UNIVERSITÀ
DI PAVIA

FACOLTA' DI INGEGNERIA
DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INDUSTRIAL AUTOMATION ENGINEERING

TESI DI LAUREA

METODI DI OTTIMIZZAZIONE DELLA NAVIGAZIONE DI UN ROBOT MOBILE MEDIANTE DIGITAL TWIN

Candidato Fabrizio Corradini

Relatore: Prof. Francesco Benzi

A.A. 2024/2025

Ringraziamenti

A me stesso, per aver pensato tutti i giorni di arrendermi senza mai farlo davvero.
Alla mia famiglia, perché è stata, è, e sarà la mia famiglia.
A chi ci avrebbe scommesso, e ancor di più a chi avrebbe scommesso il contrario.
Al dottor Francesco Capelli, per avermi indicato la luce quando attorno a me vedevo solo il buio.

Ringrazio inoltre il mio relatore Francesco Benzi per il sostegno dimostratomi

INDICE

Capitolo 1

Introduzione

1.1 Contesto e motivazione

Negli ultimi anni il settore della robotica mobile ha assunto un ruolo sempre più rilevante in ambito industriale, logistico e di ricerca. I robot mobili autonomi vengono oggi impiegati in magazzini automatizzati, ambienti produttivi, strutture sanitarie e contesti di servizio, dove è richiesta la capacità di muoversi in ambienti dinamici in modo sicuro ed efficiente.

Parallelamente, l'evoluzione delle tecnologie digitali ha introdotto nuovi strumenti per la progettazione, il monitoraggio e l'ottimizzazione dei sistemi fisici complessi. Tra questi, il concetto di *Digital Twin* si è affermato come una delle soluzioni più promettenti per integrare mondo reale e mondo digitale. Un Digital Twin consiste in una replica digitale dinamica di un sistema fisico, in grado di ricevere dati in tempo reale, elaborarli potendo così rappresentarne lo stato operativo. [7]

Nel contesto della robotica mobile, l'integrazione di un Digital Twin consente di monitorare parametri quali posizione, velocità e comportamento del sistema durante la navigazione, offrendo la possibilità di analizzare le prestazioni e individuare eventuali criticità. L'utilizzo di ambienti di simulazione, inoltre, permette di sviluppare e testare soluzioni in modo controllato, riducendo costi e

il dispendio di energie, oltre che i rischi associati alla sperimentazione su hardware reale.

L'interesse verso l'applicazione del Digital Twin in ambito robotico è motivato dalla necessità di migliorare l'affidabilità, l'efficienza e la tracciabilità dei sistemi autonomi. La possibilità di raccogliere e analizzare dati provenienti dal sistema consente di valutare in modo quantitativo il comportamento del robot, supportando decisioni progettuali e ottimizzazioni successive.

Il presente lavoro si colloca in questo scenario, proponendo lo sviluppo di un sistema di Digital Twin applicato a un robot mobile operante in ambiente simulato, con l'obiettivo di monitorarne, analizzarne e ottimizzarne le prestazioni durante la navigazione.

Esempio di robot mobile autonomo operante in contesto industriale

1.2 Obiettivo della tesi

L'obiettivo del presente lavoro è sviluppare e validare un sistema di *Digital Twin* applicato a un robot mobile operante in ambiente simulato, al fine di monitorarne in tempo reale lo stato e migliorarne le prestazioni durante la navigazione.

In particolare, il progetto prevede l'utilizzo del framework ROS (Robot Operating System) integrato con l'ambiente di simulazione Gazebo per la modellazione e il controllo di un robot mobile a cinematica differenziale. A partire dai dati generati dal sistema simulato, viene implementato appunto un gemello digitale in grado di acquisire e rappresentare parametri significativi quali posizione, velocità e traiettoria percorsa.

Il Digital Twin sviluppato non si limita alla semplice visualizzazione dello stato del robot, ma consente la raccolta e l'analisi quantitativa dei dati di navigazione, permettendo di valutare le prestazioni del sistema in differenti scenari operativi. Attraverso test sperimentali, con tecniche di *Hardware-In-The-Loop* vengono analizzati parametri come il tempo di percorrenza, la deviazione dalla traiettoria prevista e l'eventuale presenza di collisioni.

L'obiettivo finale è dimostrare come l'integrazione di un Digital Twin in ambiente simulato possa costituire uno strumento efficace per il monitoraggio e l'analisi delle prestazioni di un robot mobile, rivelandosi un fedele alleato per l'ottimizzazione dell'efficienza della navigazione del robot (in questo caso specifico) fornendo una base per possibili sviluppi futuri in contesti industriali reali.

1.3 Struttura del lavoro

Il presente elaborato è articolato in sette capitoli, ciascuno finalizzato a descrivere in modo progressivo le fasi di progettazione, implementazione e validazione del sistema sviluppato.

Dopo il presente capitolo introduttivo, nel Capitolo 2 vengono richiamati i fondamenti teorici necessari alla comprensione del lavoro svolto. In particolare, vengono analizzati i principi di funzionamento dei robot mobili a cinematica differenziale, con riferimento al modello matematico del moto e alle principali caratteristiche dinamiche. Successivamente viene descritta l'architettura del framework ROS, evidenziandone la struttura a nodi, i meccanismi di comunicazione basati su "topic" e la gestione dello stato del sistema. Infine, vengono introdotti l'ambiente di simulazione Gazebo e il concetto di Digital Twin, limitatamente agli aspetti funzionali alla realizzazione del progetto.

Il Capitolo 3 è dedicato alla progettazione del sistema sviluppato. In questa sezione viene presentato il robot mobile utilizzato ed è illustrata l'architettura software implementata, con particolare attenzione all'organizzazione dei nodi ROS e al flusso di dati tra simulazione e modello digitale. Viene inoltre descritto il processo di integrazione tra ambiente simulato e sistema di monitoraggio, evidenziando le scelte progettuali adottate e le relative motivazioni.

Nel Capitolo 4 viene descritta l'implementazione pratica del sistema. In tale sezione si illustrano le modalità di controllo del movimento del robot, l'integrazione dei sensori virtuali per la percezione dell'ambiente e lo sviluppo del Digital Twin per la raccolta e la visualizzazione dei dati. Viene inoltre approfondito il meccanismo di acquisizione dei parametri di stato e la logica utilizzata per la loro elaborazione.

Nei Capitoli 5 e 6 è presentata la fase di validazione sperimentale. Sono definiti gli scenari di prova adottati e le metriche utilizzate per l'analisi quantitativa delle prestazioni del robot. In particolare, vengono presi in considerazione il tempo di percorrenza, la lunghezza della traiettoria, il numero di collisioni e un indice di consumo energetico stimato, calcolato sulla base dell'attività dinamica del robot durante la navigazione. I risultati ottenuti vengono analizzati e discussi al fine di valutare l'efficacia del sistema sviluppato.

Infine, il Capitolo 7 riporta le conclusioni del lavoro, sintetizzando i risultati raggiunti e mettendo in evidenza i principali limiti riscontrati. Vengono inoltre delineati possibili sviluppi futuri, sia in ambito simulativo sia in prospettiva di un'implementazione su piattaforma reale.

1.4 Metodologia di sviluppo

Il lavoro è stato sviluppato seguendo un approccio progressivo e modulare, articolato in diverse fasi successive, con l'obiettivo di garantire coerenza tra progettazione teorica e implementazione pratica.

In una prima fase è stato analizzato il contesto teorico di riferimento, con particolare attenzione ai modelli di robot mobile a cinematica differenziale e all'architettura del framework ROS. Questa fase ha consentito di definire le basi concettuali necessarie alla progettazione del sistema.

Successivamente è stata realizzata l'architettura software del robot in ambiente simulato, utilizzando ROS per la gestione dei nodi e Gazebo per la modellazione

fisica e l'interazione con l'ambiente virtuale. In tale fase sono stati implementati i meccanismi di controllo del movimento e di acquisizione dei dati di stato.

Una volta stabilizzato il comportamento del robot in simulazione, è stato sviluppato il Digital Twin, concepito come modello digitale in grado di ricevere e rappresentare in tempo reale i parametri operativi del sistema [7]. L'integrazione tra simulazione e modello digitale è stata realizzata tramite la sottoscrizione ai topic ROS contenenti le informazioni di posizione e velocità.

Infine, è stata condotta una fase di validazione sperimentale, durante la quale sono stati definiti i parametri degli scenari di testing e sono state raccolte le metriche di interesse. L'analisi dei dati ha permesso di valutare le prestazioni del sistema e di verificare la coerenza tra comportamento simulato e modello digitale.

L'approccio adottato privilegia semplicità, chiarezza architetturale e replicabilità, con l'obiettivo di sviluppare un sistema funzionale e analizzabile in modo quantitativo.

In aggiunta, il Digital Twin viene impiegato come strumento di supporto a un processo di ottimizzazione parametrica iterativa della navigazione. In particolare, a partire dai dati raccolti in ciascuna esecuzione, vengono calcolate metriche quantitative e una funzione obiettivo sintetica; successivamente, uno o più parametri di navigazione (ad esempio la velocità massima o i parametri del controllo) vengono aggiornati e il test ripetuto. Tale approccio consente di confrontare configurazioni alternative in modo sistematico, mantenendo una struttura semplice e replicabile all'interno dell'ambiente simulato.

Il fine di tale applicazione è dimostrare la validità del Digital Twin come prezioso alleato nell'ottimizzazione della navigazione del nostro robot mobile.

Capitolo 2

Richiami teorici

2.1 Robot mobile a cinematica differenziale

I robot mobili a cinematica differenziale rappresentano una delle configurazioni più diffuse nell'ambito della robotica mobile terrestre, grazie alla loro semplicità costruttiva, facilità di controllo e buona manovrabilità in ambienti strutturati.
Questa architettura prevede due ruote motrici indipendenti, montate sullo stesso asse, e una o più ruote passive di supporto per garantire la stabilità del sistema. Il movimento del robot è ottenuto variando in modo indipendente la velocità angolare delle due ruote motrici. [1]
Nel presente lavoro viene considerata una piattaforma ispirata al **TurtleBot3**, robot educativo e di ricerca ampiamente utilizzato in ambienti ROS per sperimentazione e sviluppo di algoritmi di navigazione.

Modello cinematico. [1],[11]
Il modello cinematico descrive il legame tra le velocità delle ruote e il moto del robot nello spazio, senza considerare gli effetti dinamici quali masse, inerzie o forze esterne.
Si consideri un sistema di riferimento globale (X, Y)e un sistema di riferimento locale solidale al robot (x, y), con orientamento definito dall'angolo θ.

Siano:

- r il raggio delle ruote
- L la distanza tra le due ruote
- ω_r la velocità angolare della ruota destra
- ω_l la velocità angolare della ruota sinistra

La velocità lineare del centro del robot risulta:

$$v = \frac{r}{2}(\omega_r + \omega_l)$$

La velocità angolare attorno all'asse verticale è invece:

$$\omega = \frac{r}{L}(\omega_r - \omega_l)$$

Le equazioni del moto nello spazio cartesiano sono quindi:

$$\dot{x} = v \cos \theta$$
$$\dot{y} = v \sin \theta$$
$$\dot{\theta} = \omega$$

Queste relazioni costituiscono il modello cinematico non lineare del robot a

trazione differenziale.

Interpretazione del moto

Il comportamento del robot dipende dal rapporto tra le velocità delle due ruote:

- Se $\omega_r = \omega_l$, il robot si muove in linea retta.
- Se $\omega_r = -\omega_l$, il robot ruota su sé stesso.
- Se $\omega_r \neq \omega_l$, il robot segue una traiettoria circolare.

Il raggio di curvatura della traiettoria è dato da:

$$R = \frac{v}{\omega}$$

Questo parametro è fondamentale nella progettazione di algoritmi di navigazione, poiché determina la capacità del robot di eseguire manovre in spazi ristretti.

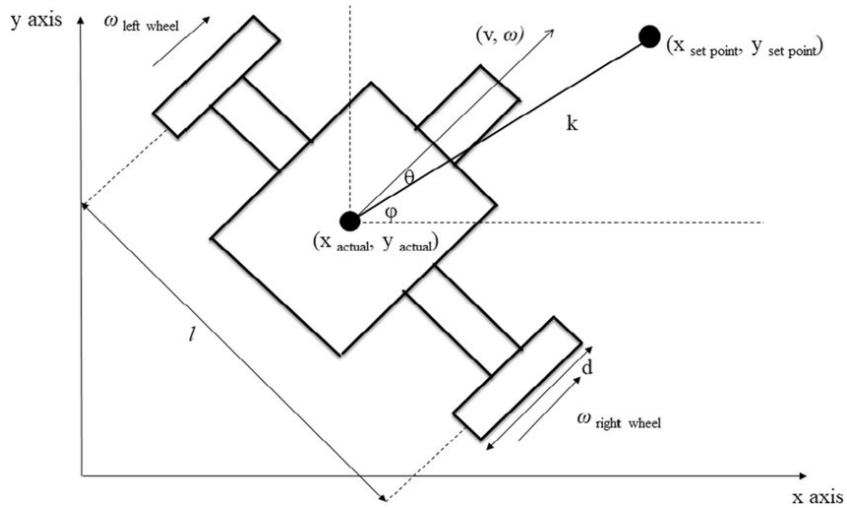
Vincolo di non olonomia

Il robot a cinematica differenziale è un sistema non olonomo, ovvero soggetto a un vincolo cinematico che limita il moto laterale.
In particolare, il robot non può muoversi direttamente lungo l'asse laterale y del sistema solidale al corpo. Tale vincolo può essere espresso come:

$$\dot{y}_{body} = 0$$

Questo aspetto ha implicazioni importanti nella pianificazione del percorso,

poiché non è possibile raggiungere una configurazione arbitraria con un singolo movimento elementare, ma è necessario comporre sequenze di rotazioni e traslazioni.



Schema cinematico di un robot a trazione differenziale con sistema di riferimento solidale al robot.

Collegamento con la simulazione

Il modello cinematico descritto costituisce la base teorica per l'implementazione del controllo del robot in ambiente simulato.

Nell'ambito della simulazione con ROS e Gazebo, le velocità lineari e angolari vengono tipicamente pubblicate su specifici topic e tradotte dal modello in movimento del robot nell'ambiente virtuale.

La corretta comprensione di tali equazioni è essenziale per:

- progettare il controllo del movimento;
- analizzare le traiettorie generate;
- implementare strategie di ottimizzazione nel Digital Twin.

Modello dinamico semplificato

Sebbene il controllo implementato nel presente lavoro si basi prevalentemente su un modello cinematico, è opportuno introdurre una formulazione dinamica semplificata al fine di evidenziare i limiti del modello puramente geometrico.

Nel modello dinamico si considerano le grandezze fisiche del sistema, in particolare:

- m massa totale del robot
- I_z momento d'inerzia rispetto all'asse verticale
- τ_r, τ_l coppie applicate alle ruote
- r raggio delle ruote
- L distanza tra le ruote

Si assumono:

- assenza di slittamento laterale
- assenza di attrito volvente
- nessuna perdita di energia
- risposta ideale degli attuatori

Equazioni del moto

La forza totale lungo la direzione di avanzamento è data da:

$$F = \frac{\tau_r + \tau_l}{r}$$

Applicando la seconda legge di Newton lungo l'asse longitudinale:

$$m\dot{v} = \frac{\tau_r + \tau_l}{r}$$

Per la rotazione attorno all'asse verticale:

$$I_z\dot{\omega} = \frac{L}{2r}(\tau_r - \tau_l)$$

Si ottiene quindi il sistema dinamico:

$$\dot{v} = \frac{1}{mr}(\tau_r + \tau_l)$$

$$\dot{\omega} = \frac{L}{2I_z r}(\tau_r - \tau_l)$$

Forma in spazio di stato

Definendo lo stato del sistema come:

$$x = [v\omega]^T$$

e l'ingresso come:

$$u = [\tau_r \tau_l]^T$$

si può esprimere il modello in forma compatta:

$$\dot{x} = Bu$$

dove B è la matrice dei coefficienti dinamici dipendente dai parametri fisici del robot.

Questa formulazione evidenzia come, a differenza del modello cinematico, la velocità non possa variare istantaneamente ma sia soggetta a dinamica inerziale.

Implicazioni per il controllo

Il modello cinematico assume variazioni istantanee di velocità, risultando adeguato per:

- pianificazione geometrica;
- controllo a bassa velocità;
- simulazioni ideali.

Il modello dinamico, seppur semplificato, evidenzia invece:

- presenza di accelerazioni finite;
- dipendenza del moto dalle coppie applicate;
- influenza dei parametri fisici del sistema.

Nel contesto della presente tesi, la simulazione in ***Gazebo*** utilizza un motore fisico che tiene conto delle grandezze dinamiche. Tuttavia, l'algoritmo di controllo e il Digital Twin si basano su una rappresentazione cinematica, scelta coerente con l'obiettivo di ottimizzazione parametrica della navigazione.

2.1 Architettura ROS

Il ***Robot Operating System*** (ROS) è un middleware open source progettato per facilitare lo sviluppo di applicazioni robotiche modulari e distribuite. [2]

Non si tratta di un sistema operativo nel senso tradizionale del termine, si pone come *middleware* che opera su un sistema operativo già esistente, più specificatamente di un'infrastruttura software che fornisce strumenti, librerie e convenzioni per la comunicazione tra processi dedicati al controllo e alla gestione di sistemi robotici.

Nel presente lavoro ROS costituisce il livello di coordinamento tra:

- il robot simulato in Gazebo;
- i nodi di controllo del movimento;
- il modulo Digital Twin;
- i sistemi di acquisizione e analisi dei dati.

Struttura a nodi

L'architettura ROS è basata su un modello a grafo distribuito.

Ogni componente del sistema è implementato come **nodo**, ovvero un processo indipendente che svolge una funzione specifica, ad esempio:

- generazione di comandi di velocità;
- lettura dei sensori virtuali;
- elaborazione dati;
- monitoraggio dello stato del robot.

Questa suddivisione favorisce:

- modularità;
- riusabilità del codice;
- scalabilità del sistema.

Comunicazione tramite topic

La comunicazione tra nodi avviene principalmente attraverso i **topic**, canali asincroni basati sul paradigma publish–subscribe.

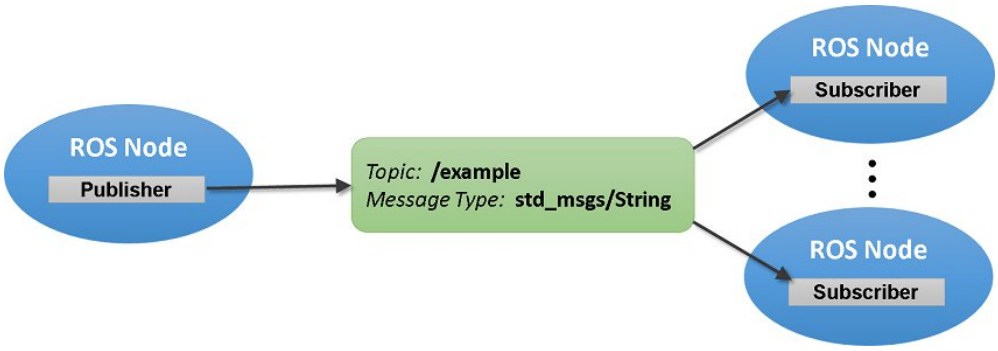
Un nodo può:

- pubblicare (publisher) messaggi su un topic;
- sottoscriversi (subscriber) ad un topic per riceverne i dati.

Ad esempio, nel controllo del robot mobile:

- un nodo pubblica comandi di velocità lineare e angolare;
- il simulatore riceve tali comandi e aggiorna lo stato del robot;
- un altro nodo può sottoscriversi ai dati di odometria per monitorarne la posizione.

Questo modello elimina la necessità di connessioni dirette punto-punto tra moduli, riducendo l'accoppiamento tra componenti.



Rappresentazione semplificata del grafo computazionale di ROS basato su nodi e topic.

Servizi e parameter server

Oltre ai topic, ROS mette a disposizione:

- **Service**, meccanismo di comunicazione sincrona basato su richiesta-risposta;
- **Parameter Server**, archivio centralizzato di parametri configurabili.

Nel contesto della presente tesi, il Parameter Server può essere utilizzato per:

- definire parametri cinematici (raggio ruote, distanza tra le ruote);
- impostare velocità massime;
- modificare soglie di controllo senza alterare il codice sorgente.

2.3 Simulazione con Gazebo

La simulazione rappresenta uno strumento fondamentale nello sviluppo di sistemi robotici, poiché consente di progettare, testare e validare algoritmi di controllo senza l'utilizzo immediato di hardware fisico, potendo così sviluppare la fase preliminare in un ambiente esente da rischi fisici.

Nel contesto di questa tesi, Gazebo viene utilizzato come ambiente di simulazione per modellare un robot mobile a cinematica differenziale e il relativo scenario operativo.

Gazebo è un simulatore robotico open-source che integra [4]:

- Motore fisico realistico (dinamica, attriti, collisioni)
- Rendering 3D dell'ambiente
- Simulazione di sensori (Lidar, IMU, telecamere, encoder)
- Integrazione nativa con Robot Operating System (ROS)

L’obiettivo è riprodurre in modo coerente il comportamento dinamico che il robot avrebbe in un ambiente reale.

Struttura della simulazione

In Gazebo, la simulazione si basa su tre elementi principali:

1. **Modello del robot**
   Descritto tramite file URDF o SDF, che definiscono:
   - Geometria
   - Masse e inerzie
   - Giunti
   - Attuatori
2. **Ambiente (World)**
   Include:
   - Superficie di appoggio
   - Ostacoli
   - Elementi statici o dinamici
   - Parametri fisici globali (gravità, attrito)
3. **Plugin di integrazione ROS**
   Consentono lo scambio di dati tra simulatore e nodi ROS tramite topic, servizi e parametri.

Questa architettura permette di separare la parte fisica simulata dalla logica di controllo implementata nel software.

Simulazione dei sensori

Uno degli aspetti più rilevanti è la possibilità di simulare sensori virtuali, come:

- Lidar 2D per rilevamento ostacoli
- Sensori di distanza
- Odometro
- IMU

I dati generati vengono pubblicati su topic ROS, esattamente come accadrebbe in un sistema reale. Questo consente di sviluppare e testare algoritmi di navigazione e evitamento ostacoli senza modifiche strutturali nel passaggio dalla simulazione all'hardware.

Naturalmente, la simulazione introduce semplificazioni: rumore limitato, condizioni ideali, assenza di usura meccanica, ma per lo sviluppo e la validazione preliminare è più che adeguata.

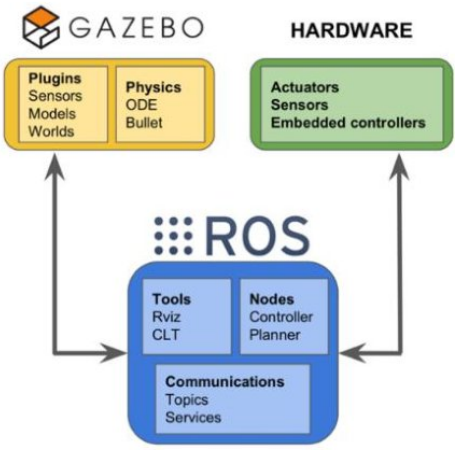
Vantaggi

L'utilizzo di Gazebo nel presente lavoro offre diversi benefici:

- Riduzione dei costi hardware

- Possibilità di test ripetibili in condizioni controllate
- Raccolta sistematica dei dati
- Integrazione diretta con il Digital Twin

In particolare, la simulazione permette di generare flussi di dati continui relativi a posizione, velocità e collisioni, che saranno successivamente elaborati nel modello di Digital Twin per l'analisi e l'ottimizzazione della navigazione.



Integrazione tra ambiente di simulazione Gazebo e middleware ROS.

Limiti della simulazione

È importante evidenziare che una simulazione, per quanto accurata, non sostituisce completamente la sperimentazione reale.

