: <https://f1tenth.org>.
- [2] F1TENTH. *Documentazione ufficiale*. URL: <https://f1tenth.org/learn.html>.
- [3] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette e William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022). DOI: [10.1126/scirobotics.abm6074](https://doi.org/10.1126/scirobotics.abm6074).
- [4] Steven Gong. *Middleware*. URL: <https://stevengong.co/notes/Robot-Operating-System>.
- [5] ROS 2. *Concetti di ROS 2*. URL: <https://docs.ros.org/en/humble/Concepts/Basic.html>.
- [6] Johannes Betz, Hongrui Zheng, Alexander Liniger, Ugo Rosolia, Phillip Karle, Madhur Behl, Venkat Krovi e Rahul Mangharam. «Autonomous Vehicles on the Edge: A Survey on Autonomous Vehicle Racing». In: *IEEE Open Journal of Intelligent Transportation Systems* 3 (2022), pp. 458–488. DOI: [10.1109/ojits.2022.3181510](https://doi.org/10.1109/ojits.2022.3181510).
- [7] University of Pennsylvania. *Lezione 7 di F1TENTH. Filtering: Introduction to state estimation and recursive bayes filters*. Con comm. di Zhijun Zhuang. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [8] University of Pennsylvania. *Lezione 8 di F1TENTH. Particle Filter: Mapping, localization, AMCL and particle filter Tuning*. Con comm. di Rahul Mangharam. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [9] University of Pennsylvania. *Lezione 9 di F1TENTH. Introduction to Graph-based SLAM: Graph SLAM, Probability and Scan Matching, Sparse Pose Adjustment*. Con comm. di Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [10] University of Pennsylvania. *Lezione 12 di F1TENTH. Local Planning: RRT, Spline Based Planner*. Con comm. di Rahul Mangharam e Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.

- [11] Sertac Karaman e Emilio Frazzoli. «Sampling-based algorithms for optimal motion planning». In: *The International Journal of Robotics Research* 30.7 (2011), pp. 846–894. DOI: [10.1177/0278364911406761](https://doi.org/10.1177/0278364911406761).
- [12] University of Pennsylvania. *Lezione 10 di F1TENTH. Pure Pursuit as a geometric control method*. Con comm. di Rahul Mangharam e Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [13] University of Pennsylvania. *Lezione 4 di F1TENTH. PID for Wall Following: Introduction to reference tracking using PID control*. Con comm. di Rahul Mangharam e Hongrui Zheng. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [14] Nantian Electronics. *Differenza tra Open Loop e Closed Loop*. URL: <https://www.ntchip.com/electronics-news/difference-between-open-loop-and-closed-loop>.
- [15] Alexander Wischnewski, Phillip Karle e Markus Lienkamp. *Autonomous Driving Software Engineering - Lecture 08: Control*. URL: https://www.researchgate.net/publication/352322266_Autonomous_Driving_Software_Engineering_-_Lecture_08_Control.
- [16] PLCynergy. *Schema di PID*. URL: <https://plcynergy.com/pid-controller/>.
- [17] University of Pennsylvania. *Lezione 13 di F1TENTH. Optimization Basics and MPC: Introduction to constrained optimal control, MPC formulation and application*. Con comm. di Rahul Mangharam e Ahmad Amine. URL: <https://ahmadamine998.github.io/6150-Spring2024-Website/lectures/>.
- [18] Jacob Olausson e Jacob Larsson. «Optimal Control and Race Line Planning for an Autonomous Race Car». Tesi di laurea mag. Linköping University, Vehicular Systems, 2021, p. 82.
- [19] Martin Behrendt. *Schema di base di MPC*. URL: https://en.m.wikipedia.org/wiki/File:MPC_scheme_basic.svg.
- [20] Cyrill Stachniss. *Model Predictive Control - Part 1: Introduction to MPC*. Con comm. di Lasse Peters. URL: <https://youtu.be/XaD8Lngfkzk>.
- [21] Matthias Althoff, Markus Koschi e Stefanie Manzinger. «CommonRoad: Composable benchmarks for motion planning on roads». In: *Proc. of the IEEE Intelligent Vehicles Symposium*. 2017.
- [22] Achin Jain, Matthew O'Kelly, Pratik Chaudhari e Manfred Morari. «BayesRace: Learning to race autonomously using prior experience». In: *arXiv preprint arXiv:2005.04755* (2020).
- [23] B. Stellato, G. Banjac, P. Goulart, A. Bemporad e S. Boyd. «OSQP: an operator splitting solver for quadratic programs». In: *Mathematical Programming Computation* 12.4 (2020), pp. 637–672. DOI: [10.1007/s12532-020-00179-2](https://doi.org/10.1007/s12532-020-00179-2). URL: <https://doi.org/10.1007/s12532-020-00179-2>.

- [24] José L. Vázquez, Marius Brühlmeier, Alexander Liniger, Alisa Rupenyan e John Lygeros. «Optimization-based hierarchical motion planning for autonomous racing». In: *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)* (2020), pp. 2397–2403. DOI: [10.1109/IROS45743.2020.9341731](https://doi.org/10.1109/IROS45743.2020.9341731).
- [25] F1TENTH. *F1TENTH Racetracks*. URL: https://github.com/f1tenth/f1tenth_racetracks.
- [26] Yashom Dighe, Youngjin Kim, Smit Rajguru, Yash Turkar, Tarunraj Singh e Karthik Dantu. «Kinematics-only differential flatness based trajectory tracking for autonomous racing». In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2023), pp. 1629–1636. DOI: [10.1109/IROS55552.2023.10341603](https://doi.org/10.1109/IROS55552.2023.10341603).
- [27] Jonathan Becker, Nadine Imholz, Luca Schwarzenbach, Edoardo Ghignone, Nicolas Baumann e Michele Magno. «Model-and acceleration-based pursuit controller for high-performance autonomous racing». In: *2023 IEEE International Conference on Robotics and Automation (ICRA)* (2023), pp. 5276–5283. DOI: [10.1109/ICRA48891.2023.10161472](https://doi.org/10.1109/ICRA48891.2023.10161472).
- [28] Alexandra Tătulea-Codrean, Tommaso Mariani e Sebastian Engell. «Design and Simulation of a Machine-learning and Model Predictive Control Approach to Autonomous Race Driving for the F1/10 Platform». In: *IFAC-PapersOnLine* 53.2 (2020), pp. 6031–6036. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2020.12.1669>.
- [29] Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza e Peter Dürr. «Super-human performance in gran turismo sport using deep reinforcement learning». In: *IEEE Robotics and Automation Letters* 6.3 (2021), pp. 4257–4264.
- [30] Haoru Xue, Edward L. Zhu, John M. Dolan e Francesco Borrelli. «Learning Model Predictive Control with Error Dynamics Regression for Autonomous Racing». In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. 2024, pp. 13250–13256. DOI: [10.1109/ICRA57147.2024.10611628](https://doi.org/10.1109/ICRA57147.2024.10611628).
- [31] Ugo Rosolia e Francesco Borrelli. «Learning how to autonomously race a car: a predictive control approach». In: *IEEE Transactions on Control Systems Technology* 28.6 (2019), pp. 2713–2719.