Le principali criticità riguardano:

- Differenze nei parametri fisici reali

- Modellazione semplificata dell'attrito
- Assenza di incertezze ambientali complesse

Tuttavia, per un progetto focalizzato sulla progettazione software e sull'analisi dati, l'ambiente simulato costituisce una piattaforma adeguata e coerente con gli obiettivi della tesi.

2.4 Concetto di Digital Twin

Il concetto di Digital Twin nasce in ambito industriale come evoluzione dei sistemi cyber-fisici e consiste nella creazione di una replica digitale dinamica di un sistema reale.

Si tratta quindi di un'entità virtuale capace di ricevere dati dal sistema fisico, elaborarli e restituire informazioni utili per monitoraggio, analisi e ottimizzazione [8].

Il Digital Twin può essere definito come:

"Una rappresentazione digitale sincronizzata di un sistema fisico, connessa tramite flussi di dati bidirezionali, in grado di riprodurne comportamento e stato operativo nel tempo." [7]

Il concetto è stato formalizzato in ambito accademico e industriale, trovando applicazione in settori come aerospazio, manifattura e automazione.

Un sistema basato su Digital Twin è composto da tre elementi principali:

1. Sistema fisico

   Nel caso di questa tesi: il robot mobile simulato in Gazebo.

2. Modello digitale

   Replica computazionale che riceve dati di stato (posizione, velocità, collisioni, ecc.).

3. Connessione dati

   Canale di comunicazione che permette la sincronizzazione continua tra sistema e modello.

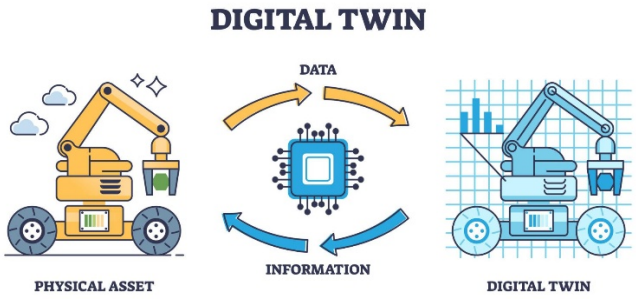
Nel contesto sviluppato in questo lavoro, la connessione avviene tramite i topic ROS, che trasmettono informazioni in tempo reale dal simulatore al modello di analisi.

Differenza tra simulazione e Digital Twin

Facciamo un'opportuna distinzione tra i due concetti:

- Simulazione: ambiente virtuale che riproduce un sistema per testarne il comportamento.
- Digital Twin: modello digitale connesso che analizza lo stato del sistema e può contribuire all'ottimizzazione delle prestazioni.

La simulazione è lo spazio operativo, Il Digital Twin è il livello analitico sopra di essa.



Schema concettuale di Digital Twin con flusso bidirezionale tra sistema reale e modello digitale.

Nel presente lavoro, il Digital Twin raccoglie dati di navigazione per:

- Valutare prestazioni
- Individuare inefficienze
- Supportare un'ottimizzazione parametrica iterativa

Esistono diversi livelli di complessità di un Digital Twin:

- Livello descrittivo (monitoraggio)
- Livello diagnostico (analisi delle cause)
- Livello predittivo (previsione comportamento)
- Livello prescrittivo (ottimizzazione automatica)

La presente tesi si colloca tra il livello descrittivo e quello prescrittivo di base, poiché il modello digitale verrà utilizzato per analizzare le prestazioni della navigazione e contribuire alla regolazione dei parametri di controllo.

Nel sistema sviluppato:

Gazebo genera i dati dinamici del robot, ROS trasmette i dati, il Digital Twin li elabora ed i risultati influenzano l'ottimizzazione dei parametri di navigazione.

Si crea quindi un ciclo di miglioramento iterativo, in cui il modello digitale contribuisce a perfezionare il comportamento del sistema simulato.

Capitolo 3

Progettazione del sistema

3.1 TurtleBot3

Il sistema sviluppato nel presente lavoro fa riferimento alla piattaforma robotica TurtleBot3, una delle soluzioni open-source più diffuse nell'ambito della robotica mobile accademica e della ricerca applicata [5].

La scelta di tale piattaforma deriva da una combinazione di fattori tecnici e metodologici. Il TurtleBot3 rappresenta infatti un compromesso efficace tra semplicità strutturale, flessibilità software e ampia compatibilità con l'ecosistema **ROS**, risultando particolarmente adatto a sperimentazioni in ambiente simulato tramite **Gazebo**.

Dal punto di vista progettuale, l'utilizzo di una piattaforma standardizzata consente di:

- evitare la progettazione hardware da zero
- ridurre la complessità del modello
- concentrarsi sugli aspetti di controllo e Digital Twin
- garantire replicabilità del lavoro

Architettura meccanica

Il TurtleBot3 è un robot mobile a cinematica differenziale, configurazione tra le più comuni nella robotica planare. La struttura meccanica è composta da [5]:

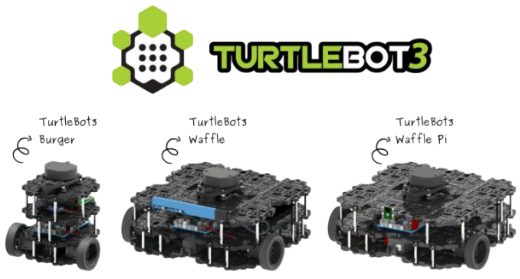
- due ruote motrici indipendenti posizionate lateralmente
- una ruota passiva di supporto (caster)

- telaio modulare compatto
- sensore Lidar montato superiormente

La locomozione avviene mediante variazione indipendente delle velocità angolari delle due ruote. Tale configurazione consente:

- movimento rettilineo (velocità uguali)
- rotazione sul posto (velocità opposte)
- traiettorie curve (velocità differenti)

Questa soluzione presenta vantaggi evidenti in termini di semplicità nella costruzione di un modello cinematico valido, permette altresì un controllo diretto della velocità lineare e angolare e vanta una grande compatibilità con modelli matematici consolidati.
Il sistema non è omnidirezionale e non può muoversi lateralmente senza rotazione. Tuttavia, tale limitazione è coerente con l'obiettivo del lavoro e non costituisce un vincolo penalizzante nel contesto simulato considerato.



Famiglia TurtleBot3

Modello cinematico adottato

Nel presente lavoro il robot viene descritto attraverso un modello cinematico planare semplificato. Lo stato del sistema è definito dal vettore:

$$\mathbf{x} = [x, y, \theta]$$

dove:

- x e y rappresentano la posizione nel piano cartesiano globale
- θ rappresenta l'orientamento rispetto all'asse x globale

Le equazioni cinematiche del modello differenziale assumono la forma:

$$\dot{x} = v \cos \theta$$
$$\dot{y} = v \sin \theta$$
$$\dot{\theta} = \omega$$

dove:

- v è la velocità lineare del robot
- ω è la velocità angolare

Tali grandezze sono ricavate a partire dalle velocità angolari delle ruote destra e sinistra, secondo le relazioni classiche della cinematica differenziale.
È importante sottolineare che in questa fase progettuale si è scelto di non includere:

- effetti di attrito volvente
- slittamento delle ruote
- dinamica dei motori
- ritardi di attuazione

L’introduzione di un modello dinamico completo avrebbe aumentato la complessità senza apportare benefici proporzionati agli obiettivi preposti, si è privilegiata la controllabilità del modello rispetto ad un realismo estremo.

Sensori e capacità percettive

Il TurtleBot3 è equipaggiato, nella configurazione standard simulata, con un sensore Lidar 2D (LDS-01), utilizzato per [6]:

- rilevamento ostacoli
- costruzione di mappe locali
- supporto agli algoritmi di navigazione

In ambiente simulato, il Lidar produce una scansione angolare del piano circostante, generando una serie di misure di distanza rispetto agli oggetti presenti nell’ambiente virtuale.

Accanto al Lidar, il sistema utilizza:

- odometria stimata
- trasformazioni di coordinate tramite tf

Il pacchetto ***tf*** di ROS consente la gestione dinamica delle trasformazioni tra i diversi sistemi di riferimento del robot [3]. Attraverso una struttura ad albero di frame, è possibile mantenere aggiornate in tempo reale le relazioni spaziali tra base del robot, sensori e sistema di riferimento globale.

Questi dati costituiscono l’insieme informativo su cui opera il modulo di Digital Twin, che non interviene sulla fisica del robot ma sull’analisi delle sue prestazioni.

3.2 Architettura software del sistema

L'architettura software sviluppata nel presente lavoro è basata su **ROS**, framework middleware progettato per la gestione modulare di sistemi robotici complessi. [2] La scelta di ROS consente di integrare simulazione, controllo e Digital Twin in un'unica struttura comunicante e scalabile.

ROS, come già illustrato nel capitolo precedente, adotta un'architettura distribuita fondata sul concetto di nodi indipendenti che comunicano attraverso un sistema publish /subscribe basato su topic. Questo paradigma consente di separare logicamente le funzionalità del sistema, evitando accoppiamenti rigidi tra i moduli. In altre parole, ogni componente svolge un compito specifico, ma rimane sostituibile o modificabile senza compromettere l'intera architettura.

Nel caso specifico, il sistema è stato strutturato distinguendo quattro macro-funzioni:

- controllo del movimento
- simulazione fisica
- acquisizione dati sensoriali
- modulo Digital Twin

Il nodo di controllo è responsabile della generazione dei comandi di velocità lineare e angolare, pubblicati sul topic ***/cmd_vel***. Tali comandi vengono recepiti dall'interfaccia di simulazione in **Gazebo**, che aggiorna lo stato del robot secondo il modello cinematico implementato.

Gazebo si occupa esclusivamente della componente fisica simulata: gestione delle collisioni, aggiornamento della posa, generazione dei dati dei sensori virtuali. La simulazione non contiene alcuna logica di ottimizzazione o analisi prestazionale.

Questa distinzione è progettualmente rilevante, perché consente di separare il livello “fisico” da quello “analitico”.

I dati prodotti dall’ambiente simulato, in particolare odometria e scansioni Lidar, vengono pubblicati su specifici topic ROS e resi disponibili agli altri nodi del sistema. Qui interviene il modulo di Digital Twin, che si configura come entità software separata incaricata di acquisire lo stato del robot, elaborarlo e calcolare metriche di prestazione.

La gestione delle trasformazioni tra i diversi sistemi di riferimento è affidata al pacchetto *tf*, che mantiene aggiornate le relazioni spaziali tra il frame globale, il frame odometrico e il frame solidale al robot [3]. Questo aspetto è essenziale per garantire coerenza geometrica nell’analisi dei dati, evitando incongruenze tra posizione stimata e percezione sensoriale.

Dal punto di vista logico, l’architettura realizzata può essere interpretata come un sistema a livelli:

1. livello fisico simulato (Gazebo)
2. livello di comunicazione e coordinamento (ROS)
3. livello decisionale e analitico (Digital Twin)

Questa suddivisione consente di mantenere chiarezza concettuale e facilita eventuali estensioni future verso un robot reale, poiché l’interfaccia tra controllo e analisi rimane indipendente dalla natura fisica o simulata della piattaforma.

Una scelta progettuale rilevante è stata quella di evitare soluzioni monolitiche. Integrare controllo, simulazione e analisi in un unico nodo avrebbe ridotto la complessità apparente, ma avrebbe compromesso la modularità e la leggibilità del sistema. Nel seguente lavoro si è quindi preferito dare priorità alla “trasparenza” dell’architettura proposta.

In sintesi, l’architettura software sviluppata realizza un sistema modulare, scalabile e aderente agli obiettivi. La separazione tra simulazione fisica e Digital Twin consente di isolare il processo di ottimizzazione dalla dinamica del modello,

rendendo l'analisi delle prestazioni più chiara e metodologicamente controllabile.

3.3 Flusso dei dati tra simulazione e Digital Twin

L'elemento centrale del sistema sviluppato è il flusso informativo che collega l'ambiente simulato al modulo di Digital Twin, trattasi di un processo strutturato che realizza un ciclo chiuso di osservazione, analisi e adattamento parametrico.
Il robot simulato in **Gazebo** riceve comandi di velocità dal nodo di controllo tramite l'infrastruttura di comunicazione fornita da **ROS**. A ogni iterazione temporale, il simulatore aggiorna la posa del robot e genera i dati dei sensori virtuali, in particolare odometria e scansioni Lidar.
Queste informazioni vengono pubblicate su topic dedicati e rese disponibili al modulo di Digital Twin, così che il sistema possa effettivamente tradursi in un modello analitico osservabile.

Struttura del ciclo informativo

Il flusso può essere descritto come una sequenza iterativa:

1. Il nodo di controllo genera un comando di velocità (v, ω).
2. Il simulatore aggiorna lo stato del robot secondo il modello cinematico.
3. I sensori virtuali producono misure coerenti con la nuova configurazione spaziale.
4. Il Digital Twin acquisisce posizione, velocità e dati ambientali.
5. Vengono calcolate metriche di prestazione.
6. Se previsto, alcuni parametri di controllo vengono aggiornati.
7. Il ciclo ricomincia.

Questa struttura realizza un sistema ad anello chiuso in ambiente simulato. Tuttavia, è importante distinguere tra controllo cinematico e ottimizzazione parametrica: il Digital Twin non sostituisce il controllore di base, ma opera a un livello superiore, influenzandone i parametri.

Formalizzazione del processo

Indicando con:

$$\mathbf{x}(t) = [x(t), y(t), \theta(t)]$$

lo stato del robot e con:

$$\mathbf{u}(t) = [v(t), \omega(t)]$$

il vettore di controllo, la simulazione implementa l'evoluzione:

$$\mathbf{x}(t+1) = f(\mathbf{x}(t), \mathbf{u}(t))$$

Il Digital Twin riceve la sequenza di stati osservati e costruisce un insieme di indicatori prestazionali J, funzione dello storico del moto:

$$J = g(\mathbf{x}(0 \ldots T), \mathbf{u}(0 \ldots T))$$

Nell'analisi effettuata tali metriche includono:

- tempo totale di percorrenza

- lunghezza della traiettoria
- numero di collisioni o quasi-collisioni
- deviazione dalla traiettoria desiderata

L'eventuale aggiornamento dei parametri di controllo può essere descritto come:

$$\mathbf{p}_{k+1} = h(\mathbf{p}_k, J)$$

dove $\mathbf{p}$ rappresenta l'insieme dei parametri regolabili (ad esempio guadagni o soglie decisionali).
In questa formulazione il Digital Twin assume il ruolo di osservatore e regolatore di secondo livello, senza intervenire direttamente sulla dinamica fisica simulata.

Un aspetto progettuale di grande rilievo è la netta separazione tra:

- livello fisico (simulazione dinamica in Gazebo)
- livello informativo (raccolta e distribuzione dati in ROS)
- livello analitico (Digital Twin)

Il simulatore rappresenta il "mondo virtuale", con le sue leggi fisiche e le sue interazioni.
Il Digital Twin, invece, rappresenta una replica logica del sistema, focalizzata sull'interpretazione dei dati e sulla valutazione delle prestazioni.
Questa distinzione evita sovrapposizioni di responsabilità e mantiene il sistema metodologicamente pulito. Il Twin non altera direttamente la fisica, ma modifica parametri decisionali che influenzano il comportamento futuro del robot.

Il ciclo realizzato è un'espressione intrinseca del concetto stesso di Digital Twin

applicato alla robotica mobile: un'entità software che osserva il sistema, ne misura le prestazioni e contribuisce al miglioramento iterativo del comportamento.
In sostanza, il comportamento del robot comportamento viene analizzato, valutato e progressivamente raffinato.
Ed è proprio questa retroazione informativa a orientare la simulazione proposta verso il processo di ottimizzazione.

E' importante soffermarsi anche sulla sincronizzazione temporale tra i diversi moduli software. Nel sistema sviluppato, simulazione fisica, controllo e Digital Twin operano con frequenze di aggiornamento potenzialmente differenti. Questa distinzione è rilevante, poiché influenza la stabilità e la coerenza del ciclo di ottimizzazione.
Il simulatore **Gazebo** aggiorna lo stato del robot con una frequenza interna elevata (tipicamente dell'ordine di decine o centinaia di Hz), garantendo continuità nella dinamica del modello. I comandi di controllo pubblicati tramite **ROS** vengono applicati secondo la frequenza del nodo di controllo, generalmente inferiore rispetto alla dinamica fisica del simulatore.
Il modulo di Digital Twin, invece, non necessita di operare alla stessa frequenza della simulazione, pur operando con dinamiche compatibili con il tempo reale. L'analisi prestazionale e l'eventuale aggiornamento parametrico avvengono quindi su finestre temporali più ampie, al fine di:

- evitare reazioni eccessivamente rapide a variazioni locali
- ridurre la sensibilità al rumore sensoriale
- mantenere stabilità nel processo di ottimizzazione

Indicando con:

- f_s la frequenza della simulazione
- f_c la frequenza del controllo

- f_{DT} la frequenza di aggiornamento del Digital Twin

si assume la relazione:

$$f_s \geq f_c > f_{DT}$$

Questa gerarchia temporale garantisce che il Digital Twin operi come regolatore di livello superiore, intervenendo su parametri globali del comportamento piuttosto che sulle singole iterazioni cinematiche.
Dal punto di vista ingegneristico, tale scelta riduce il rischio di instabilità dovute a retroazioni troppo rapide e mantiene separato il controllo locale del moto dall'ottimizzazione globale delle prestazioni.
In sintesi, il sistema implementa una struttura multilivello non solo dal punto di vista funzionale, ma anche temporale. La dinamica simulata evolve in modo continuo, il controllo agisce su scala intermedia e il Digital Twin interviene con una frequenza più lenta, coerente con il suo ruolo analitico.

Capitolo 4

Implementazione

4.1 Controllo del movimento del robot

L’implementazione del controllo del movimento rappresenta il primo passo concreto nello sviluppo del sistema. Dopo aver definito nel Capitolo 2 il modello cinematico del robot a trazione differenziale, in questa fase tale modello viene tradotto in un comportamento effettivo all’interno dell’ambiente simulato.
Il robot utilizzato è il TurtleBot3, simulato in ambiente Gazebo e gestito tramite ROS. Il controllo del moto si basa sull’invio di comandi di velocità lineare e angolare al *topic /cmd_vel*, secondo la convenzione standard di ROS. Tali comandi sono espressi nel formato *geometry_msgs/Twist*, che consente di definire la velocità lungo l’asse x e la velocità angolare attorno all’asse z del sistema di riferimento del robot.
Dal punto di vista implementativo, il movimento è stato inizialmente testato tramite un nodo di teleoperazione, che pubblica comandi di velocità a frequenza costante. Questo approccio permette di validare rapidamente il corretto collegamento tra il livello di comando e il modello dinamico simulato in Gazebo, senza introdurre logiche di controllo complesse.
La generazione del moto rispetta il modello cinematico differenziale illustrato nel paragrafo 2.1. In particolare:

- se $\omega = 0$ e $v \neq 0$ il robot si muove lungo una traiettoria rettilinea;
- se $v = 0$ e $\omega \neq 0$ il robot ruota attorno al proprio asse verticale;
- se $v \neq 0$ e $\omega \neq 0$ la traiettoria risultante è circolare.

La coerenza tra teoria e simulazione è stata verificata osservando il comportamento del robot in diversi scenari di movimento elementare: avanzamento lineare, rotazione sul posto e traiettorie curve. I test hanno confermato che la traiettoria simulata risulta coerente con le equazioni cinematiche e che non si verificano instabilità o oscillazioni indesiderate nel controllo di base.

Per garantire la tracciabilità del comportamento del sistema, durante le prove di movimento sono stati registrati i principali topic ROS coinvolti nel controllo e nella percezione del robot, tra cui */cmd_vel*, */odom*, */scan* e */tf*. La registrazione è stata effettuata mediante *rosbag*, consentendo di salvare l'evoluzione temporale dei comandi inviati e dello stato stimato del robot. Questa fase risulta fondamentale in prospettiva Digital Twin, poiché fornisce i dati necessari per l'analisi offline del comportamento del sistema e per la successiva costruzione del modello digitale.
L'adozione di un controllo inizialmente semplice, privo di regolatori avanzati o strategie di navigazione autonoma, è stata una scelta metodologica consapevole. Tale approccio consente di isolare il livello cinematico di base, verificandone la correttezza prima di introdurre livelli superiori di complessità, come l'evitamento ostacoli o l'ottimizzazione tramite Digital Twin.
La validazione del controllo di base costituisce quindi il fondamento dell'intero sistema: eventuali errori a questo livello si propagherebbero inevitabilmente agli strati superiori, compromettendo l'affidabilità delle analisi successive.

4.2 Integrazione dei sensori virtuali

Dopo la validazione del controllo del movimento, il passo successivo nello sviluppo del sistema è stato l'integrazione e la verifica dei sensori virtuali all'interno dell'ambiente simulato. Nel modello TurtleBot3 utilizzato in Gazebo, il principale sensore di percezione è un LIDAR bidimensionale montato frontalmente, che consente la rilevazione della distanza dagli ostacoli nel piano orizzontale.
Il sensore pubblica i dati sul topic /scan nel formato *sensor_msgs/LaserScan [10]*. Ogni messaggio contiene un insieme di misurazioni distribuite angolarmente e rappresentate attraverso i seguenti parametri principali:

- *angle_min* e *angle_max*, che definiscono l'intervallo angolare coperto dal sensore;
- *angle_increment*, che specifica la risoluzione angolare tra due misure consecutive;
- *ranges*, vettore contenente le distanze misurate lungo ciascun raggio;
- *range_min* e *range_max*, che definiscono i limiti operativi del sensore.

Questa struttura consente di ricostruire una rappresentazione bidimensionale dell'ambiente circostante, fornendo una mappa locale delle distanze rispetto agli ostacoli. In ambiente simulato, il comportamento del LIDAR è gestito da un plugin Gazebo integrato nel modello URDF del robot, che riproduce il comportamento ideale del sensore senza introdurre rumore significativo o distorsioni sistematiche.
Parallelamente, il *topic /odom* fornisce la stima della posa del robot nel piano, espressa in termini di posizione (x, y) e orientamento (yaw). I dati di odometria derivano dall'integrazione del modello cinematico differenziale simulato e rappresentano la stima dello stato interno del sistema. Tali informazioni risultano fondamentali per correlare le misure percettive con la configurazione del robot nello spazio.
Un ulteriore elemento chiave è rappresentato dal *topic /tf*, che gestisce le trasformazioni tra i diversi frame di riferimento (ad esempio *odom*, *base_link*, *laser*). Il corretto funzionamento della catena di trasformazioni è essenziale per garantire la coerenza spaziale tra dati di percezione e stato del robot. Durante le prove è stata verificata la corretta pubblicazione delle trasformazioni, assicurando che non vi fossero interruzioni o incongruenze nei frame. [3]
Dal punto di vista sperimentale, l'integrazione dei sensori è stata validata attraverso:

- monitoraggio dei topic mediante strumenti ROS di ispezione;
- verifica della frequenza di pubblicazione dei messaggi;

- osservazione della variazione dei valori di distanza in presenza di ostacoli nel campo visivo del LIDAR;
- controllo della coerenza tra movimento del robot e variazione delle letture sensoriali.

Le prove effettuate in Gazebo hanno confermato che il sistema percettivo risponde in modo coerente agli spostamenti del robot e alla configurazione dell'ambiente simulato. In particolare, durante il movimento verso un ostacolo, si è osservata una riduzione progressiva dei valori contenuti nel vettore *ranges*, mentre le rotazioni sul posto producevano una variazione angolare coerente nella distribuzione delle distanze rilevate.
Per consentire analisi successive e garantire la riproducibilità degli esperimenti, durante le sessioni di prova sono stati registrati simultaneamente i *topic /scan*, */odom*, */cmd_vel* e */tf* mediante *rosbag*. Questa registrazione ha permesso di ottenere una base dati temporizzata contenente:

- comandi inviati al robot,
- stato stimato tramite odometria,
- misure percettive del LIDAR,
- trasformazioni tra i sistemi di riferimento.

La sincronizzazione temporale tra tali flussi informativi rappresenta un elemento centrale nell'architettura proposta. Infatti, la costruzione del Digital Twin richiede una corrispondenza coerente tra ingresso (comandi), stato interno e percezione ambientale. L'utilizzo di rosbag consente di preservare tale coerenza e di riprodurre offline l'intero esperimento, rendendo possibile l'analisi dettagliata del comportamento del sistema.
L'integrazione dei sensori virtuali costituisce quindi il ponte tra il livello puramente

cinematico e la dimensione informativa necessaria alla costruzione del modello digitale, abilitando la successiva fase di sviluppo del Digital Twin.

4.3 Sviluppo del Digital twin

4.3.1 Architettura del modulo software

Il Digital Twin sviluppato nel presente lavoro è stato implementato come modulo software dedicato all'analisi offline dei dati generati dalla simulazione del robot mobile in ambiente Gazebo.

L'architettura adottata è di tipo modulare e si basa su tre componenti principali:

1. Acquisizione dati – I dati prodotti dalla simulazione vengono registrati mediante rosbag, includendo in particolare i topic relativi alla velocità comandata (*/cmd_vel*) e alla velocità stimata dall'odometria (*/odom*).
2. Elaborazione e sincronizzazione – I dati registrati vengono esportati in formato CSV e successivamente elaborati tramite uno script Python dedicato.

3. Analisi e valutazione – Il modulo calcola metriche quantitative di errore e genera rappresentazioni grafiche comparative tra comando e risposta.

Il Digital Twin non opera in tempo reale, ma in modalità offline, al fine di garantire riproducibilità delle prove, controllo completo sul processo di elaborazione e assenza di problematiche legate alla latenza del sistema.

Dal punto di vista implementativo, il modulo è stato realizzato in linguaggio Python

utilizzando librerie per l'elaborazione numerica e la gestione dei dati temporali. Lo script principale (*plot_vel.py*) si occupa dell'importazione dei dataset, dell'allineamento temporale dei segnali e del calcolo delle metriche di errore.
L'architettura risulta volutamente semplice e focalizzata sul validare quantitativamente la coerenza dinamica tra comando e risposta del sistema simulato.

4.3.2 Flusso dei dati tra simulazione e Twin

Il flusso dei dati tra la simulazione e il Digital Twin può essere descritto come una pipeline strutturata in più fasi consecutive.
Durante l'esecuzione della simulazione in Gazebo, il robot mobile riceve comandi di velocità lineare e angolare tramite il topic */cmd_vel*. Tali comandi vengono elaborati dal modello dinamico del robot e producono una risposta osservabile attraverso il topic */odom*, che fornisce la stima della velocità e della posa del sistema.
I dati generati vengono registrati mediante *rosbag*, creando un file contenente la sequenza temporale dei messaggi scambiati nei topic di interesse. In una fase successiva, i contenuti del rosbag vengono esportati in formato CSV, separando i dati relativi ai comandi di velocità e quelli relativi all'odometria.
Lo script del Digital Twin importa i dataset così ottenuti e costruisce due serie temporali distinte:

- velocità comandata
- velocità stimata dall'odometria

Queste serie costituiscono la base per il confronto quantitativo e qualitativo del

comportamento dinamico del robot simulato.
La separazione tra fase di acquisizione (simulazione) e fase di analisi (Digital Twin) garantisce modularità del sistema e consente di ripetere le elaborazioni senza dover rieseguire la simulazione, favorendo così la riproducibilità degli esperimenti.

4.3.3 Sincronizzazione temporale dei segnali

Per poter confrontare correttamente la velocità comandata e la velocità stimata dal sistema di odometria è stato necessario affrontare il problema della sincronizzazione temporale dei segnali.
I topic */cmd_vel* e */odom* presentano infatti frequenze di pubblicazione differenti e non garantiscono una corrispondenza uno-a-uno tra i campioni. In particolare, il numero di messaggi associati ai due segnali può essere diverso e i timestamp non risultano perfettamente allineati.
Al fine di consentire un confronto punto a punto tra i due andamenti, è stata adottata una procedura di riallineamento temporale basata su associazione al campione più vicino (nearest-neighbor).
La procedura implementata prevede i seguenti passaggi:

- costruzione di due serie temporali indicizzate tramite timestamp;
- ordinamento cronologico dei campioni;
- rimozione di eventuali timestamp duplicati nel segnale di odometria, mantenendo l'ultimo campione disponibile;
- riallineamento del segnale di odometria sulla griglia temporale del segnale di comando mediante ricerca del campione temporalmente più vicino.

In questo modo è stato possibile ottenere due vettori temporali coerenti e confrontabili, consentendo il calcolo dell'errore istantaneo tra velocità comandata e velocità effettiva.
La scelta dell'associazione *nearest-neighbor* rappresenta un compromesso tra semplicità implementativa e affidabilità del confronto, risultando adeguata per un'analisi offline e per le frequenze di campionamento adottate nella simulazione.

4.3.4 Metriche di valutazione

Una volta ottenute due serie temporali sincronizzate e confrontabili, è stato possibile definire una misura quantitativa della coerenza tra velocità comandata e velocità stimata dal sistema di odometria.
L'errore istantaneo è stato definito come la differenza tra il valore della velocità lineare comandata e quello della velocità lineare stimata:

$$e(t_i) = v_{cmd}(t_i) - v_{odom}(t_i)$$

Sulla base di tale errore sono state adottate due metriche quantitative di valutazione:

- Mean Absolute Error (MAE)
- Root Mean Square Error (RMSE)

Il MAE fornisce una misura dell'errore medio assoluto tra i due segnali, risultando poco sensibile a picchi isolati e rappresentando l'errore medio globale del sistema.

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |e_i|$$

Il RMSE, invece, attribuisce maggiore peso agli errori di ampiezza elevata, risultando più sensibile alla presenza di transienti o oscillazioni dinamiche.

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2}$$

L'adozione combinata di queste due metriche consente di ottenere una valutazione più completa del comportamento dinamico del sistema, distinguendo tra errore medio distribuito e presenza di picchi localizzati.
Le metriche sono state calcolate tramite elaborazione numerica in ambiente Python a partire dai dati sincronizzati, costituendo la base quantitativa per la validazione preliminare del Digital Twin.

4.4 Raccolta e visualizzazione dei dati

Prima di procedere con la registrazione e l'analisi dei dati, è stato definito uno scenario sperimentale di riferimento volto a caratterizzare il comportamento del

sistema in condizioni controllate.

Per il primo test è stata scelta una traiettoria a lemniscata (forma a “otto”), implementata come successione continua di punti di riferimento nel piano cartesiano. La scelta di tale traiettoria non è casuale: la lemniscata presenta variazioni di curvatura significative e cambi di direzione alternati, risultando quindi più rappresentativa rispetto a una semplice traiettoria rettilinea o circolare. Essa consente di sollecitare il sistema di controllo sia in termini di modulazione della velocità lineare sia in termini di velocità angolare, evidenziando eventuali limiti nel tracking in presenza di curvature variabili.

La lemniscata è stata generata in forma parametrica, in funzione di un parametro temporale t. Una possibile formulazione, utilizzata per la generazione dei punti di riferimento nel codice Python, è la seguente:

$$x(t) = A\sin(\omega t)$$
$$y(t) = A\sin(\omega t)\cos(\omega t)$$

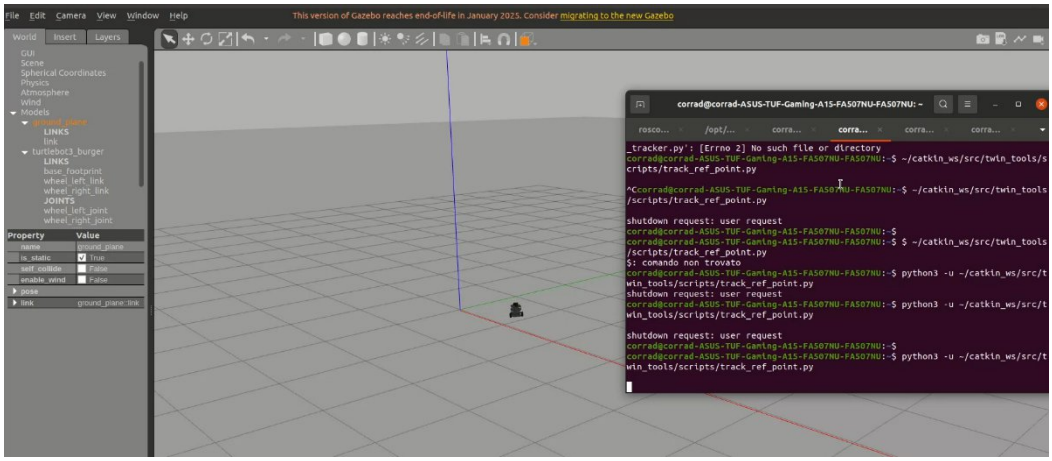
dove:

- A rappresenta l’ampiezza della traiettoria,
- ω è la pulsazione che ne regola la velocità di percorrenza,
- t è il tempo simulato.

Il test è stato condotto in un ambiente simulato privo di ostacoli (empty world), al fine di isolare il comportamento del controllore dalla componente di evitamento ostacoli. Questa scelta metodologica permette di ottenere una baseline prestazionale, utile per valutare esclusivamente le capacità di inseguimento della traiettoria senza interferenze esterne.

L’assenza di ostacoli garantisce inoltre che le metriche calcolate riflettano

unicamente le prestazioni del sistema di controllo e non siano influenzate da deviazioni imposte da vincoli ambientali. Tale configurazione costituisce il punto di partenza per le successive analisi comparative, in cui verranno introdotte condizioni più complesse.



Turtlebot3 durante l'esecuzione della traiettoria

4.4.1 Registrazione tramite Rosbag

La fase di acquisizione dei dati rappresenta il primo passo fondamentale per la costruzione del Digital Twin e per l'analisi quantitativa delle prestazioni del sistema. In ambiente ROS, la registrazione delle informazioni scambiate tra i nodi è stata effettuata tramite il tool rosbag [9], che consente di salvare in modo sincronizzato i messaggi pubblicati sui topic di interesse durante l'esecuzione della simulazione.

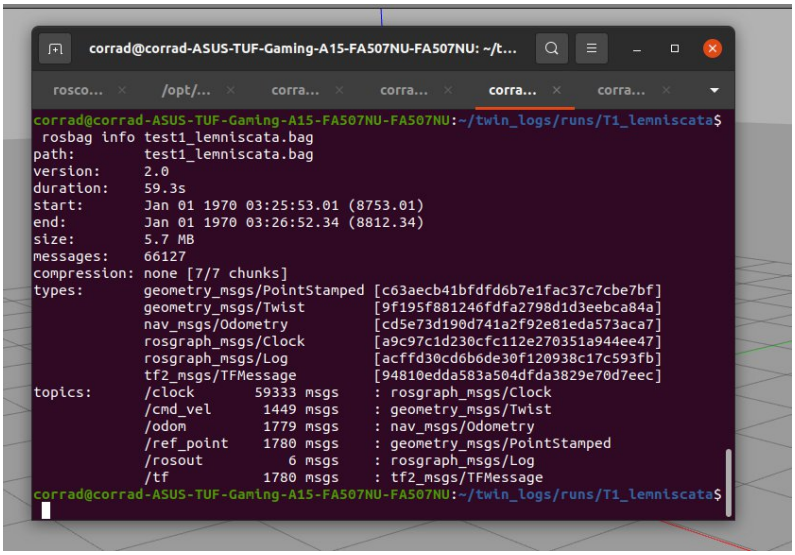
Nel Test 1, relativo all'inseguimento della traiettoria a lemniscata in ambiente privo di ostacoli, sono stati registrati i seguenti topic:

- */odom* – odometria del robot (posizione e velocità)
- */cmd_vel* – comandi di velocità inviati al robot
- */ref_point* – punto di riferimento della traiettoria
- */tf* – trasformazioni tra frame
- */clock* – tempo simulato
- */rosout* – log di sistema

La registrazione è stata eseguita per l'intera durata dell'esperimento, comprendente sia la fase transitoria iniziale (aggancio alla traiettoria) sia la fase a regime. Il file risultante (*test1_lemniscata.bag*) ha una durata complessiva di circa 59s e contiene oltre 66.000 messaggi, garantendo una risoluzione temporale adeguata per l'analisi dinamica del sistema.

Poiché l'esperimento è stato condotto in ambiente simulato (Gazebo), il tempo associato ai messaggi non corrisponde all'orario reale del sistema operativo, ma al tempo pubblicato sul topic */clock*. Questa scelta consente di mantenere coerenza temporale tra la dinamica simulata e i dati acquisiti, evitando disallineamenti dovuti al tempo di sistema.

La registrazione tramite rosbag garantisce inoltre la riproducibilità dell'esperimento, permettendo di rieseguire offline l'analisi dei dati senza necessità di rilanciare la simulazione. Tale caratteristica è fondamentale nell'ottica dell'ottimizzazione iterativa implementata nel Digital Twin, in quanto consente di separare la fase di acquisizione da quella di elaborazione.



Informazioni del file rosbag relativo al Test 1

4.4.2 Esportazione e pre-processing dei dati

Una volta completata la registrazione del Test 1, i dati contenuti nel file rosbag sono stati esportati in formato CSV al fine di consentire un'elaborazione offline tramite script Python. Questa scelta metodologica permette di separare la fase di acquisizione dalla fase di analisi, garantendo maggiore flessibilità e riproducibilità.

L'esportazione è stata effettuata tramite il comando *rostopic echo -b*, che consente di leggere i messaggi contenuti nel file rosbag e salvarli in formato tabellare. In particolare, sono stati estratti:

- i dati di odometria *(/odom*), contenenti posizione e velocità del robot;
- i punti di riferimento della traiettoria (*/ref_point*).

Il risultato è costituito da due file CSV distinti, contenenti per ciascun campione il timestamp associato e i valori delle variabili di interesse.

Poiché i topic */odom* e */ref_point* possono essere pubblicati a frequenze differenti, è stata eseguita una fase di pre-processing volta a garantire coerenza temporale tra i dati. In particolare, i timestamp sono stati convertiti da nanosecondi a secondi, e il riferimento di traiettoria è stato interpolato sui timestamp dell'odometria mediante interpolazione lineare.

Questa operazione è cruciale per evitare errori artificiali nel calcolo della distanza tra traiettoria reale e riferimento. Un confronto diretto per indice, infatti, potrebbe introdurre uno sfasamento temporale non fisicamente significativo. L'interpolazione consente invece di confrontare le posizioni reale e desiderata nello stesso istante temporale simulato, rendendo il calcolo dell'errore coerente dal punto di vista dinamico.

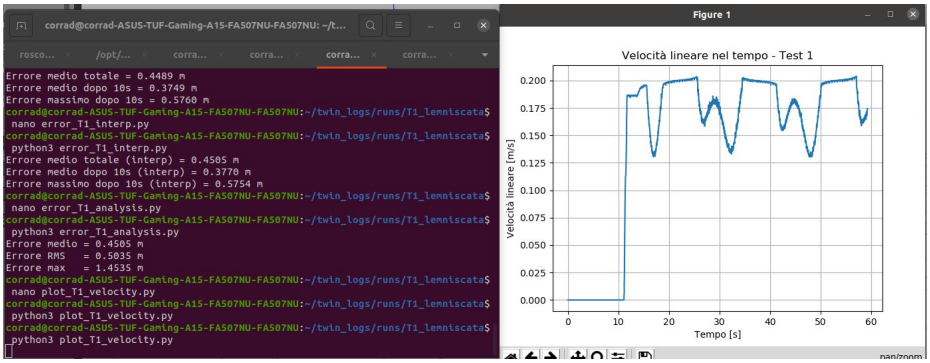
Il pre-processing rappresenta quindi il passaggio che trasforma un insieme di dati grezzi in un dataset strutturato e pronto per l'analisi quantitativa, costituendo la base operativa del Digital Twin.

4.4.3 Visualizzazione grafica delle velocità

Oltre all'analisi della traiettoria nel piano XY, è stata condotta un'analisi temporale delle grandezze dinamiche principali del robot, in particolare della velocità lineare e della velocità angolare. Questa fase consente di comprendere il comportamento del controllore durante l'inseguimento della traiettoria e di individuare eventuali fenomeni di saturazione, instabilità o oscillazioni eccessive.

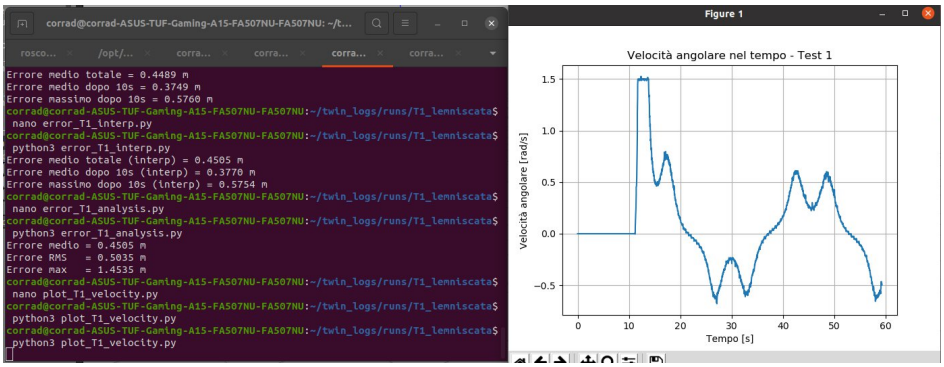
La velocità lineare lungo l'asse principale del robot è stata analizzata in funzione del tempo simulato. Dopo una fase iniziale a velocità nulla, corrispondente al periodo antecedente l'attivazione del controllo, il sistema mostra un rapido incremento della velocità fino a valori prossimi a 0.18–0.20 m/s. Successivamente,

si osservano oscillazioni periodiche della velocità, coerenti con la variazione di curvatura della traiettoria a lemniscata. In corrispondenza delle porzioni più strette della traiettoria, la velocità lineare tende a ridursi, mentre nelle zone a curvatura minore si mantiene su valori più elevati.



Velocità lineare nel tempo durante il Test 1

L'analisi della velocità angolare evidenzia un comportamento complementare. Nella fase iniziale si osserva un picco significativo, associato alla manovra di riallineamento del robot rispetto al riferimento. Una volta raggiunta la traiettoria, la velocità angolare assume un andamento periodico alternato, con segno positivo e negativo in corrispondenza dei due lobi della lemniscata. Tale andamento conferma la stabilità del sistema e l'assenza di fenomeni divergenti.



Velocità angolare nel tempo durante il Test 1

Complessivamente, l'analisi delle velocità conferma che il sistema di controllo è stabile e reattivo, ma presenta un comportamento oscillatorio periodico legato alla dinamica della traiettoria, aspetto che costituisce un potenziale margine di miglioramento nell'ambito dell'ottimizzazione parametrica successiva.

4.4.4 Preparazione dei dati per l'analisi quantitativa

Una volta completata l'estrazione e la sincronizzazione temporale dei dati, è stata effettuata una fase di analisi quantitativa finalizzata alla caratterizzazione oggettiva delle prestazioni del sistema di controllo.
A partire dalle traiettorie reale e di riferimento, è stato calcolato l'errore di tracking come distanza euclidea istantanea tra posizione desiderata e posizione effettiva del robot. L'errore è stato valutato nel tempo, distinguendo tra fase transitoria iniziale e fase a regime, al fine di isolare il contributo dovuto all'aggancio iniziale della traiettoria.
Sono state quindi definite le seguenti metriche:

- Errore medio, indicativo della precisione complessiva del sistema;
- Errore RMS (Root Mean Square), più sensibile ai picchi e quindi rappresentativo della qualità globale del tracking;
- Errore massimo, utile per evidenziare eventuali condizioni critiche;
- Durata dell'esperimento e distanza percorsa, per una valutazione complessiva delle prestazioni cinematiche.

Formulazione matematica delle metriche di errore

Le metriche sopra elencate sono state definite analiticamente come segue:
Sia definita la traiettoria reale del robot come:

$$\mathrm{p}(t_i) = \begin{bmatrix} x(t_i) \\ y(t_i) \end{bmatrix}$$

e la traiettoria di riferimento come:

$$\mathrm{p}_{ref}(t_i) = \begin{bmatrix} x_{ref}(t_i) \\ y_{ref}(t_i) \end{bmatrix}$$

dove $t_i$rappresenta l'i-esimo istante temporale dopo la sincronizzazione dei dati.
Errore istantaneo di tracking
L'errore istantaneo è stato definito come distanza euclidea tra posizione reale e posizione di riferimento:

$$e(t_i) = \| \, p(t_i) - p_{ref}(t_i) \, \|$$

Esplicitamente:

$$e(t_i) = \sqrt{\left(x(t_i) - x_{ref}(t_i)\right)^2 + \left(y(t_i) - y_{ref}(t_i)\right)^2}$$

Errore medio

Indicando con Nil numero totale di campioni considerati, l'errore medio è definito come:

$$e_{medio} = \frac{1}{N}\sum_{i=1}^{N} e(t_i)$$

Questa metrica fornisce una stima della deviazione media complessiva del robot rispetto alla traiettoria desiderata.

Errore RMS (Root Mean Square)

L'errore quadratico medio è stato calcolato come:

$$e_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e^2(t_i)}$$

Questa metrica attribuisce maggiore peso agli errori di ampiezza elevata, risultando più sensibile ai picchi presenti durante la fase transitoria o in corrispondenza di variazioni brusche di curvatura.

Errore massimo

L'errore massimo osservato durante l'esperimento è definito come:

$$e_{max} = \max_{i=1,\dots,N} \left(e(t_i)\right)$$

Tale valore consente di individuare la massima deviazione registrata rispetto alla

traiettoria di riferimento.

Durata dell'esperimento

La durata totale è stata calcolata come:

$$T = t_N - t_1$$

dove t_1e $t_N$rappresentano rispettivamente il primo e l'ultimo istante temporale registrato.

Distanza percorsa

La lunghezza della traiettoria reale è stata stimata mediante somma incrementale:

$$d = \sum_{i=1}^{N-1} \sqrt{\left(x(t_{i+1}) - x(t_i)\right)^2 + \left(y(t_{i+1}) - y(t_i)\right)^2}$$

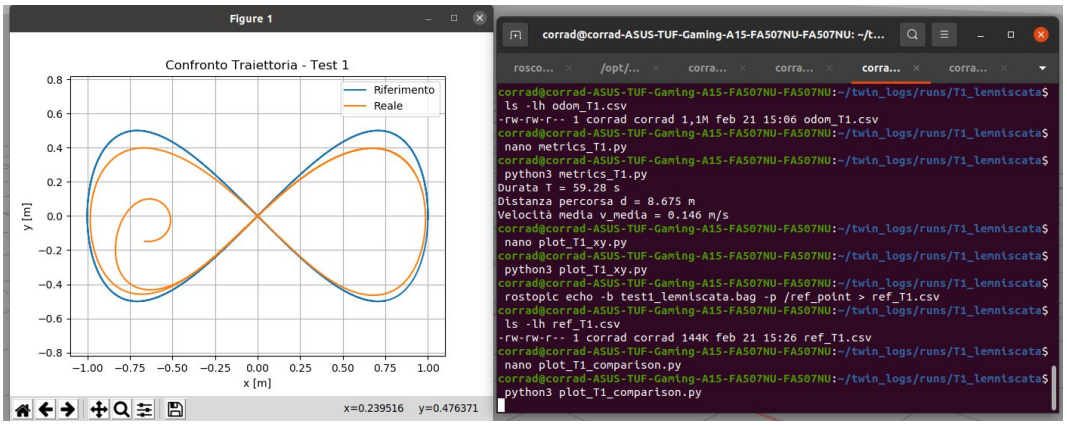
L'analisi dell'errore nel tempo ha evidenziato un picco iniziale associato alla fase di riallineamento del robot rispetto al riferimento, seguito da una fase oscillatoria periodica a regime. Tale comportamento è coerente con la natura della traiettoria a lemniscata e con la struttura del controllore implementato.

Questa fase rappresenta il passaggio fondamentale che consente di trasformare dati cinematici grezzi in indicatori numerici sintetici, utilizzabili come criteri oggettivi di valutazione.

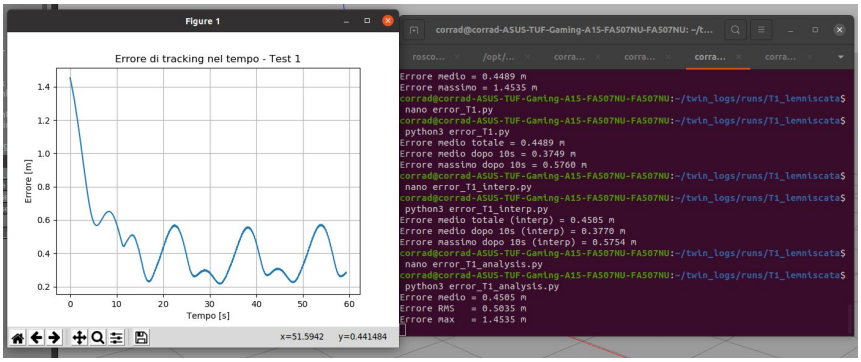
In particolare, le metriche calcolate costituiscono la base per la definizione della funzione costo che verrà utilizzata nel processo di ottimizzazione automatica del Digital Twin nel Capitolo 5. L'obiettivo non sarà soltanto ridurre l'errore medio, ma bilanciare più criteri (accuratezza, tempo di esecuzione, energia stimata) nel

rispetto di vincoli prestazionali prefissati.

In questa prospettiva, il Digital Twin non si limita a monitorare il comportamento del sistema, ma utilizza le metriche derivate dai dati per guidare un processo iterativo di miglioramento parametrico della navigazione.



Confronto tra riferimento e traiettoria eseguita



Errore di tracking nel tempo durante il Test 1

Capitolo 5

Metodologia di ottimizzazione della navigazione

5.1 Costruzione dell'ambiente di testing

Dopo aver validato il corretto funzionamento del sistema di controllo e delle logiche di estrazione metriche in ambienti semplici (Capitolo 4), si è proceduto alla definizione di uno scenario sperimentale realistico, finalizzato a valutare le prestazioni del robot mobile in un contesto coerente con applicazioni logistiche industriali.

L'ambiente selezionato per gli scenari S1 ed S2 riproduce un magazzino industriale caratterizzato da corridoi stretti, scaffalature parallele, colli posizionati lungo i percorsi e presenza di mezzi operativi. La scelta di tale configurazione è motivata dalla crescente diffusione di robot mobili autonomi (AMR) in ambito logistico, dove la navigazione avviene in spazi vincolati e dinamicamente complessi, nei quali risulta fondamentale bilanciare rapidità di movimento, sicurezza e stabilità dinamica.

La progettazione dell'ambiente in Gazebo è stata guidata dai seguenti requisiti:

- Corridoi longitudinali definiti, con larghezza tale da consentire il passaggio del robot ma con margini ridotti, in modo da rendere significativa la valutazione della distanza minima dagli ostacoli.

- Scaffalature parallele disposte in modo regolare, al fine di creare vincoli geometrici ripetitivi e simulare configurazioni tipiche di magazzini industriali.
- Colli e pallet distribuiti in posizione pseudo-casuale, per introdurre variabilità spaziale e situazioni in cui il planner locale è costretto a operare in prossimità di ostacoli.
- Inserimento di muletti industriali statici, con funzione di ostacoli

volumetrici complessi, al fine di aumentare il realismo della scena.

- Assenza di compenetrazioni tra modelli, verificata manualmente durante la fase di progettazione per garantire coerenza fisica e corretto funzionamento della simulazione.

Tali vincoli consentono di ottenere un ambiente strutturato ma non banale, nel quale il robot deve affrontare situazioni di navigazione realistiche, caratterizzate da passaggi stretti e traiettorie obbligate.

L'ambiente è stato realizzato all'interno del simulatore Gazebo mediante:

- definizione di un world personalizzato;
- inserimento progressivo dei modelli di scaffalature e colli;
- verifica delle collisioni e della coerenza spaziale;
- controllo delle distanze minime tra gli elementi strutturali.

Particolare attenzione è stata dedicata alla disposizione dei colli, distribuiti lungo i corridoi senza intersezioni con scaffali o pareti, al fine di evitare condizioni di collisione non realistiche o errori nella generazione delle *costmap* che verrà definita in seguito.
L'ambiente risultante presenta una configurazione strutturata e ripetibile, idonea all'esecuzione di prove comparative e alla valutazione oggettiva delle metriche definite nel Digital Twin.

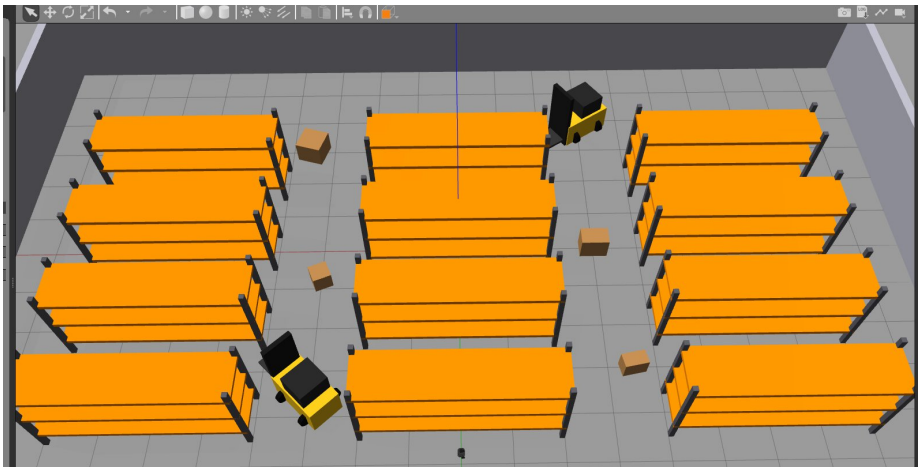
L'introduzione di un magazzino industriale simulato consente di superare i limiti dei test condotti in ambienti semplificati. In particolare, tale configurazione:

- rende significativo il parametro di distanza minima dagli ostacoli $dmin$
- introduce vincoli geometrici che influenzano direttamente la scelta dei

parametri del planner locale;

- permette di analizzare il compromesso tra velocità massima consentita e sicurezza operativa;
- fornisce un contesto realistico per l'applicazione dell'ottimizzazione tramite Digital Twin.

Lo scenario S1 costituisce pertanto il banco di prova principale per l'analisi comparativa delle strategie di navigazione e per la successiva fase di ottimizzazione dei parametri dinamici del robot.



Scenario industriale di testing costruito ad hoc

Integrazione del TurtleBot3 nell'ambiente

Una volta completata la configurazione dell'ambiente di magazzino in Gazebo, è stato necessario integrare il modello del robot mobile all'interno del world personalizzato. L'inserimento del TurtleBot3 non è avvenuto in un ambiente predefinito fornito dal pacchetto ufficiale, bensì all'interno di uno scenario creato manualmente, contenente scaffalature, colli e ulteriori elementi statici.

L'integrazione è stata effettuata mantenendo l'architettura software standard del TurtleBot3 basata su ROS, includendo:

- caricamento del modello URDF del robot;
- attivazione dei nodi di simulazione e controllo;
- avvio del navigation stack (AMCL, costmap, planner globale e locale);
- collegamento ai topic di odometria, laser scanner e comando velocità.

Particolare attenzione è stata dedicata alla definizione della posa iniziale del robot all'interno del magazzino. Diversamente dai mondi predefiniti, nei quali la posizione iniziale è già configurata, nel world personalizzato è stato necessario specificare manualmente la posizione e l'orientamento del robot in coordinate globali, assicurando che:

- il robot fosse posizionato all'interno di un corridoio navigabile;
- non vi fossero compenetrazioni con elementi strutturali;
- la configurazione iniziale risultasse coerente con la mappa utilizzata dal sistema di localizzazione.

La corretta integrazione è stata verificata controllando la coerenza tra:

- frame di riferimento map, odom e base_link;
- corretta generazione delle costmap globale e locale;
- corretta visualizzazione in RViz del modello e dei sensori.

Questa fase ha permesso di ottenere una piattaforma simulata completamente operativa all'interno dell'ambiente di magazzino progettato, costituendo la base per le successive prove sperimentali e per l'analisi prestazionale condotta nel resto del capitolo.

5.2 Visualizzazione (Rviz) e Localizzazione (SLAM) nel contesto sperimentale

Per consentire al robot mobile di operare correttamente all'interno dell'ambiente di magazzino simulato, è stato necessario integrare strumenti di visualizzazione e localizzazione non precedentemente descritti. In particolare, sono stati utilizzati RViz come ambiente di visualizzazione e SLAM (Simultaneous Localization and Mapping) per la costruzione della mappa dell'ambiente e la successiva localizzazione del robot.

RViz: strumento di visualizzazione e diagnostica

RViz è l'ambiente di visualizzazione grafica associato a ROS [17], utilizzato per rappresentare in tempo reale lo stato del robot, i dati dei sensori e le informazioni generate dagli algoritmi di navigazione. Attraverso RViz è possibile visualizzare:

- il modello tridimensionale del robot;
- i frame di riferimento (map, odom, base_link);
- i dati provenienti dal sensore LIDAR;
- la mappa dell'ambiente;
- le costmap globale e locale;
- la traiettoria pianificata dal planner globale;
- il comportamento del planner locale.

Nel contesto della presente tesi, RViz ha svolto un ruolo fondamentale sia in fase di sviluppo sia in fase di validazione. In particolare, è stato utilizzato per:

- verificare la corretta coerenza tra i sistemi di riferimento;
- controllare la qualità della mappa generata tramite SLAM;
- analizzare il comportamento del planner in prossimità degli ostacoli;
- monitorare eventuali instabilità o oscillazioni durante la navigazione.

RViz non costituisce un componente algoritmico del sistema, ma rappresenta uno strumento essenziale per la verifica qualitativa del comportamento del robot e per l'interpretazione dei risultati sperimentali.

SLAM: Simultaneous Localization and Mapping

Il problema della navigazione autonoma in ambienti sconosciuti o non mappati richiede la capacità di costruire una rappresentazione dell'ambiente e, contemporaneamente, stimare la posizione del robot all'interno di tale rappresentazione. Questo problema è noto come SLAM (Simultaneous Localization and Mapping). [13]

Nel caso in esame, il sistema SLAM è stato utilizzato per generare la mappa bidimensionale dell'ambiente di magazzino simulato a partire dai dati del sensore LIDAR. Il processo avviene mediante:

1. acquisizione delle scansioni laser;

2. stima incrementale della posa del robot;
3. aggiornamento progressivo della mappa di occupazione.

Il risultato è una mappa di tipo occupancy grid, nella quale ciascuna cella rappresenta la probabilità di occupazione dello spazio.

Una volta generata la mappa dell'ambiente, questa viene salvata e successivamente utilizzata dal sistema di localizzazione (AMCL – Adaptive Monte Carlo Localization), che consente al robot di stimare la propria posizione all'interno della mappa durante le prove sperimentali.

Ruolo di RViz e SLAM nella tesi

Nel contesto della presente tesi, l'utilizzo combinato di SLAM e RViz ha consentito di:

- ottenere una rappresentazione coerente dell'ambiente di magazzino;
- garantire che la navigazione avvenisse in un sistema di riferimento globale stabile (frame map);
- verificare la correttezza della localizzazione durante i test di navigazione;
- monitorare in tempo reale l'effetto delle variazioni dei parametri del planner locale.

La disponibilità di una mappa accurata e di una localizzazione affidabile costituisce un prerequisito fondamentale per l'applicazione del Digital Twin e per la valutazione oggettiva delle metriche prestazionali introdotte nei paragrafi successivi.

5.2.1 Generazione della mappa dell'ambiente simulato

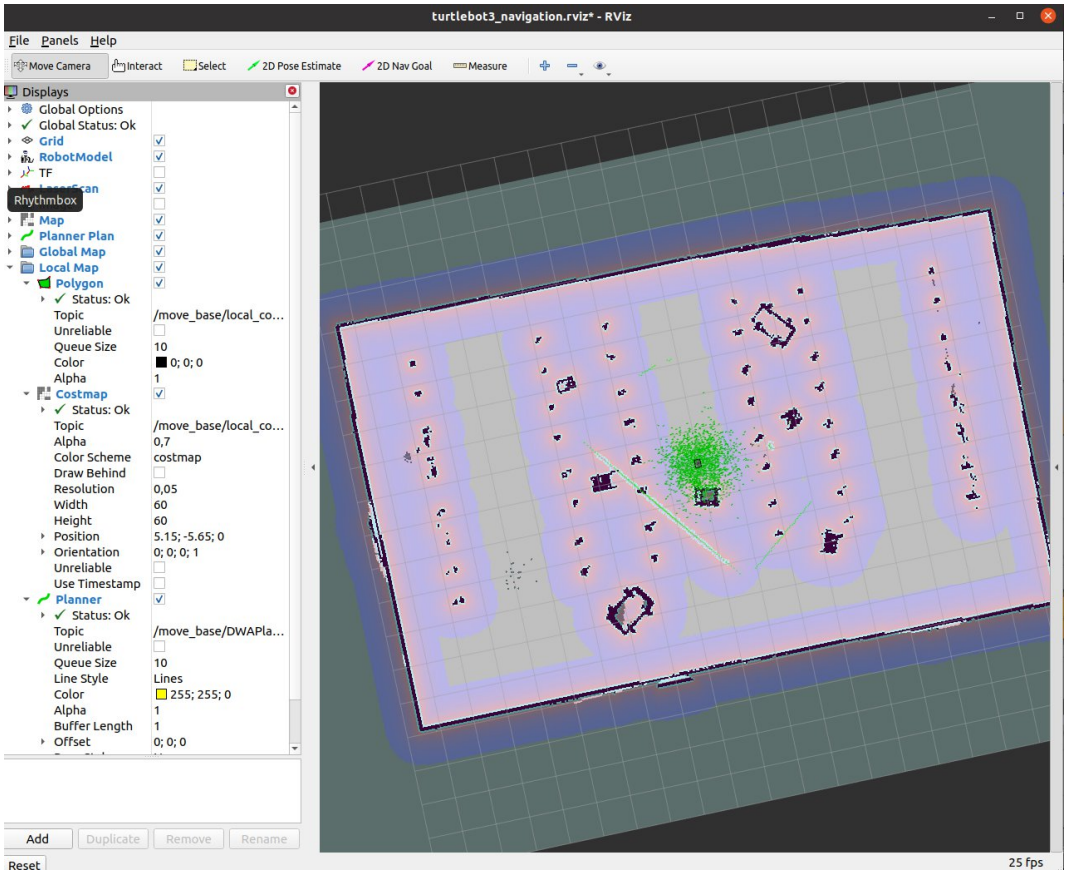
Prima di eseguire i test di navigazione è stato necessario generare una mappa dell'ambiente simulato, che potesse essere utilizzata dal sistema di localizzazione e pianificazione del percorso del robot.

L'ambiente di prova è stato modellato in Gazebo riproducendo uno scenario di tipo magazzino, caratterizzato da corridoi delimitati da scaffalature e da alcune aree di passaggio più strette. Per consentire al robot di operare correttamente in questo ambiente è stata quindi creata una mappa bidimensionale dell'area mediante una fase iniziale di esplorazione.

La procedura è stata eseguita guidando manualmente il robot all'interno dell'ambiente tramite teleoperazione. Durante questa fase il robot ha attraversato progressivamente le diverse zone del magazzino simulato, mentre il sensore LiDAR acquisiva le distanze dagli ostacoli circostanti. Le informazioni rilevate sono state utilizzate dal sistema di mappatura per costruire progressivamente una rappresentazione dell'ambiente.

La costruzione della mappa è stata monitorata tramite RViz, che ha permesso di osservare in tempo reale la formazione della rappresentazione spaziale e verificare la corretta copertura delle diverse aree dell'ambiente. L'esplorazione è stata proseguita fino a quando la struttura del magazzino risultava completamente rappresentata nella mappa.

Una volta completata la fase di mappatura, la mappa generata è stata salvata e successivamente utilizzata dal sistema di navigazione per le prove sperimentali descritte nelle sezioni successive.



Mappa dell'ambiente di magazzino simulato generata tramite esplorazione del robot e visualizzata in RViz.

5.3 Costruzione del Digital Twin

Il Digital Twin sviluppato nel presente lavoro è stato implementato come modulo software dedicato alla supervisione e all'analisi delle prestazioni del sistema di navigazione del robot mobile durante le prove sperimentali. Il Twin è stato realizzato in linguaggio **Python 3** all'interno dell'ambiente **ROS1 Noetic**, operando come componente software esterno alla simulazione fisica ma strettamente integrato con i nodi del sistema di navigazione. Il suo compito principale è quello di gestire automaticamente l'esecuzione delle prove di navigazione, raccogliere i dati generati dal robot durante le missioni e fornire un'infrastruttura software per la successiva analisi delle prestazioni.

Dal punto di vista architetturale, il Digital Twin interagisce con la simulazione e con lo stack di navigazione di ROS tramite la sottoscrizione ai principali **topic ROS** utilizzati dal robot. In particolare, le informazioni relative alla posizione e alla velocità del robot vengono acquisite dal *topic /odom*, che pubblica messaggi di tipo *nav_msgs/Odometry*. I comandi di velocità inviati al robot vengono monitorati attraverso il topic */cmd_vel*, che trasmette messaggi *geometry_msgs/Twist*, mentre le misurazioni provenienti dal sensore laser vengono ottenute dal topic */scan*, contenente messaggi di tipo *sensor_msgs/LaserScan*. L'analisi combinata di questi flussi informativi consente al Twin di ricostruire in tempo reale lo stato dinamico del robot durante l'esecuzione delle missioni di navigazione.

Il sistema implementa inoltre un meccanismo automatico per la gestione delle prove sperimentali. All'inizio di ogni trial la simulazione viene riportata a una configurazione iniziale definita, riposizionando il robot nella posa di partenza tramite il servizio ROS */gazebo/set_model_state*, che consente di modificare direttamente lo stato del modello del robot all'interno dell'ambiente Gazebo. Successivamente viene inizializzata la stima di posizione del sistema di localizzazione pubblicando un messaggio *PoseWithCovarianceStamped* sul topic */initialpose*, permettendo al nodo **AMCL** di allineare correttamente la stima della posizione del robot con la configurazione iniziale della simulazione.

Una volta completata la fase di inizializzazione, il Digital Twin invia al sistema di navigazione il goal di missione tramite l'interfaccia actionlib di ROS utilizzata dal nodo *move_base*. In particolare, il goal viene definito tramite un messaggio *MoveBaseGoal*, che specifica la posa target del robot nel sistema di riferimento globale della mappa. Il nodo *move_base* si occupa quindi di pianificare e seguire il percorso verso l'obiettivo, utilizzando il planner globale e il planner locale configurati nello stack di navigazione.

Durante l'esecuzione della missione il Twin monitora continuamente l'evoluzione dello stato del robot attraverso i dati di odometria e le misurazioni del sensore laser. A questo scopo è stato implementato un modulo di monitoraggio che registra l'evoluzione temporale della posizione, della velocità e delle distanze dagli ostacoli. Tali informazioni permettono di valutare il progresso del robot verso l'obiettivo e di individuare eventuali condizioni anomale durante la navigazione, come collisioni con ostacoli, rotazioni prolungate senza avanzamento o situazioni di stallo del robot.

Un aspetto fondamentale dell'implementazione riguarda la possibilità di modificare dinamicamente i parametri del planner locale utilizzato per la navigazione. Nel sistema sviluppato, il planner locale impiegato è il **Dynamic Window Approach (DWA)**, implementato nel nodo DWAPlannerROS dello stack move base. Il Digital Twin interagisce con tale planner mediante il meccanismo di **dynamic reconfigure** di ROS, che consente di aggiornare i parametri di configurazione del planner durante l'esecuzione degli esperimenti senza interrompere l'esecuzione dei nodi.

Attraverso questa interfaccia è possibile modificare diversi parametri che influenzano direttamente il comportamento del robot durante la navigazione. Tra questi rientrano la velocità massima consentita (*max_vel_x*), i limiti di accelerazione (*acc_lim_x*), il peso associato alla distanza dagli ostacoli (*occdist_scale*) e i parametri utilizzati dal planner per valutare le traiettorie

candidate (*path_distance_bias e goal_distance_bias*). Il Digital Twin applica tali modifiche tramite un client di dynamic_reconfigure, aggiornando automaticamente la configurazione del planner prima dell'avvio di ciascun trial.

Oltre alla supervisione della navigazione, il Twin implementa anche meccanismi di gestione delle situazioni di blocco del robot. Nel caso in cui durante l'esecuzione della missione vengano rilevate condizioni di stallo o di mancato progresso verso l'obiettivo, il sistema può attivare una strategia di recupero temporanea basata su un comportamento di evitamento degli ostacoli, che consente al robot di aggirare l'ostacolo prima di riprendere il normale processo di pianificazione tramite move_base.

Durante ogni prova sperimentale il Digital Twin registra inoltre una serie di informazioni relative al comportamento del robot, tra cui la traiettoria percorsa, il tempo di completamento della missione e alcune grandezze cinematiche derivate dai dati di odometria. Tutti i dati raccolti vengono salvati in file di log in formato **CSV**, consentendo una successiva analisi offline delle prestazioni del sistema e la generazione dei grafici utilizzati per il confronto tra le diverse configurazioni sperimentali.

Grazie a questa architettura, il Digital Twin consente di automatizzare l'intero ciclo sperimentale, che comprende l'inizializzazione della simulazione, la configurazione dei parametri di navigazione, l'esecuzione della missione e la registrazione dei risultati. Tale approccio permette di eseguire in modo sistematico sequenze di esperimenti ripetibili e di raccogliere in maniera strutturata i dati necessari alla successiva fase di valutazione delle prestazioni e ottimizzazione dei parametri del sistema di navigazione.

Schema architetturale del sistema Digital Twin sviluppato per la raccolta dei dati di navigazione e il supporto all'ottimizzazione dei parametri del robot.

Per ragioni di completezza e riproducibilità del lavoro, il codice sorgente principale utilizzato per l'esecuzione degli esperimenti è riportato integralmente nell'Appendice della presente tesi.

5.4 Definizione delle metriche e della funzione di costo

Per valutare quantitativamente le prestazioni del sistema di navigazione del robot mobile è stata definita una funzione di costo che combina diverse metriche relative al comportamento del robot durante l'esecuzione delle missioni. L'obiettivo di tale funzione è fornire un indicatore numerico sintetico della qualità della navigazione, che possa essere utilizzato dal sistema di ottimizzazione per confrontare diverse configurazioni dei parametri del planner locale.

Durante ogni trial di navigazione il Digital Twin acquisisce i dati provenienti dai principali topic ROS del sistema, tra cui odometria *(/odom*), comandi di velocità (*/cmd_vel*) e misurazioni del sensore laser (*/scan)*. A partire da questi dati vengono calcolate alcune grandezze rappresentative delle prestazioni del robot, tra cui il tempo di completamento della missione, la distanza minima dagli ostacoli e alcune grandezze cinematiche derivate dalla velocità del robot.

La funzione di costo utilizzata nel processo di valutazione è definita come combinazione lineare delle principali metriche considerate [13]:

$$J = w_T T + w_E E + w_A A_{RMS} + P$$

dove:

- T rappresenta il tempo di completamento della missione,
- E rappresenta una stima proxy dell'energia consumata,
- A_{RMS} rappresenta il valore RMS dell'accelerazione del robot,
- P rappresenta eventuali termini di penalità associati a condizioni di navigazione indesiderate,
- w_T, w_E e w_A sono i pesi associati alle diverse metriche.

Nel sistema sviluppato i pesi sono stati scelti come:

$$w_T = 1, w_E = 1, w_A = 10$$

La scelta di tali valori riflette l'obiettivo di ottenere un comportamento di navigazione non solo rapido ma anche stabile. In particolare, il peso maggiore assegnato al termine relativo all'accelerazione RMS è motivato dal fatto che accelerazioni elevate indicano un comportamento aggressivo del planner locale, caratterizzato da continui cambiamenti di velocità e oscillazioni nella traiettoria. Tali comportamenti sono indesiderabili sia dal punto di vista dell'efficienza energetica sia dal punto di vista della stabilità del moto del robot.

Tempo di missione

Il termine T rappresenta il tempo necessario al robot per raggiungere il goal assegnato. Tale grandezza viene misurata come il tempo trascorso tra l'invio del goal al nodo move_base e il raggiungimento dell'obiettivo. Questa metrica rappresenta uno degli indicatori principali delle prestazioni del sistema di navigazione, poiché un tempo di missione minore indica una pianificazione più efficiente del percorso e una migliore gestione del moto del robot.

Stima dell'energia consumata

La stima del consumo energetico del robot è stata ottenuta mediante un modello semplificato basato sui dati cinematici derivati dall'odometria. Considerando la massa del robot pari a **1 kg**, coerente con le specifiche del TurtleBot3 Burger, l'energia proxy è stata stimata come:

$$E = \int m \ | \ v(t) \cdot a(t) \ | \ dt$$

dove $v(t)$ rappresenta la velocità lineare del robot e $a(t)$ l'accelerazione stimata numericamente a partire dai dati di odometria

Nel sistema implementato, l'accelerazione viene calcolata numericamente a partire dai dati di odometria secondo la relazione:

$$a(t) = \frac{v(t) - v(t - \Delta t)}{\Delta t}$$

L'integrale viene quindi approssimato numericamente sommando i contributi discreti durante l'esecuzione della missione. Questa grandezza rappresenta una stima qualitativa del lavoro richiesto ai motori del robot per effettuare la navigazione.
La scelta di utilizzare una proxy energetica di questo tipo è coerente con le caratteristiche del robot utilizzato, il TurtleBot3 Burger, che è equipaggiato con due motori DC a trazione differenziale. Pertanto, la relazione tra velocità e accelerazione del robot costituisce una buona approssimazione del livello di sforzo richiesto ai motori durante il movimento.

Accelerazione RMS

Un ulteriore indicatore utilizzato nella funzione di costo è il valore RMS dell'accelerazione lineare del robot, definito come:

$$A_{RMS} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} a_i^2}$$

dove a_i rappresenta l'accelerazione stimata nei diversi istanti temporali durante la missione.
Questa metrica consente di quantificare il livello medio delle variazioni di velocità del robot. Valori elevati di accelerazione RMS indicano un comportamento di navigazione caratterizzato da oscillazioni e cambiamenti rapidi di velocità, che possono essere associati a una pianificazione locale instabile o a una scarsa regolazione dei parametri del planner.

Penalità e condizioni di fallimento

Oltre ai termini principali della funzione di costo, il sistema prevede l'introduzione di penalità nel caso in cui si verifichino condizioni di navigazione non accettabili. In particolare, vengono applicate penalità elevate nei casi di collisione con ostacoli, mancato progresso verso il goal o eccessivo consumo energetico. In tali situazioni la funzione di costo assume valori molto elevati, scoraggiando la selezione di configurazioni dei parametri che portano a tali comportamenti.

Condizioni di sicurezza e criteri di arresto delle prove

Oltre ai termini principali della funzione di costo, il sistema implementa una serie di condizioni di sicurezza e criteri di arresto anticipato delle prove sperimentali. Tali meccanismi hanno lo scopo di evitare che configurazioni dei parametri di navigazione che portano a comportamenti non accettabili vengano considerate valide durante il processo di ottimizzazione.

Nel Digital Twin queste condizioni sono implementate come *abort conditions*,

ovvero regole che interrompono anticipatamente un trial quando vengono rilevate situazioni di navigazione pericolose o inefficaci. Quando una di queste condizioni si verifica, la prova viene terminata e alla configurazione dei parametri viene assegnato un valore di costo molto elevato, scoraggiandone la selezione nelle iterazioni successive del processo di ottimizzazione.

Una prima condizione riguarda la distanza minima dagli ostacoli, calcolata a partire dalle misurazioni del sensore LiDAR. Durante l'esecuzione della missione viene monitorato il valore minimo delle distanze rilevate dal sensore. Se tale distanza scende al di sotto di una soglia prefissata, la navigazione viene considerata non sicura e il trial viene interrotto. Questo meccanismo consente di evitare configurazioni dei parametri del planner che portano il robot a transitare eccessivamente vicino agli ostacoli presenti nell'ambiente.

Un ulteriore criterio di arresto anticipato riguarda il mancato progresso verso l'obiettivo. Il Digital Twin monitora continuamente la distanza tra il robot e il goal della missione. Se entro una determinata finestra temporale non viene rilevato un miglioramento significativo di tale distanza, la navigazione viene considerata inefficace e il trial viene terminato. Questa condizione consente di individuare situazioni di stallo del robot, ad esempio quando il planner locale rimane intrappolato in oscillazioni o in traiettorie cicliche senza avvicinarsi al goal.

Sono inoltre previste condizioni di abort legate al tempo massimo di missione e al livello di energia stimato. Se il tempo necessario per completare la missione supera una soglia prefissata oppure se l'energia proxy accumulata durante il movimento supera un limite massimo, il trial viene interrotto e classificato come fallito. In questo modo vengono penalizzate configurazioni del planner che portano a traiettorie eccessivamente lunghe o caratterizzate da movimenti inefficienti.

Infine, il sistema rileva situazioni di rotazione prolungata senza avanzamento, che possono verificarsi quando il planner locale genera comandi di rotazione continui senza produrre un reale progresso nello spazio. Anche in questo caso il trial viene terminato anticipatamente per evitare che tali comportamenti influenzino negativamente il processo di ottimizzazione.

L'introduzione di queste condizioni di sicurezza consente di rendere il processo di valutazione più robusto, evitando che il sistema di ottimizzazione esplori configurazioni dei parametri che producono comportamenti non realistici o non accettabili dal punto di vista operativo. In questo modo la funzione di costo non valuta esclusivamente l'efficienza della navigazione, ma incorpora anche criteri di sicurezza e stabilità del comportamento del robot.

Formulazione del problema di ottimizzazione

L'obiettivo del sistema sviluppato è individuare una configurazione dei parametri del planner locale che consenta di migliorare le prestazioni di navigazione del robot all'interno dell'ambiente simulato. Tale problema può essere formalizzato come un problema di **ottimizzazione parametrica**, in cui si ricerca il vettore di parametri del sistema che minimizza la funzione di costo definita nella sezione precedente.
Sia quindi definito il vettore dei parametri del planner locale:

$$\mathbf{x} = \begin{bmatrix} \text{max\_vel\_x} \\ \text{acc\_lim\_x} \\ \text{occdist\_scale} \\ \text{inflation\_radius} \\ \text{path\_distance\_bias} \\ \text{goal\_distance\_bias} \end{bmatrix}$$

Ciascun elemento del vettore dei parametri **x** rappresenta una configurazione del planner locale che influisce direttamente sul comportamento del robot durante la navigazione. In particolare:

- **max_vel_x** rappresenta la velocità lineare massima consentita al robot lungo l'asse di avanzamento. Questo parametro limita la velocità con cui il robot può muoversi durante la pianificazione locale.
- **acc_lim_x** rappresenta il limite massimo di accelerazione lineare del robot. Tale parametro determina quanto rapidamente il robot può variare la propria velocità durante l'esecuzione delle traiettorie pianificate.
- **occdist_scale** è il peso associato alla distanza dagli ostacoli nella funzione di valutazione delle traiettorie del planner locale. Valori più elevati portano il planner a privilegiare traiettorie che mantengono una maggiore distanza dagli ostacoli.
- **inflation_radius** rappresenta il raggio di inflazione degli ostacoli nella costmap locale. Questo parametro determina quanto lo spazio occupato dagli ostacoli venga espanso nella mappa di costo, influenzando quindi la distanza di sicurezza mantenuta dal robot.
- **path_distance_bias** è il peso associato alla distanza dalla traiettoria globale pianificata. Valori più elevati incentivano il robot a seguire più fedelmente il percorso globale calcolato dal planner globale.
- **goal_distance_bias** rappresenta il peso associato alla distanza dal goal nella funzione di valutazione delle traiettorie candidate. Questo parametro

influenza la tendenza del robot a privilegiare traiettorie che riducono più rapidamente la distanza dall'obiettivo finale.

Il problema di ottimizzazione può quindi essere espresso come:

$$\min_{\mathbf{x}} J(\mathbf{x})$$

dove $J(\mathbf{x})$rappresenta la funzione di costo definita in precedenza e calcolata sulla base delle metriche di prestazione osservate durante l'esecuzione delle missioni di navigazione.

La ricerca della configurazione ottimale dei parametri è soggetta a una serie di **vincoli operativi**, che riflettono sia i limiti fisici del robot sia le condizioni di sicurezza imposte durante gli esperimenti. In particolare, ogni parametro è vincolato all'interno di un intervallo di valori compatibili con le caratteristiche del robot TurtleBot3 Burger:

$$x_i^{min} \leq x_i \leq x_i^{max}$$

Tali limiti sono stati definiti sulla base delle specifiche del robot e delle raccomandazioni presenti nella documentazione dello stack di navigazione ROS. Ad esempio, la velocità massima lineare del robot è limitata a valori compatibili con le capacità dinamiche del TurtleBot3, mentre i parametri di valutazione delle traiettorie sono vincolati a intervalli che garantiscono un comportamento stabile del planner locale.

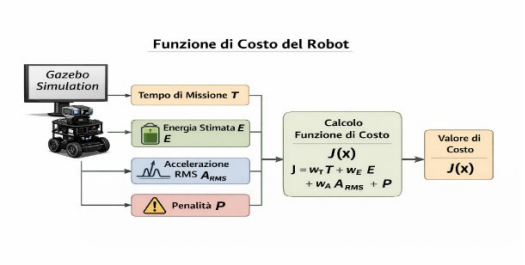
Oltre ai vincoli sui parametri, il processo di ottimizzazione è soggetto anche ai

vincoli di sicurezza introdotti nella sezione precedente. In presenza di collisioni, mancato progresso verso il goal o superamento delle soglie di sicurezza definite, il trial viene terminato anticipatamente e alla configurazione dei parametri viene assegnato un valore di costo molto elevato. Questo approccio consente di penalizzare automaticamente configurazioni che producono comportamenti non accettabili.

Il processo di valutazione della funzione di costo avviene quindi attraverso l'esecuzione di **trial di navigazione** all'interno della simulazione. Per ogni configurazione dei parametri $\mathbf{x}$, il Digital Twin esegue una missione di navigazione completa, raccoglie i dati generati dal robot e calcola il valore della funzione di costo associata alla configurazione testata.

L'insieme di questi elementi definisce quindi un problema di **ottimizzazione black-box**, in cui la funzione di costo non è espressa analiticamente ma viene valutata attraverso l'esecuzione della simulazione del sistema. In tali contesti, metodi di ottimizzazione basati su gradienti risultano generalmente inapplicabili, poiché non è possibile calcolare direttamente la derivata della funzione obiettivo rispetto ai parametri del sistema.

Per questo motivo, nella presente tesi sono stati adottati algoritmi di **ottimizzazione stocastica senza gradiente**, particolarmente adatti alla risoluzione di problemi di ottimizzazione non lineari e rumorosi come quello considerato. I metodi di ottimizzazione utilizzati e le loro principali caratteristiche sono descritti nel paragrafo successivo.



Struttura della funzione di costo J(X) utilizzata nel processo di ottimizzazione

5.5 Algoritmi di ottimizzazione

L'ottimizzazione dei parametri del sistema di navigazione rappresenta un passaggio fondamentale per migliorare le prestazioni del robot durante l'esecuzione delle missioni. Nel contesto considerato, il comportamento del robot dipende da un insieme di parametri che regolano il funzionamento del planner locale basato sul **Dynamic Window Approach (DWA) [16]**. Tali parametri influenzano direttamente la generazione e la valutazione delle traiettorie candidate durante la navigazione, determinando caratteristiche quali la velocità del robot, la distanza mantenuta dagli ostacoli e la fedeltà al percorso globale pianificato.

L'obiettivo dell'ottimizzazione è quindi individuare una configurazione dei parametri del planner che consenta di minimizzare la funzione di costo definita nella sezione precedente, migliorando al contempo l'efficienza e la stabilità del comportamento di navigazione. Dal punto di vista matematico, questo problema può essere interpretato come una ricerca nello spazio dei parametri del sistema, in cui ogni configurazione dei parametri viene valutata attraverso l'esecuzione di una missione di navigazione nella simulazione.

A differenza di molti problemi classici di ottimizzazione, la funzione di costo considerata in questo lavoro non è disponibile in forma analitica. Il valore della funzione obiettivo può essere ottenuto soltanto eseguendo una simulazione completa del sistema di navigazione e analizzando i dati generati dal robot durante la missione. Questo tipo di problema è comunemente indicato come **problema di ottimizzazione black-box**, in cui la funzione obiettivo è accessibile solo attraverso valutazioni sperimentali.

Un'ulteriore difficoltà deriva dal fatto che la funzione di costo presenta caratteristiche fortemente **non lineari** e può risultare affetta da **rumore** dovuto alla variabilità del comportamento del robot durante le simulazioni. Inoltre, la

presenza di condizioni di abort e di penalità introduce discontinuità nella funzione obiettivo, rendendo difficile l'utilizzo di metodi di ottimizzazione tradizionali basati sul calcolo del gradiente.

Per queste ragioni, nel presente lavoro si è scelto di utilizzare **algoritmi di ottimizzazione senza gradiente (gradient-free)**. Questa classe di metodi non richiede il calcolo esplicito della derivata della funzione obiettivo rispetto ai parametri del sistema e risulta particolarmente adatta per problemi di ottimizzazione complessi in cui la funzione di costo è valutabile solo tramite simulazione.

Gli algoritmi gradient-free consentono di esplorare lo spazio dei parametri attraverso strategie stocastiche o evolutive, valutando iterativamente diverse configurazioni del sistema e aggiornando progressivamente la ricerca verso regioni dello spazio dei parametri che producono prestazioni migliori. Tali metodi risultano quindi particolarmente efficaci in contesti come quello considerato, in cui il comportamento del sistema emerge dall'interazione di numerosi fattori dinamici e non può essere descritto tramite un modello matematico semplice.

5.5.1 Particle Swarm Optimization (PSO)

Tra i metodi di ottimizzazione adottati nel presente lavoro, un primo approccio è stato basato sul Particle Swarm Optimization (PSO), algoritmo di ottimizzazione stocastica ispirato al comportamento collettivo osservabile in sistemi naturali quali sciami di insetti, stormi di uccelli o banchi di pesci [14]. L'idea alla base del metodo consiste nel far evolvere nel tempo un insieme di soluzioni candidate, dette *particelle*, all'interno dello spazio dei parametri del problema, sfruttando un meccanismo di cooperazione tra esplorazione individuale ed esperienza collettiva.

Nel contesto del problema affrontato in questa tesi, ciascuna particella rappresenta una specifica configurazione dei parametri del planner locale di navigazione. In particolare, la posizione della particella nello spazio di ricerca coincide con un vettore dei parametri da ottimizzare, mentre a ogni configurazione corrisponde un valore della funzione di costo ottenuto mediante l'esecuzione di un trial completo nella simulazione. Lo script di ottimizzazione basato su PSO agisce infatti proprio su un vettore di sei parametri del planner locale e delle costmap, comprendente la velocità massima lineare, il limite di accelerazione, il peso della distanza dagli ostacoli, il raggio di inflazione e i due parametri di valutazione delle traiettorie del planner DWA.

Dal punto di vista matematico, si consideri uno sciame costituito da N particelle. La particella i-esima, alla generica iterazione k, è descritta da una posizione $\mathrm{x}_i^{(k)}$ e da una velocità $\mathrm{v}_i^{(k)}$. L'evoluzione dello sciame avviene aggiornando iterativamente queste due grandezze secondo le relazioni che definiscono il PSO:

$$\mathrm{v}_i^{(k+1)} = w\,\mathrm{v}_i^{(k)} + c_1 r_1 \left(\mathrm{p}_i - \mathrm{x}_i^{(k)}\right) + c_2 r_2 \left(\mathrm{g} - \mathrm{x}_i^{(k)}\right)$$

$$\mathrm{x}_i^{(k+1)} = \mathrm{x}_i^{(k)} + \mathrm{v}_i^{(k+1)}$$

dove:

- w è il coefficiente di inerzia, che regola la tendenza della particella a mantenere la direzione di moto precedente;
- c_1 è il coefficiente cognitivo, che pondera l'attrazione verso la migliore posizione trovata dalla particella stessa;
- c_2 è il coefficiente sociale, che pondera l'attrazione verso la migliore posizione trovata dall'intero sciame;

- r_1 e r_2 sono variabili casuali uniformemente distribuite in $[0,1]$;
- p_i rappresenta la miglior posizione personale (*personal best*) trovata dalla particella i;
- g rappresenta la miglior posizione globale (*global best*) trovata dallo sciame.

L'aggiornamento della velocità può essere interpretato come la combinazione di tre contributi distinti. Il primo è il termine d'inerzia, che favorisce la continuità del moto nello spazio dei parametri e impedisce che la ricerca diventi eccessivamente erratica. Il secondo è il termine cognitivo, che spinge ciascuna particella a ritornare verso regioni dello spazio in cui essa ha già osservato buone prestazioni. Il terzo è il termine sociale, che introduce la condivisione dell'informazione tra le particelle e orienta l'intero sciame verso la miglior soluzione globale individuata fino a quel momento.

Dal punto di vista operativo, il funzionamento dell'algoritmo può essere descritto come segue. Inizialmente viene generata una popolazione di particelle distribuite nello spazio dei parametri ammessi. Per ciascuna particella viene eseguito un trial di navigazione completo, ottenendo il valore della funzione di costo associato alla configurazione corrispondente. Sulla base di tale valore vengono aggiornati sia il miglior risultato individuale di ciascuna particella sia il miglior risultato globale dell'intero sciame. Successivamente le particelle modificano la propria velocità e la propria posizione secondo le equazioni sopra riportate, producendo un nuovo insieme di configurazioni candidate. Il processo viene ripetuto iterativamente fino al raggiungimento di un criterio di arresto, che può dipendere dal numero massimo di iterazioni, dal numero massimo di valutazioni della funzione obiettivo o dal raggiungimento di una soluzione soddisfacente.

Un aspetto importante del PSO è che esso non richiede alcuna informazione sul gradiente della funzione obiettivo. Tale caratteristica lo rende particolarmente adatto a problemi di ottimizzazione black-box, come quello affrontato in questo

lavoro, in cui il valore della funzione di costo è ottenuto esclusivamente eseguendo una simulazione del sistema robotico e non può essere espresso in forma chiusa. Inoltre, la presenza di penalità, abort condition e fenomeni di non linearità rende la funzione obiettivo non regolare e potenzialmente non derivabile, condizione in cui i metodi di ottimizzazione classici basati sul gradiente risultano poco adatti o direttamente inapplicabili.

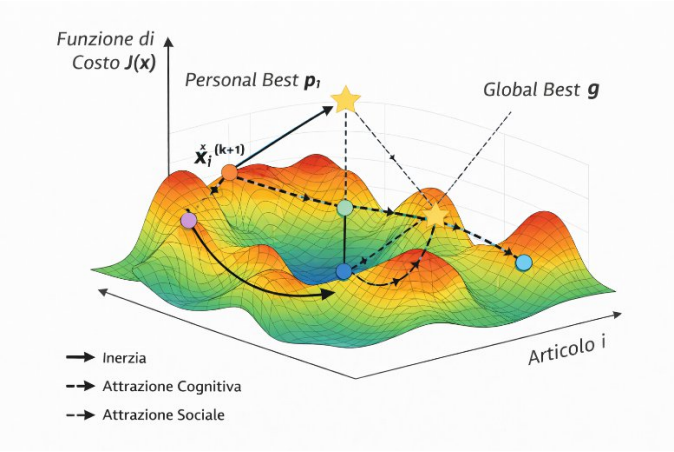
L'algoritmo PSO è stato scelto nel presente lavoro per diverse ragioni. In primo luogo, si tratta di un metodo relativamente semplice da implementare e da integrare in un framework sperimentale basato su simulazioni ripetute. In secondo luogo, PSO presenta un buon compromesso tra capacità di esplorazione globale e semplicità computazionale, risultando efficace in spazi di ricerca di dimensione moderata come quello considerato in questa tesi, costituito da un numero limitato di parametri ma caratterizzato da una funzione obiettivo complessa e rumorosa. Infine, il metodo è particolarmente adatto quando ogni valutazione della funzione di costo richiede l'esecuzione di un esperimento completo, poiché consente di esplorare progressivamente lo spazio dei parametri senza richiedere modelli analitici del sistema.

Nel caso specifico sviluppato in questa tesi, lo script PSO utilizza il Digital Twin per orchestrare ogni valutazione della funzione obiettivo. Per ciascuna particella, il sistema reimposta la simulazione, aggiorna dinamicamente i parametri del planner locale attraverso il meccanismo di *dynamic_reconfigure*, invia il goal di navigazione e monitora l'intera missione fino al completamento o all'interruzione del trial. Il valore della funzione di costo associato alla configurazione della particella viene quindi calcolato a partire dalle metriche di navigazione raccolte durante la prova. Questo schema rende il PSO un livello di decisione superiore rispetto alla simulazione e alla navigazione locale: esso non controlla direttamente il robot, ma seleziona progressivamente configurazioni parametriche in grado di migliorare il comportamento del sistema.

Dal punto di vista delle prestazioni, PSO presenta alcuni vantaggi rilevanti nel

problema considerato. La componente stocastica dell'algoritmo riduce il rischio di convergere prematuramente a soluzioni fortemente subottimali, mentre la memoria delle migliori soluzioni individuali e globali consente di sfruttare efficacemente l'informazione accumulata durante le iterazioni. Tuttavia, come molti metodi evolutivi e swarm-based, anche PSO può mostrare una certa sensibilità alla scelta degli iperparametri dell'algoritmo, come il peso d'inerzia e i coefficienti cognitivo e sociale. Una regolazione non adeguata di tali parametri può infatti portare a una ricerca eccessivamente dispersiva oppure, al contrario, a una convergenza troppo rapida e poco esplorativa.

Nonostante tali limiti, PSO rappresenta una scelta metodologicamente coerente con il problema affrontato in questa tesi. Il metodo consente infatti di affrontare in modo naturale un'ottimizzazione non lineare, rumorosa e priva di gradiente, mantenendo al tempo stesso una struttura algoritmica sufficientemente chiara da poter essere integrata senza ambiguità nel framework sperimentale sviluppato. Per queste ragioni, esso è stato adottato come primo algoritmo di riferimento per l'ottimizzazione dei parametri del planner locale, prima di introdurre e confrontare un secondo approccio evolutivo più sofisticato, descritto nel paragrafo successivo.



Rappresentazione del funzionamento dell'algoritmo Particle Swarm Optimization (PSO).

L'algoritmo PSO è stato implementato direttamente nello script Python del Digital Twin mediante un aggiornamento iterativo delle particelle secondo le

equazioni classiche del metodo. In particolare, lo script gestisce l'inizializzazione delle particelle, l'aggiornamento delle velocità e delle posizioni e la valutazione della funzione di costo tramite l'esecuzione dei trial di navigazione.

5.5.2 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

Il secondo metodo di ottimizzazione adottato nel presente lavoro è il **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)**, algoritmo evolutivo stocastico particolarmente adatto alla risoluzione di problemi di ottimizzazione continui, non lineari e privi di gradiente [15]. A differenza di approcci basati su regole di aggiornamento semplici, come nel caso del PSO, CMA-ES costruisce e aggiorna progressivamente una distribuzione di probabilità nello spazio dei parametri, utilizzandola per generare nuove soluzioni candidate e adattando nel tempo la direzione e l'ampiezza della ricerca.

L'idea di base del metodo consiste nel rappresentare la ricerca della soluzione ottima non come il moto di particelle nello spazio dei parametri, ma come l'evoluzione di una **distribuzione gaussiana multivariata** da cui vengono campionate, a ogni iterazione, le nuove configurazioni candidate. In questo modo, l'algoritmo non si limita a "muoversi" verso una migliore soluzione nota, ma apprende progressivamente la struttura locale del problema, adattando la forma della distribuzione in funzione delle regioni dello spazio dei parametri che producono valori più favorevoli della funzione obiettivo.

Dal punto di vista matematico, alla generica iterazione k, CMA-ES genera λ soluzioni candidate secondo la relazione:

$$\mathbf{x}_j^{(k)} = \mathbf{m}^{(k)} + \sigma^{(k)} \mathcal{N} \left(\mathbf{0}, \mathbf{C}^{(k)}\right), j = 1, \ldots, \lambda$$

dove:

- $\mathbf{m}^{(k)}$ è il **vettore medio** della distribuzione alla iterazione k, che rappresenta il centro corrente della ricerca;
- $\sigma^{(k)}$ è il **passo globale di ricerca** (*step size*), che controlla l'ampiezza complessiva delle variazioni generate;
- $\mathbf{C}^{(k)}$ è la **matrice di covarianza**, che descrive forma, orientazione e correlazioni della distribuzione nello spazio dei parametri;
- $\mathcal{N}\left(\mathbf{0}, \mathbf{C}^{(k)}\right)$ indica una variabile gaussiana multivariata a media nulla e covarianza $\mathbf{C}^{(k)}$.

Ogni campione $\mathbf{x}_j^{(k)}$ corrisponde quindi a una configurazione dei parametri del planner locale, che nel contesto di questa tesi viene valutata eseguendo un trial completo di navigazione tramite il Digital Twin. Per ciascuna configurazione viene calcolato il valore della funzione di costo e, al termine della valutazione della popolazione, le soluzioni vengono ordinate in base alla qualità ottenuta.

L'algoritmo aggiorna poi il centro della ricerca calcolando una media pesata delle migliori μ soluzioni della popolazione. Indicando con $\mathbf{x}_{i:\lambda}^{(k)}$ la i-esima migliore soluzione tra le λ campionate, l'aggiornamento del vettore medio può essere scritto come:

$$\mathbf{m}^{(k+1)} = \sum_{i=1}^{\mu} w_i \, \mathbf{x}_{i:\lambda}^{(k)}$$

dove i coefficienti w_i rappresentano i pesi associati alle migliori soluzioni e soddisfano tipicamente la condizione:

$$\sum_{i=1}^{\mu} w_i = 1$$

In questo modo il nuovo centro della distribuzione viene spostato verso le regioni dello spazio dei parametri che hanno prodotto i valori più bassi della funzione di costo.

L'elemento distintivo del CMA-ES rispetto ad altri metodi evolutivi è però l'aggiornamento della **matrice di covarianza**. Tale matrice consente all'algoritmo di apprendere quali direzioni nello spazio dei parametri risultino più promettenti e quali variabili presentino correlazioni significative. Se, ad esempio, due parametri tendono a migliorare la funzione obiettivo quando vengono modificati congiuntamente in una certa direzione, questa informazione viene progressivamente incorporata nella covarianza della distribuzione. In termini intuitivi, la nuvola di campionamento non rimane isotropa, ma si deforma e si orienta lungo le direzioni più favorevoli alla discesa della funzione di costo.

L’aggiornamento della matrice di covarianza può essere espresso in forma semplificata come:

$$\mathbf{C}^{(k+1)} = (1 - c_c)\mathbf{C}^{(k)} + c_c\, \mathbf{C}_{\text{new}}$$

dove c_cè un coefficiente di apprendimento e $\mathbf{C}_{\text{new}}$rappresenta il contributo derivante dalle migliori soluzioni osservate alla iterazione corrente. Nella formulazione completa del CMA-ES, tale aggiornamento è arricchito dall’utilizzo di percorsi evolutivi (*evolution paths*) che accumulano memoria della direzione di miglioramento nel corso delle iterazioni, rendendo l’adattamento più stabile ed efficace.

Oltre alla covarianza, anche il passo globale di ricerca σ viene aggiornato automaticamente. Questa caratteristica è fondamentale, perché consente all’algoritmo di iniziare con un’esplorazione relativamente ampia dello spazio dei parametri e di restringere progressivamente la ricerca man mano che la distribuzione si concentra attorno a regioni promettenti. Il meccanismo di adattamento del passo evita quindi sia una ricerca eccessivamente dispersiva, sia una convergenza troppo precoce.

Dal punto di vista operativo, il funzionamento di CMA-ES nel presente lavoro può essere riassunto nel seguente schema iterativo. Si parte da una configurazione iniziale dei parametri, che costituisce il centro iniziale della distribuzione, e da un passo iniziale di esplorazione. A ogni iterazione vengono campionate diverse configurazioni candidate del planner locale; per ciascuna configurazione il Digital Twin reimposta la simulazione, aggiorna i parametri del planner DWA e delle costmap, esegue la missione di navigazione e calcola il valore della funzione di

costo. Sulla base dei risultati ottenuti, CMA-ES aggiorna la media della distribuzione, la matrice di covarianza e il passo di ricerca, producendo una nuova popolazione di configurazioni candidate.

CMA-ES è stato scelto nel presente lavoro per diverse ragioni metodologiche. In primo luogo, esso è particolarmente efficace nei problemi di ottimizzazione **continui** con un numero moderato di variabili, come nel caso del vettore dei parametri del planner locale. In secondo luogo, il metodo è noto per la sua robustezza in presenza di funzioni obiettivo **non lineari, multimodali e rumorose**, caratteristiche tipiche del problema affrontato in questa tesi. La funzione di costo, infatti, non deriva da un'espressione analitica regolare, ma dall'esecuzione di trial simulati in cui il comportamento del robot è influenzato da dinamiche di navigazione, soglie di sicurezza, abort condition e penalità. In un contesto di questo tipo, la possibilità di adattare automaticamente sia la direzione della ricerca sia l'ampiezza delle perturbazioni rappresenta un vantaggio significativo.

Un ulteriore aspetto favorevole di CMA-ES riguarda la sua capacità di gestire in maniera naturale le **correlazioni tra parametri**. Nel problema considerato, i parametri ottimizzati non agiscono in modo indipendente: ad esempio, la velocità massima, i limiti di accelerazione e i pesi di valutazione del planner locale concorrono congiuntamente a determinare il comportamento del robot. Un metodo che apprende tali correlazioni può quindi risultare più efficace rispetto ad approcci che trattano implicitamente i parametri come dimensioni indipendenti.

Rispetto al PSO, CMA-ES presenta una struttura più complessa e un costo computazionale per iterazione generalmente superiore, dovuto all'aggiornamento della matrice di covarianza e alla gestione della distribuzione multivariata. Tuttavia, questa maggiore complessità è compensata da una migliore capacità di adattamento alla geometria locale del problema e da una maggiore robustezza in scenari caratterizzati da superfici di costo irregolari. Per questo motivo, nel presente lavoro CMA-ES è stato adottato come secondo algoritmo di

ottimizzazione, con l'obiettivo di confrontarne le prestazioni con quelle del PSO nello stesso scenario sperimentale e sul medesimo insieme di parametri del planner.

In sintesi, CMA-ES rappresenta una scelta particolarmente coerente con il problema affrontato in questa tesi: esso consente di affrontare un'ottimizzazione continua, priva di gradiente, rumorosa e caratterizzata da interazioni non banali tra i parametri, mantenendo un elevato livello di adattività durante l'esplorazione dello spazio di ricerca. Per queste ragioni, il metodo costituisce un riferimento particolarmente significativo nel confronto sperimentale sviluppato nel seguito del capitolo.

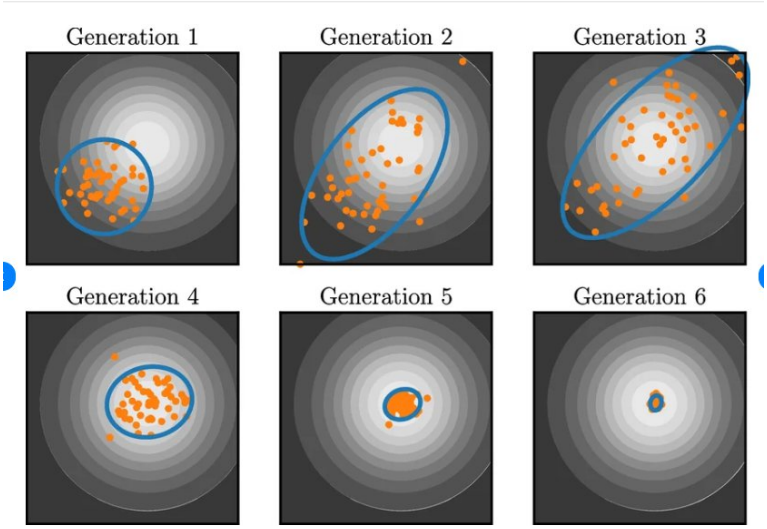


Illustrazione del principio di funzionamento del metodo CMA-ES

Per quanto riguarda CMA-ES, l'implementazione è stata realizzata utilizzando la libreria Python cma, che fornisce una implementazione efficiente dell'algoritmo. Lo script utilizza le funzioni *CMAEvolutionStrategy*, *ask*() e *tell*() per generare le

soluzioni candidate, valutarle tramite il Digital Twin e aggiornare iterativamente la distribuzione gaussiana utilizzata per l'esplorazione dello spazio dei parametri.

5.6 Algoritmo di recovery basato su Artificial Potential Field

Nel corso delle prove sperimentali relative allo Scenario 2 è emersa la necessità di introdurre un meccanismo di recovery dedicato alla gestione delle situazioni in cui il planner locale e globale non risultavano temporaneamente in grado di garantire un avanzamento affidabile del robot. In particolare, durante la navigazione potevano verificarsi condizioni di prossimità eccessiva a un ostacolo, stallo cinematico, rotazioni improduttive o assenza di progresso verso il goal. Per affrontare tali condizioni è stata implementata una strategia di recovery basata su Artificial Potential Field (APF), richiamata solo quando il normale ciclo di navigazione con move_base veniva interrotto dal monitor di esecuzione. Nello script dei twin, infatti, il recovery non coincide con il planner principale, ma è realizzato tramite una classe dedicata, *LocalPotentialFieldAvoider*, attivata dopo l'annullamento del goal corrente e prima del reinvio del goal a move base. Questa logica è presente sia nella versione PSO sia nella versione CMA-ES del twin, mantenendo invariato il meccanismo di recovery e differenziando esclusivamente l'algoritmo di ottimizzazione dei parametri.

L'Artificial Potential Field è una tecnica classica di navigazione reattiva in cui il robot viene modellato come una particella soggetta all'azione simultanea di un campo attrattivo, diretto verso l'obiettivo, e di un campo repulsivo, generato dagli ostacoli percepiti [18][19]. Il principio è semplice ma molto efficace nei contesti

locali: il robot non pianifica una traiettoria globale complessa, bensì calcola istante per istante una direzione di moto risultante dalla composizione vettoriale delle forze artificiali agenti su di esso. La forza totale può essere espressa, in forma generale, come

$$\mathrm{F} = \mathrm{F}_{att} + \mathrm{F}_{rep}$$

dove F_{att} rappresenta il contributo che spinge il robot verso il goal e F_{rep} il contributo che lo allontana dagli ostacoli circostanti. Nel caso in esame, questa metodologia è stata scelta non come soluzione completa di navigazione, ma come strumento di disimpegno locale da utilizzare in fase di recovery, cioè in un intervallo limitato di tempo e solo in presenza di criticità rilevate dal sistema di monitoraggio.

La scelta dell'APF per questo problema è motivata da diverse ragioni. In primo luogo, si tratta di un metodo reattivo, quindi adatto a intervenire rapidamente in situazioni locali non previste o mal gestite dal planner corrente. In secondo luogo, il costo computazionale è molto contenuto, poiché il calcolo si basa direttamente sulle misure del sensore laser e sulla posizione del goal, senza richiedere la ricostruzione di una nuova traiettoria globale. Inoltre, per uno scenario di recovery come quello qui considerato, l'obiettivo non è trovare il percorso ottimo in senso globale, ma semplicemente ottenere una manovra robusta che consenta al robot di uscire da una configurazione critica e restituire poi il controllo al sistema di navigazione principale. Infine, l'APF si integra in modo naturale con il framework ROS già sviluppato, poiché può utilizzare direttamente *i topic /scan* e */odom* per ricavare rispettivamente la geometria locale degli ostacoli e la posa corrente del robot. Questa scelta è dunque coerente con l'architettura complessiva del sistema e con il ruolo ausiliario assegnato al recovery nello Scenario 2.

Dal punto di vista implementativo, la componente attrattiva è ottenuta a partire dalla differenza tra la posizione del goal p_g e la posizione corrente del robot p. Indicando con $\mathrm{d} = \mathrm{p}_g - \mathrm{p}$ il vettore che punta dal robot verso il goal, la forza attrattiva è definita nella forma

$$\mathrm{F}_{att} = k_{att} \frac{\mathrm{d}}{\| \mathrm{d} \|}$$

dove k_{att} è un coefficiente di guadagno positivo. Questa formulazione, adottata nel codice in forma normalizzata, ha il vantaggio di mantenere stabile la direzione verso l'obiettivo indipendentemente dalla distanza, evitando che la magnitudine della forza cresca eccessivamente all'aumentare della separazione dal goal. In pratica, il campo attrattivo agisce come un vettore unitario orientato verso la destinazione, opportunamente scalato da un parametro di tuning.

La componente repulsiva è invece ricavata dalle misure del *LaserScan*. Ogni raggio del lidar, se intercetta un ostacolo entro un certo raggio di influenza R_{rep}, contribuisce alla generazione di una forza opposta alla direzione dell'ostacolo stesso. In forma semplificata, per ciascuna misura valida r_i inferiore a R_{rep}, si può descrivere un contributo repulsivo come

$$\mathrm{F}_{rep,i} = - \, k_{rep} \left(\frac{1}{r_i} - \frac{1}{R_{rep}} \right) \begin{bmatrix} \cos\theta_i \\ \sin\theta_i \end{bmatrix}$$

dove k_{rep} è il guadagno repulsivo e θ_i l'angolo del raggio nel sistema di riferimento del robot. La forza repulsiva complessiva è quindi ottenuta sommando i contributi dei singoli raggi:

$$\mathrm{F}_{rep} = \sum_i \mathrm{F}_{rep,i}$$

Questa formulazione consente di attribuire un peso maggiore agli ostacoli più vicini, producendo un effetto di allontanamento crescente quando il robot si avvicina a regioni pericolose. Nel codice del twin viene infatti verificato, per ogni misura del lidar, se essa rientra nel raggio di influenza repulsiva; in caso positivo, viene accumulato un contributo vettoriale negativo rispetto alla direzione del raggio.

Una volta calcolata la forza risultante F, il sistema ne ricava la direzione desiderata di avanzamento. Se $\phi = \mathrm{atan2}\left(F_y,F_x\right)$ indica l'angolo della forza totale nel piano, e ψ l'orientamento corrente del robot, l'errore angolare può essere espresso come

$$e_\psi = \mathrm{atan2}\left(\sin\,(\phi-\psi),\cos\,(\phi-\psi)\right)$$

in modo da mantenere l'errore nell'intervallo $(-\pi,\pi)$. La velocità angolare viene quindi calcolata con una legge proporzionale del tipo

$$\omega = k_\psi e_\psi$$

saturata entro un valore massimo compatibile con il robot. La velocità lineare, invece, viene modulata in funzione dell'allineamento con la direzione desiderata. Nel codice è utilizzata una relazione del tipo

$$v = v_{max} \max\,(0,\cos\, e_{\psi})$$

successivamente limitata tra un valore minimo e uno massimo. Questa scelta è tecnicamente molto opportuna: se il robot è fortemente disallineato rispetto alla direzione del campo, la componente $\cos e_{\psi}$ diminuisce e quindi la velocità lineare si riduce, favorendo la rotazione prima dell'avanzamento; viceversa, quando il robot è ben orientato verso la direzione risultante del campo, può avanzare più rapidamente. In questo modo il recovery evita comportamenti troppo bruschi e mantiene una logica di controllo coerente con la cinematica differenziale del TurtleBot3.

Un aspetto importante del metodo implementato è che esso non viene lasciato operare indefinitamente. L'APF è stato progettato come meccanismo temporaneo di recovery locale, con durata massima limitata e con controlli specifici sul progresso. Nello script sono presenti soglie di *timeout*, criteri di *stuck detection* e una manovra di *escape rotation* qualora il robot non registri un avanzamento sufficiente per un certo intervallo temporale. In altri termini, se il campo potenziale non riesce a produrre un disimpegno utile entro il tempo assegnato, il recovery viene considerato fallito. Se invece il robot riesce a superare la zona critica, il sistema arresta brevemente il robot, pulisce le costmap e riaffida il controllo a move_base, reinviando il goal originale. Questa logica è particolarmente importante perché mostra chiaramente che l'APF non sostituisce

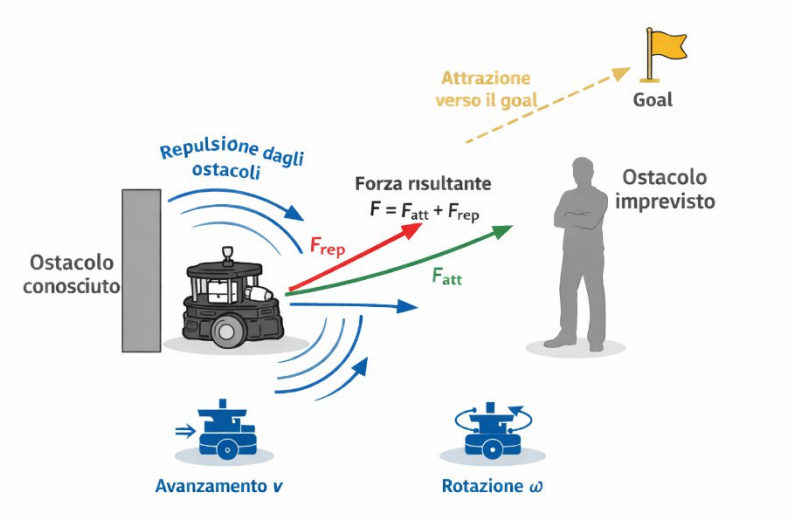
né il planner globale né il planner locale standard, ma agisce come livello reattivo di supporto, confinato alle sole fasi di recupero.

Dal punto di vista metodologico, questa soluzione è particolarmente adatta allo Scenario 2 per almeno tre motivi. Il primo è che lo scenario presenta un ambiente più complesso rispetto alle prove preliminari, con maggiore probabilità di configurazioni locali sfavorevoli e di interazioni ravvicinate con ostacoli o restringimenti. Il secondo è che, in presenza di ottimizzazione automatica dei parametri di navigazione, alcune configurazioni esplorate da PSO o CMA-ES possono rendere il comportamento del planner più aggressivo o meno robusto; disporre di un recovery reattivo separato consente di limitare l'impatto di tali configurazioni e di rendere l'esperimento più stabile. Il terzo è che l'APF, operando in modo diretto su pose e scansioni, risulta indipendente dalla logica interna dell'algoritmo di ottimizzazione e può quindi essere mantenuto identico nei due twin, garantendo omogeneità sperimentale nel confronto tra PSO e CMA-ES.

Va tuttavia sottolineato che gli Artificial Potential Fields presentano anche limiti noti in letteratura [18], come la possibilità di incorrere in minimi locali, oscillazioni in ambienti molto stretti o comportamenti non ottimali in presenza di geometrie particolarmente sfavorevoli. Nel contesto di questa tesi, però, tali limiti risultano accettabili, poiché l'APF non è stato impiegato come algoritmo di navigazione completo, ma esclusivamente come recovery locale e temporaneo. In questa funzione, i suoi vantaggi in termini di semplicità, rapidità di calcolo e capacità di reagire immediatamente alle condizioni del sensore superano le sue debolezze teoriche [19]. Per questo motivo, l'adozione dell'APF nello Scenario 2 rappresenta un compromesso efficace tra robustezza operativa e contenimento della complessità implementativa.

In conclusione, la strategia di recovery mediante Artificial Potential Field ha costituito un elemento importante dell'architettura sperimentale dello Scenario 2. Essa ha permesso di gestire condizioni critiche di navigazione che il normale

stack move_base non riusciva a risolvere autonomamente, fornendo al robot una capacità di reazione locale rapida e computazionalmente leggera. L'integrazione di tale meccanismo all'interno dei twin ha migliorato la continuità delle prove sperimentali e ha reso più affidabile il confronto tra i due algoritmi di ottimizzazione, mantenendo invariata la logica di recupero in entrambe le configurazioni sperimentali.



Schema del metodo Artificial Potential Field (APF) per l'evitamento degli ostacoli.

Capitolo 6

Valutazione sperimentale

6.1 Scenario 1

Il primo scenario sperimentale è stato progettato per valutare le prestazioni degli algoritmi di ottimizzazione all'interno di un ambiente strutturato che riproduce un tipico contesto di logistica industriale. L'ambiente simulato rappresenta un magazzino con corsie delimitate da scaffalature e con la presenza di alcuni ostacoli statici distribuiti lungo il percorso. Come mostrato in Figura, il robot deve raggiungere una posizione di goal G partendo da una posizione iniziale S, attraversando diverse corsie e affrontando cambi di direzione e possibili zone di congestione dovute alla disposizione degli scaffali e degli oggetti presenti.

Questo scenario è stato scelto per diverse ragioni. In primo luogo rappresenta una configurazione realistica di navigazione autonoma in ambienti interni strutturati, dove i robot mobili sono comunemente impiegati per attività di trasporto e movimentazione di materiali. In secondo luogo il percorso include una combinazione di tratti rettilinei, cambi di direzione e passaggi relativamente stretti tra gli ostacoli, rendendolo adatto a valutare sia la velocità di percorrenza sia la capacità del sistema di mantenere margini di sicurezza adeguati rispetto agli ostacoli.

Al fine di garantire la ripetibilità dei test e isolare il comportamento degli algoritmi di ottimizzazione, durante gli esperimenti è stato disattivato il comportamento di

navigazione reattiva basato sull'algoritmo **APF**, che verrà approfondito in seguito, normalmente utilizzato per la gestione di ostacoli imprevisti. In questo modo il robot opera esclusivamente all'interno di uno scenario con ostacoli noti e statici, permettendo di valutare in maniera controllata l'effetto della regolazione dei parametri del planner locale sulla qualità della navigazione. Tale scelta consente di evitare variazioni imprevedibili del comportamento del robot e rende i risultati più comparabili tra i diversi trial.

Per ciascun algoritmo di ottimizzazione sono state implementate tre modalità operative distinte: **baseline**, **optimization** e **best replay**. Questa suddivisione consente di separare chiaramente le diverse fasi dell'esperimento e di garantire una valutazione corretta dei risultati.

La modalità **baseline** rappresenta la configurazione di riferimento del sistema. In questa modalità il robot esegue la missione utilizzando i parametri standard del planner senza alcun processo di ottimizzazione. I trial baseline permettono quindi di stabilire il livello prestazionale iniziale del sistema, che verrà successivamente confrontato con quello ottenuto tramite l'ottimizzazione.

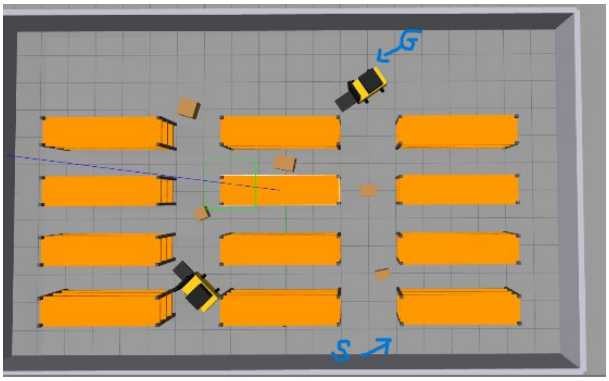
La modalità **optimization** costituisce la fase principale dell'esperimento. In questa modalità l'algoritmo di ottimizzazione (PSO o CMA-ES) esplora lo spazio dei parametri del planner locale eseguendo una sequenza di trial nel simulatore. Per ogni trial viene calcolata una funzione di costo che sintetizza diverse metriche di prestazione, tra cui il tempo di completamento della missione, la distanza minima dagli ostacoli e la stabilità del moto. I parametri che producono valori di costo più bassi vengono progressivamente selezionati e utilizzati per guidare la ricerca verso configurazioni sempre più performanti. Durante questa fase vengono registrati tutti i dati necessari all'analisi statistica dei risultati.

Infine, la modalità **best replay** viene utilizzata per validare la soluzione migliore individuata durante il processo di ottimizzazione. In questa modalità il robot esegue nuovamente la missione utilizzando il set di parametri ottimali trovato

dall'algoritmo, consentendo di verificare che il miglioramento osservato non sia dovuto a variazioni casuali del comportamento del sistema ma sia effettivamente riproducibile.

L'intero processo di ottimizzazione è stato eseguito separatamente per ciascun algoritmo. Per ogni metodo sono stati effettuati **80 trial di ottimizzazione**, con una durata complessiva di circa **un'ora per algoritmo**, includendo il tempo necessario all'esecuzione delle simulazioni e alla registrazione dei dati. Tutti i risultati sperimentali sono stati salvati automaticamente dal digital twin sotto forma di file CSV e grafici riassuntivi generati dagli script Python sviluppati per questo progetto. Gli script responsabili dell'esecuzione dei trial, della registrazione dei dati e della generazione delle statistiche sono descritti nel Capitolo 5 e riportati integralmente nell'Appendice.

Questo scenario costituisce quindi il banco di prova principale per valutare l'efficacia dei due algoritmi di ottimizzazione considerati, consentendo di analizzare in modo sistematico il miglioramento ottenuto rispetto alla configurazione baseline e di confrontare le prestazioni dei diversi metodi.



Percorso Scenario 1

Nel processo di ottimizzazione sono stati considerati sei parametri del sistema di navigazione basato su **DWA local planner** e costmap. Per ciascun parametro è

stato definito un intervallo di ricerca all'interno del quale gli algoritmi di ottimizzazione (CMA-ES e PSO) possono esplorare possibili configurazioni.

La scelta dei limiti è stata effettuata considerando:

- i **limiti fisici del robot TurtleBot3 Burger**,
- i valori comunemente utilizzati nella configurazione del **DWA planner in ROS**,
- la necessità di evitare configurazioni instabili durante i trial di navigazione.

I parametri ottimizzati e i relativi bounds sono i seguenti:

- **max_vel_x**

  $$-0.10 \leq max\_vel\_x \leq 0.22 \text{m/s}$$

  Il limite superiore corrisponde alla velocità massima del TurtleBot3 Burger, mentre il limite inferiore consente brevi manovre di retrocessione utili nelle situazioni di disimpegno.
- **acc_lim_x**

  $$0.30 \leq acc\_lim\_x \leq 1.20 \text{m/s}^2$$

  Il parametro controlla la reattività dinamica del robot. L'intervallo consente di esplorare comportamenti compresi tra guida conservativa e guida più aggressiva.
- **occdist_scale**

  $$0.01 \leq occdist\_scale \leq 0.12$$

  Determina il peso della distanza dagli ostacoli nella funzione di costo del planner, influenzando il compromesso tra sicurezza e velocità.
- **inflation_radius**

  $0.15 \leq inflation\_radius \leq 0.35 \text{m}$

  Definisce la distanza di sicurezza con cui gli ostacoli vengono inflazionati nella costmap.

- **path_distance_bias**

$$5.0 \leq path\_distance\_bias \leq 60.0$$

  Controlla quanto il robot tende a seguire il percorso globale pianificato.
- **goal_distance_bias**

$$5.0 \leq goal\_distance\_bias \leq 40.0$$

  Regola quanto il planner privilegia traiettorie che riducono la distanza dal goal.

Gli stessi intervalli di ricerca sono stati utilizzati per entrambi gli algoritmi di ottimizzazione al fine di garantire un confronto equo tra i metodi.

6.1.1 Ottimizzazione con PSO

In questa sezione vengono presentati i risultati ottenuti applicando l'algoritmo **Particle Swarm Optimization (PSO)** all'ottimizzazione dei parametri del planner locale nello scenario descritto nella Sezione 6.1.

Gli esperimenti sono stati eseguiti utilizzando il digital twin sviluppato nel Capitolo 5, che consente di eseguire automaticamente sequenze di trial nel simulatore e registrare le prestazioni del robot per ciascuna configurazione dei parametri.

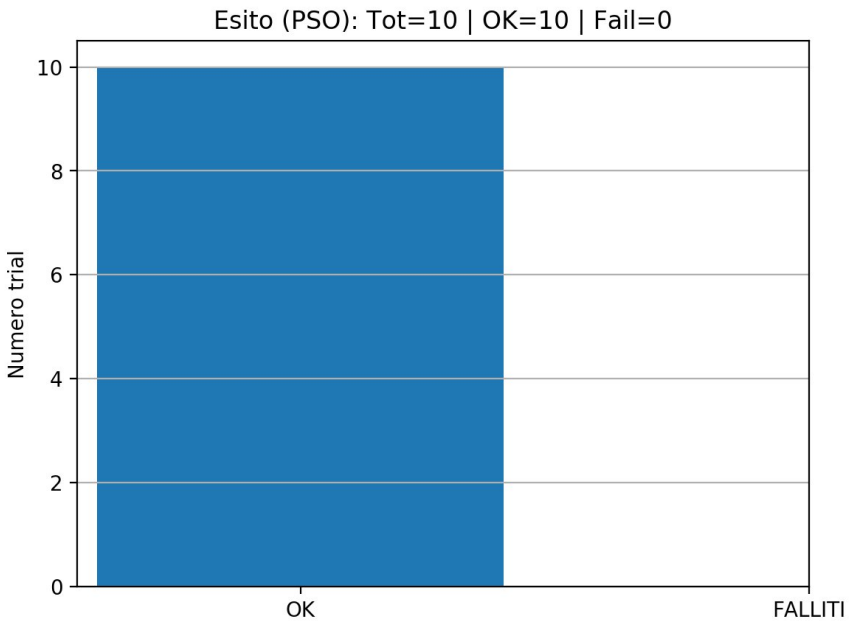
Per l'algoritmo PSO sono state considerate le tre modalità operative:

- **baseline**, che rappresenta la configurazione iniziale del sistema;
- **optimization**, in cui l'algoritmo esplora lo spazio dei parametri per minimizzare la funzione di costo;
- **best replay**, utilizzata per rieseguire la missione utilizzando il miglior set di parametri individuato.

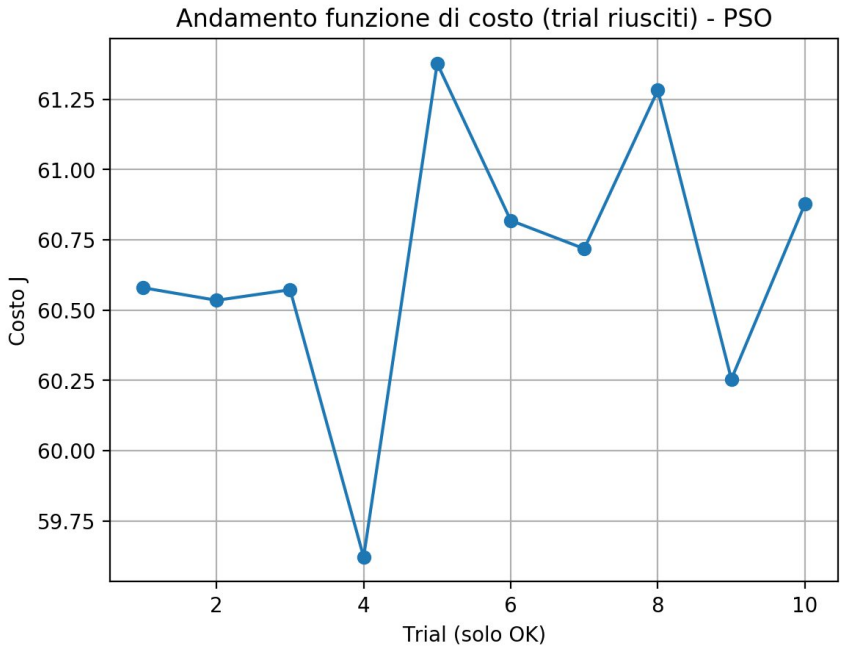
Le prestazioni vengono valutate utilizzando le metriche definite nel Capitolo 5, tra cui il tempo di missione, la distanza minima dagli ostacoli e la stabilità del movimento.

Modalità baseline

La modalità baseline rappresenta il comportamento del sistema utilizzando i parametri standard del planner locale, senza alcun processo di ottimizzazione.



Scenario1 PSO baseline andamento trial



Scenario1 PSO baseline andamento J

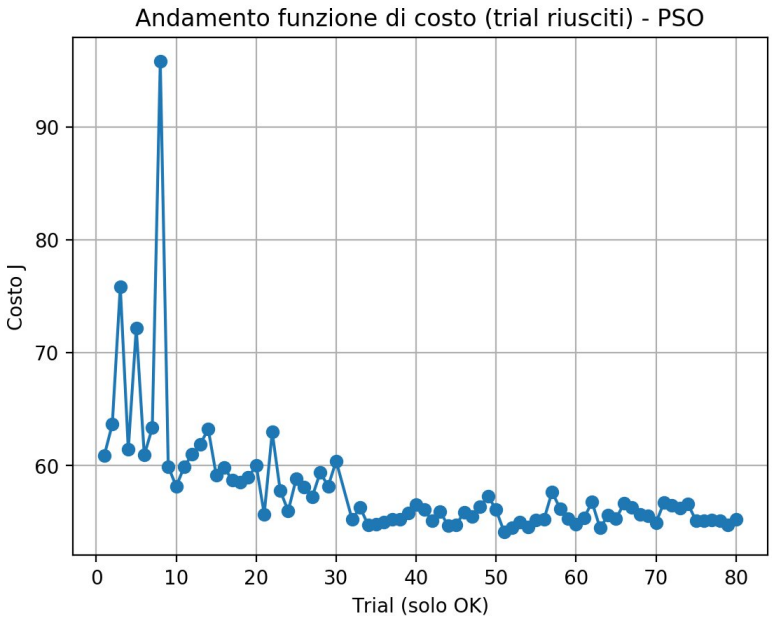
La Figura in alto riporta l'esito dei trial eseguiti in modalità baseline. Tutti i test sono stati completati con successo, con 10 missioni riuscite su 10, senza fallimenti o condizioni di arresto anticipato.

La Figura in basso mostra invece l'andamento della funzione di costo Jnei **10 trial riusciti** eseguiti nella configurazione baseline. I valori risultano compresi in un intervallo relativamente ristretto (circa **59.6 – 61.4**), indicando un comportamento stabile e ripetibile del sistema nella configurazione iniziale.

Modalità Optimization

Durante la fase di optimization, l'algoritmo PSO esplora lo spazio dei parametri

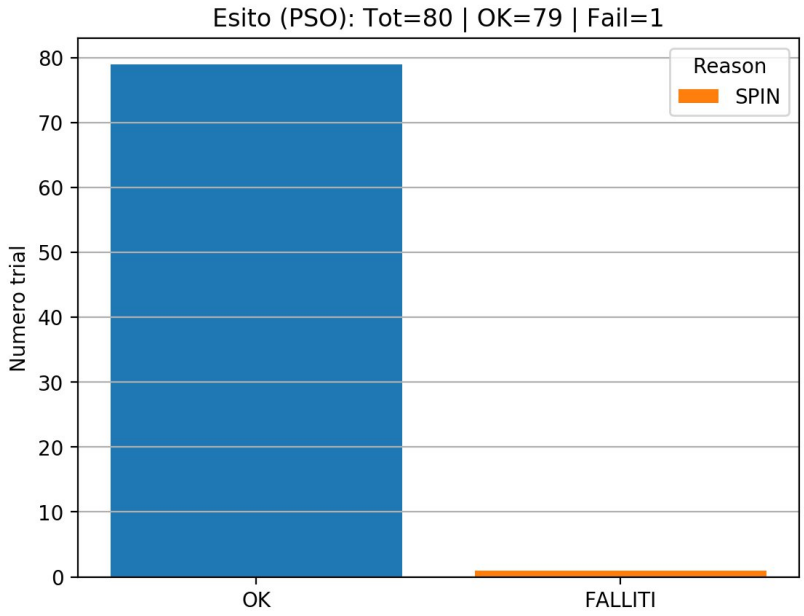
del planner locale eseguendo 80 trial. La Figura mostra l'andamento della funzione di costo Jper i trial riusciti.



Scenario1 PSO baseline andamento J

Nelle prime iterazioni si osserva una maggiore variabilità dei valori di costo, dovuta alla fase di esplorazione iniziale dell'algoritmo. Successivamente i valori tendono progressivamente a ridursi e stabilizzarsi intorno a valori inferiori rispetto alla configurazione baseline, indicando la convergenza verso configurazioni più performanti.

Qui invece possiamo osservare l'esito dei trial eseguiti durante l'ottimizzazione. Su 80 trial complessivi, 79 sono stati completati con successo, mentre un trial è terminato con fallimento, associato a una condizione di spin del robot.



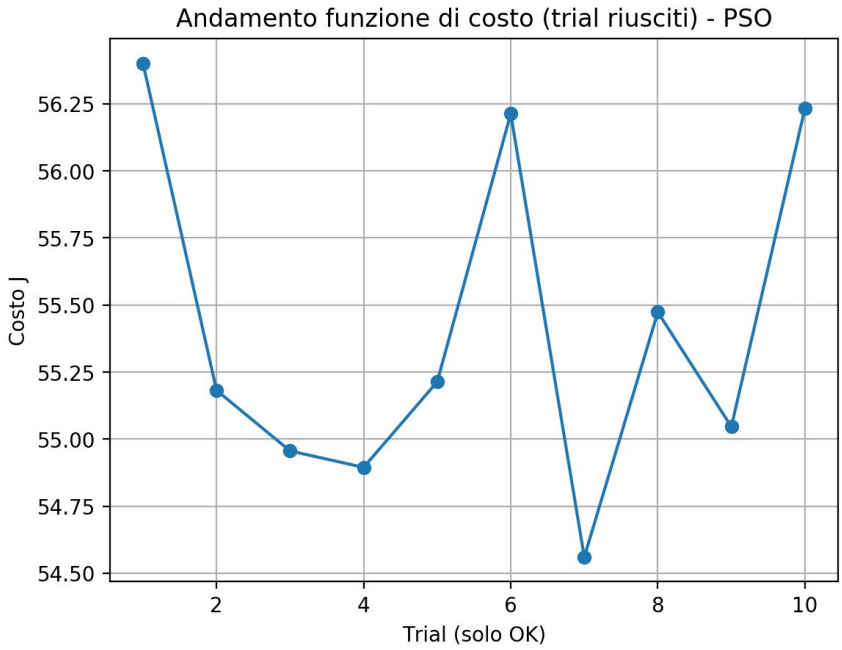
Scenario1 PSO baseline andamento Trial

Il processo di ottimizzazione ha individuato un set di parametri in grado di migliorare le prestazioni di navigazione, con un tempo di missione minimo pari a 51.71 s e un valore minimo della funzione di costo Jpari a circa 54.12.

Modalità Best replay

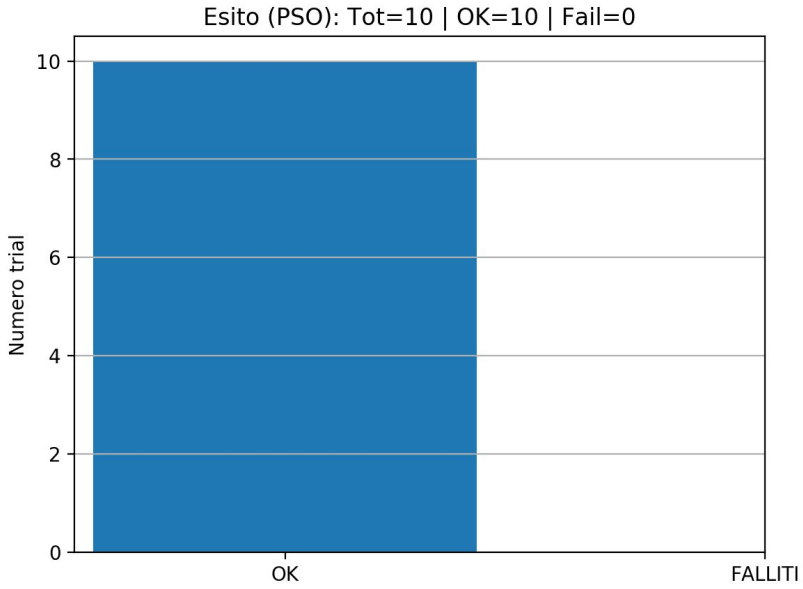
La modalità best replay viene utilizzata per verificare le prestazioni del robot utilizzando il miglior set di parametri individuato durante la fase di ottimizzazione.

In figura possiamo osservare l'andamento della funzione di costo J nei 10 trial eseguiti utilizzando i parametri ottimizzati. I valori risultano complessivamente inferiori rispetto alla configurazione baseline, confermando il miglioramento ottenuto attraverso il processo di ottimizzazione



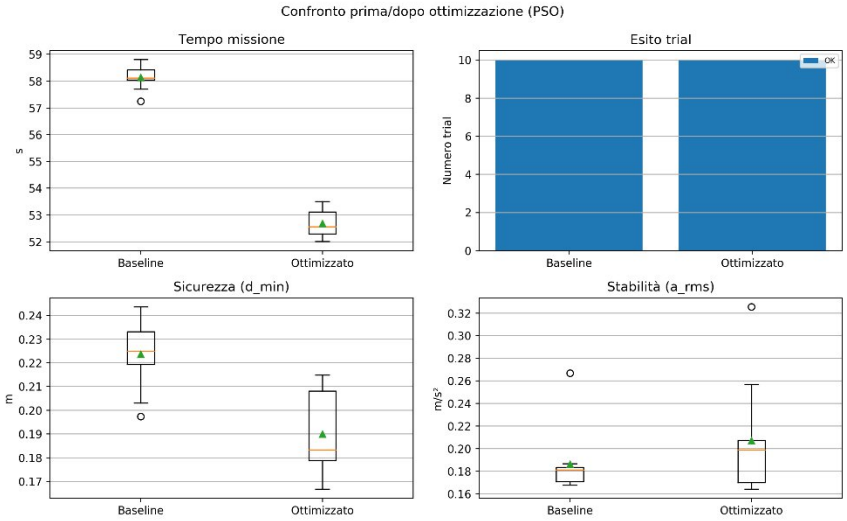
Scenario1 PSO best replay andamento J

Notiamo l'esito dei trial di validazione. Tutte le esecuzioni sono state completate con successo, con 10 missioni riuscite su 10, confermando la stabilità e la riproducibilità delle prestazioni ottenute con i parametri ottimizzati.



Scenario1 PSO best replay andamento J

La figura successiva riporta il confronto tra la configurazione baseline e la configurazione ottimizzata con PSO, valutate tramite i trial di validazione in modalità best replay. Il confronto evidenzia un miglioramento netto della prestazione temporale: il tempo mediano di missione passa infatti da 58.11 s nella configurazione baseline a 52.56 s nella configurazione ottimizzata, con una riduzione pari a circa 9.5%. Anche il valore medio del tempo di missione mostra una diminuzione analoga, confermando che il miglioramento non è dovuto a un singolo trial particolarmente favorevole ma rappresenta un effetto complessivo e ripetibile.



Scenario 1 PSO Baseline vs Best replay

Dal punto di vista dell'esito delle missioni, entrambe le configurazioni considerate nel confronto finale mostrano un success rate pari al 100% nei 10 trial di validazione, indicando che l'ottimizzazione non ha compromesso la capacità del robot di raggiungere il goal nello scenario assegnato. È tuttavia opportuno distinguere questo risultato dalla fase di optimization, nella quale il PSO ha registrato 79 successi su 80 trial, con un unico fallimento dovuto a una condizione di spin del robot. Questo aspetto suggerisce che, pur essendo efficace nel trovare configurazioni più rapide, il processo di esplorazione del PSO può occasionalmente attraversare regioni dello spazio dei parametri meno stabili.

L'analisi delle metriche di sicurezza e stabilità mostra un comportamento più articolato. In particolare, la distanza minima dagli ostacoli d_{min} si riduce da circa 0.225 m a circa 0.185 m, con una variazione negativa dell'ordine del 18%. Questo risultato indica che il robot, nella configurazione ottimizzata, tende a seguire traiettorie più aggressive o più aderenti agli ostacoli, probabilmente per ridurre il tempo complessivo di percorrenza. In altre parole, il miglioramento in velocità è ottenuto a scapito del margine di sicurezza.

Un andamento analogo si osserva per la metrica di stabilità dinamica. Il valore mediano di accelerazione RMS a_{rms} passa infatti da circa 0.18 m/s² a circa 0.20 m/s², con un incremento di circa 11%. Tale aumento suggerisce che il moto del robot risulta mediamente meno fluido dopo l'ottimizzazione, con variazioni di velocità più marcate lungo il percorso. Anche in questo caso, il comportamento è coerente con una configurazione che privilegia la rapidità di esecuzione rispetto alla regolarità del moto.

Complessivamente, i risultati mostrano che l'algoritmo PSO è stato in grado di individuare una configurazione del planner locale capace di ridurre sensibilmente il tempo di missione mantenendo elevata la riuscita dei test finali. Tuttavia, il miglioramento ottenuto non è bilanciato su tutte le metriche: alla maggiore efficienza temporale corrispondono infatti una riduzione della distanza minima dagli ostacoli e un peggioramento della stabilità dinamica. Si può quindi affermare che, nello Scenario 1, il PSO produce una soluzione efficace in termini di rapidità, ma caratterizzata da un compromesso meno favorevole sul piano della sicurezza e della fluidità del moto. Ecco una tabella riassuntiva dei risultati ottenuti con PSO

Metrica	Baseline	PSO ottimizzato	Variazione	Commento
Tempo missione mediano (s)	58.11	52.56	-9.5%	Miglioramento netto della rapidità di navigazione
Tempo missione medio (s)	circa 58.2	circa 52.7	-9.4%	Riduzione coerente anche sul valore medio
Success rate validazione	100% (10/10)	100% (10/10)	Invariato	L'ottimizzazione non peggiora l'esito finale dei test
d_min mediano (m)	circa 0.225	circa 0.185	-18%	Riduzione del margine di sicurezza rispetto agli ostacoli
a_rms mediano (m/s²)	circa 0.18	circa 0.20	+11%	Moto meno fluido, con variazioni dinamiche più marcate

6.1.2 Ottimizzazione con CMA-ES

In questa sezione vengono presentati i risultati ottenuti applicando l'algoritmo Covariance Matrix Adaptation Evolution Strategy (CMA-ES) all'ottimizzazione dei parametri del planner locale nello scenario descritto nella Sezione 6.1.

Come nel caso precedente, gli esperimenti sono stati eseguiti utilizzando il digital twin sviluppato nel Capitolo 5, che consente di eseguire automaticamente sequenze di trial nel simulatore Gazebo, modificando i parametri del planner locale e registrando le prestazioni del robot per ciascuna configurazione testata.

L'algoritmo CMA-ES esplora lo spazio dei parametri attraverso una strategia evolutiva basata sull'adattamento della matrice di covarianza della distribuzione delle soluzioni candidate. Questo meccanismo consente all'algoritmo di adattare progressivamente la direzione e l'ampiezza della ricerca nello spazio dei parametri, favorendo la convergenza verso regioni con valori di costo più bassi.

Analogamente al caso PSO, il processo sperimentale è stato suddiviso in tre modalità operative:

- baseline, che rappresenta il comportamento del sistema con i parametri standard del planner;
- optimization, in cui l'algoritmo CMA-ES esegue una sequenza di trial per minimizzare la funzione di costo J;
- best replay, utilizzata per validare le prestazioni ottenute con il miglior set di parametri individuato.

Le prestazioni del sistema vengono valutate utilizzando le metriche definite nel Capitolo 5, tra cui il tempo di missione, la distanza minima dagli ostacoli e la stabilità del moto, sintetizzate all'interno della funzione di costo utilizzata durante il processo di ottimizzazione.

Nelle sezioni successive vengono analizzati i risultati ottenuti nelle tre modalità considerate, con particolare attenzione al comportamento dell'algoritmo durante la fase di ottimizzazione e al confronto tra la configurazione baseline e quella ottimizzata.

Modalità baseline

Rivediamo la modalità **baseline** che rappresenta il comportamento del sistema utilizzando i parametri standard del planner locale, senza alcun processo di ottimizzazione.

La Figura X mostra l'andamento della funzione di costo Jnei **10 trial riusciti** eseguiti nella configurazione baseline. I valori risultano compresi in un intervallo relativamente ristretto, indicativamente tra **60.0 e 62.6**, indicando un comportamento complessivamente stabile e ripetibile del sistema nella configurazione iniziale

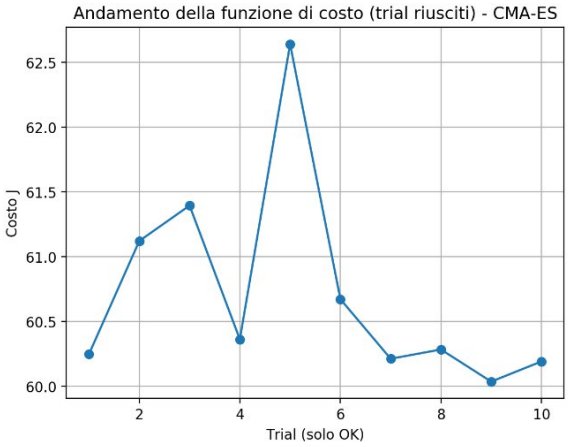
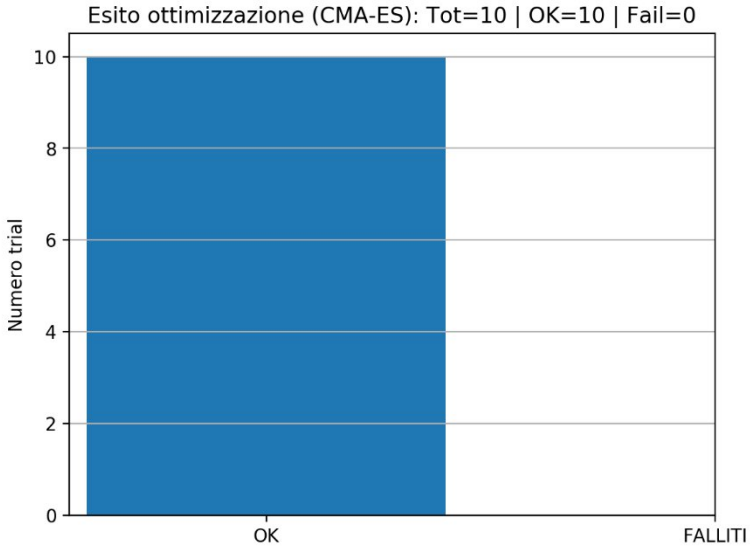


Figura 1Scenario1 CMA-ES baseline andamento J

Notiamo sempre in figura l'esito dei trial eseguiti in modalità baseline. Tutti i test sono stati completati con successo, con 10 missioni riuscite su 10, senza fallimenti o condizioni di arresto anticipato.



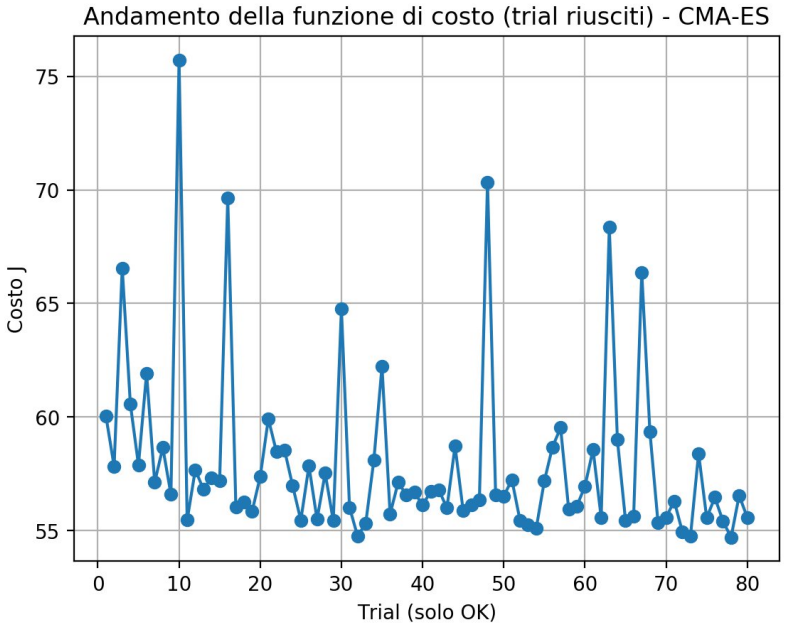
Scenario1 CMA-ES baseline andamento trial

Modalità Optimization

Durante la fase di **optimization**, l'algoritmo CMA-ES esplora lo spazio dei parametri del planner locale eseguendo **80 trial** nel simulatore. Per ogni trial viene calcolato il valore della funzione di costo J, che sintetizza le prestazioni di navigazione del robot.

L'andamento della funzione di costo mostrato nella figura seguente evidenzia una fase iniziale caratterizzata da maggiore variabilità, dovuta all'esplorazione dello spazio dei parametri. Con il progredire dei trial i valori tendono progressivamente a stabilizzarsi attorno a configurazioni con costo inferiore, indicando la

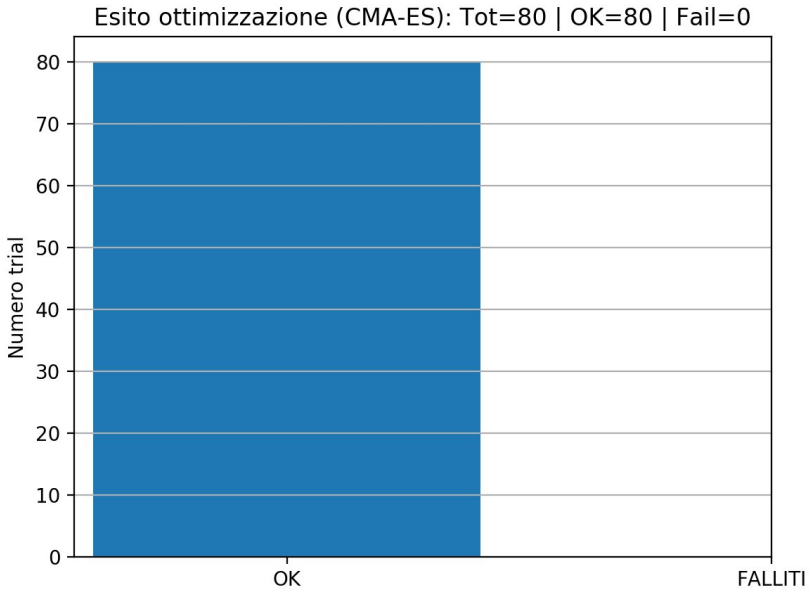
convergenza dell'algoritmo verso regioni più promettenti dello spazio delle soluzioni.



Scenario1 CMA-ES optimization andamento J

Rispetto al comportamento osservato con PSO, la distribuzione dei valori di costo appare complessivamente più regolare e con minori oscillazioni, suggerendo una dinamica di ricerca più stabile.

La Figura giù riporta di nuovo l'esito dei trial eseguiti durante la fase di ottimizzazione. Tutti i **80 trial sono stati completati con successo**, senza fallimenti o condizioni di arresto anticipato.



Scenario1 CMA-ES optimization andamento trial

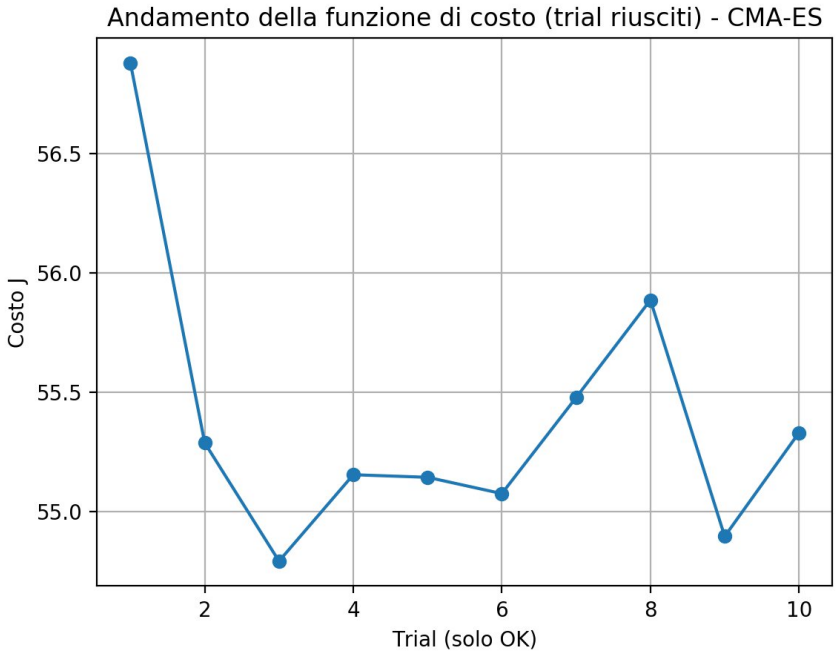
Questo risultato evidenzia una buona robustezza del processo di ottimizzazione, indicando che le configurazioni dei parametri generate dall'algoritmo rimangono all'interno di regioni dello spazio dei parametri compatibili con una navigazione stabile del robot.

Il miglior risultato ottenuto durante la fase di ottimizzazione corrisponde a un tempo di missione pari a circa 52.46 s, con un valore minimo della funzione di costo J pari a circa 54.70.

Modalità Best replay

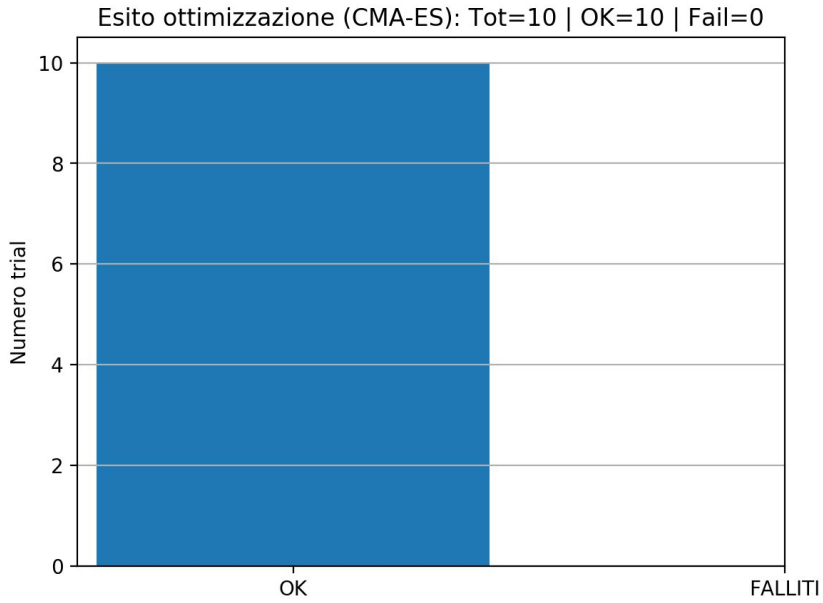
La modalità **best replay** viene utilizzata per verificare le prestazioni del robot utilizzando il miglior set di parametri individuato durante la fase di ottimizzazione con CMA-ES.

La Figura seguente mostra l'andamento della funzione di costo Jnei **10 trial di validazione** eseguiti con i parametri ottimizzati. I valori risultano generalmente compresi tra **circa 54.8 e 56.8**, con una distribuzione relativamente compatta che conferma la stabilità delle prestazioni ottenute dopo l'ottimizzazione.



Scenario1 CMA-ES best replay andamento J

La Figura X riporta l'esito dei trial di validazione. Tutti i **10 test sono stati completati con successo**, senza fallimenti o condizioni di arresto anticipato, confermando la **riproducibilità delle prestazioni** ottenute con i parametri ottimizzati.

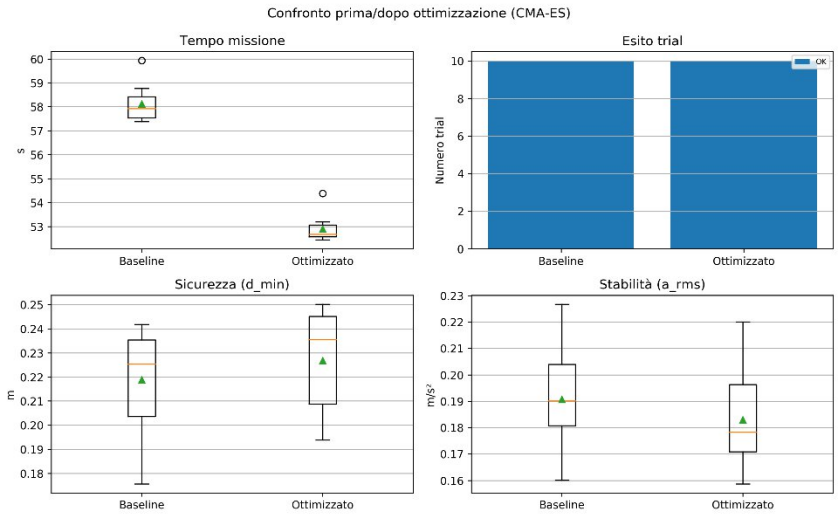


Scenario1 CMA-ES best replay andamento trial

La Figura proposta più in basso riporta il confronto tra la configurazione **baseline** e la configurazione **ottimizzata con CMA-ES**, valutate attraverso i trial di validazione eseguiti in modalità **best replay**. L'analisi evidenzia un miglioramento chiaro della prestazione temporale: il **tempo mediano di missione** si riduce infatti da circa **58.1 s** a circa **52.8 s**, con una variazione pari a circa **−9.1%**. Anche il **tempo medio di missione** mostra un andamento coerente, passando da circa **58.0 s** a circa **53.0 s**, a conferma del fatto che il miglioramento non dipende da singoli trial isolati, ma rappresenta un effetto sistematico della nuova configurazione dei parametri.

Dal punto di vista dell'affidabilità, il confronto tra baseline e configurazione ottimizzata mostra un **success rate invariato e pari al 100%**, con **10 missioni riuscite su 10** in entrambi i casi. A questo risultato si aggiunge un ulteriore elemento positivo osservato durante la fase di ottimizzazione: CMA-ES ha completato con successo **tutti gli 80 trial**, senza registrare fallimenti o condizioni

di arresto anticipato. Questo comportamento suggerisce che l'algoritmo esplora lo spazio dei parametri in modo robusto, evitando configurazioni eccessivamente instabili o incompatibili con il corretto svolgimento della navigazione.



Scenario1 CMA-ES Baseline vs Best replay

L'analisi della metrica di sicurezza mostra inoltre un lieve miglioramento rispetto alla configurazione baseline. La **distanza minima dagli ostacoli** d_{min} aumenta infatti da circa **0.225 m** a circa **0.235 m**, con una variazione positiva dell'ordine del **4%**. Ciò indica che il robot, oltre a muoversi più velocemente, mantiene anche un margine di sicurezza leggermente superiore rispetto agli ostacoli presenti nello scenario. Questo risultato è particolarmente rilevante, poiché dimostra che la riduzione del tempo di missione non è stata ottenuta sacrificando la sicurezza della traiettoria.

Anche la metrica di stabilità dinamica evidenzia un miglioramento. Il valore mediano di **accelerazione RMS** a_{rms} si riduce da circa **0.19 m/s²** a circa **0.18 m/s²**, con una variazione di circa **−5%**. Tale riduzione suggerisce un moto mediamente

più fluido, con variazioni di velocità meno brusche e una dinamica complessivamente più regolare lungo il percorso.

Nel complesso, i risultati mostrano che l'algoritmo **CMA-ES** è stato in grado di individuare una configurazione del planner locale che migliora in modo congiunto le principali metriche considerate nello Scenario 1. Rispetto alla baseline, il robot raggiunge il goal **più rapidamente**, mantiene **una distanza leggermente maggiore dagli ostacoli** e presenta **un moto più regolare**, senza penalizzazioni nel tasso di successo. A differenza di quanto osservato per PSO, l'ottimizzazione ottenuta con CMA-ES appare quindi **più equilibrata**, in quanto il miglioramento temporale non viene ottenuto a scapito della sicurezza o della stabilità.

Qui di seguito possiamo apprezzare delle tabelle riassuntive dei risultati ottenuti con CMA-ES.

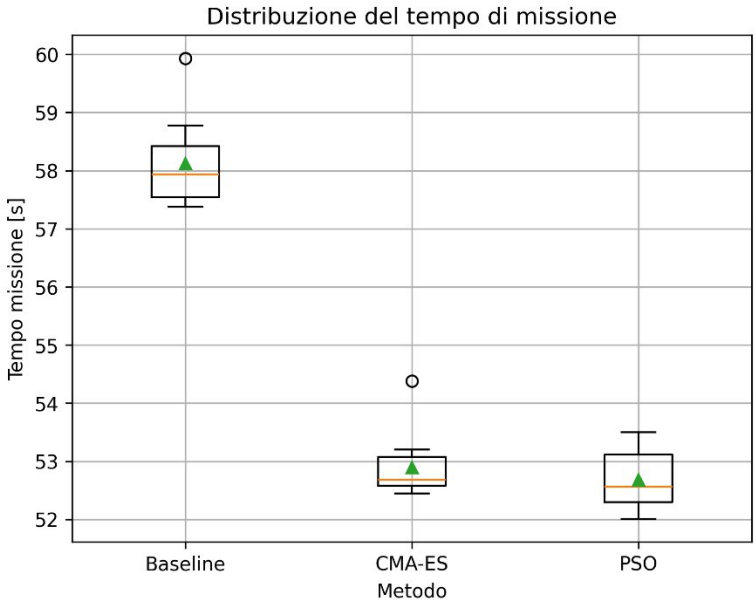
Metrica	Baseline	CMA-ES ottimizzato	Variazione	Commento
Tempo missione mediano (s)	58.1	52.8	-9.1%	Riduzione netta del tempo di navigazione
Tempo missione medio (s)	58.0	53.0	-8.6%	Miglioramento coerente anche sul valore medio
Success rate validazione	100% (10/10)	100% (10/10)	Invariato	Prestazioni riproducibili senza perdita di affidabilità
d_min mediano (m)	0.225	0.235	+4%	Leggero aumento del margine di sicurezza
a_rms mediano (m/s^2)	0.19	0.18	-5%	Moto più fluido e dinamicamente più regolare

Statistica fase di ottimizzazione CMA-ES	Valore
Numero trial	80
Successi	80
Fallimenti	0
Tasso di successo	100%
Miglior costo J	54.7
Costo medio finale	56.0
Range costi iniziali	56 - 76
Range costi finali	54 - 58

In sintesi, CMA-ES ha prodotto un miglioramento complessivo più bilanciato rispetto alla baseline, riducendo il tempo di missione e migliorando simultaneamente sicurezza e stabilità del moto.

6.1.3 Confronto tra i metodi di ottimizzazione

La Figura riportata di seguito mostra la distribuzione del tempo di missione ottenuto nei trial di validazione per le tre configurazioni considerate: baseline, CMA-ES e PSO.

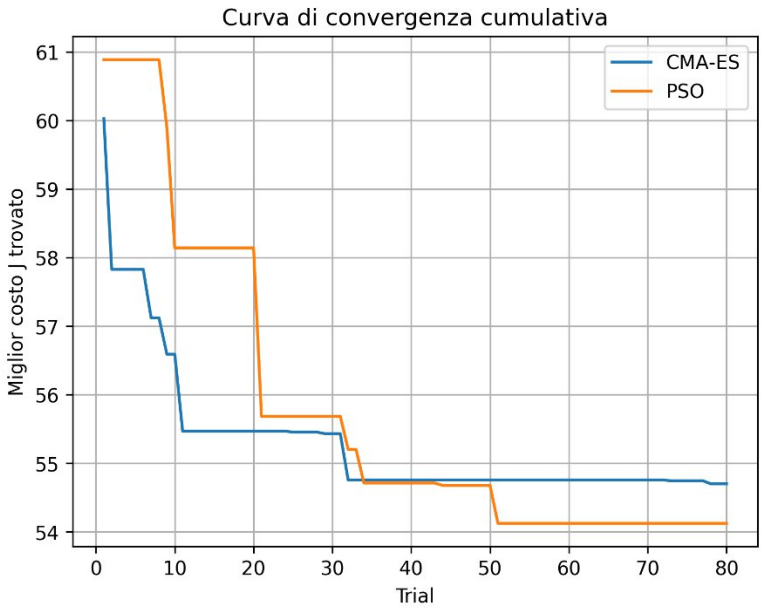


Distribuzione tempo missione scenario 1 PSO vs CMA-ES

In particolare, il tempo mediano di missione passa **da 57.93** s nella configurazione baseline a **52.68** s con CMA-ES e **52.56** s con PSO. La riduzione del tempo di navigazione è quindi pari rispettivamente a circa **−9.1%** e **−9.3%,** indicando che entrambi i metodi sono in grado di individuare configurazioni dei parametri del planner locale che migliorano sensibilmente l'efficienza della navigazione.

La distribuzione dei tempi mostra inoltre una minore variabilità nelle configurazioni ottimizzate rispetto alla baseline, suggerendo che i parametri ottimizzati producono un comportamento più consistente del robot tra i diversi trial.
Un ulteriore elemento di confronto riguarda la dinamica di convergenza degli algoritmi durante la fase di ottimizzazione.

La Figura seguente mostra l'andamento cumulativo del miglior valore della funzione di costo J individuato nel corso degli 80 trial di ottimizzazione per entrambi i metodi.

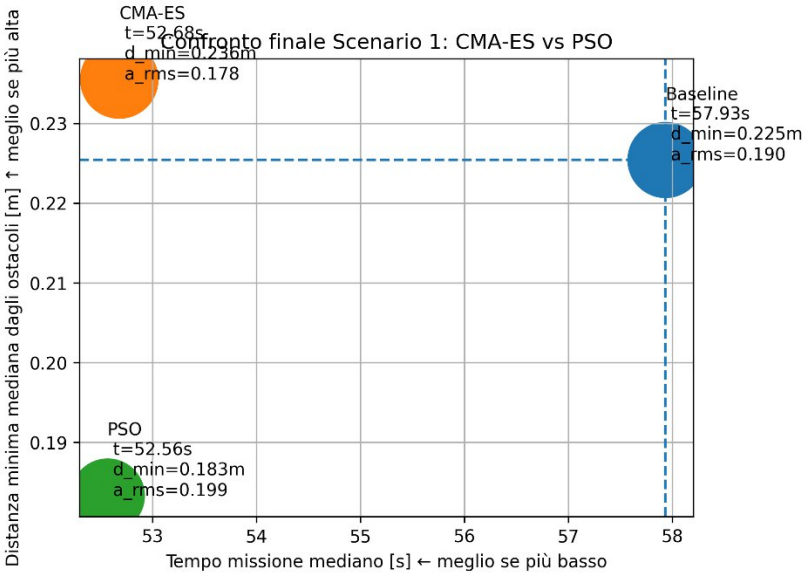


L'algoritmo CMA-ES mostra una riduzione rapida del costo nelle prime iterazioni, indicando una buona capacità di individuare rapidamente regioni promettenti dello spazio dei parametri. Successivamente, l'algoritmo converge gradualmente verso una soluzione stabile con un valore di costo pari a circa **54.7.**

L'algoritmo PSO, invece, mostra una fase iniziale di convergenza leggermente più lenta, ma riesce progressivamente a raggiungere valori di costo comparabili, ottenendo un risultato finale leggermente inferiore (circa **54.1**). Questo comportamento è coerente con la natura degli algoritmi: PSO tende a esplorare lo spazio delle soluzioni attraverso il movimento cooperativo delle particelle, mentre CMA-ES utilizza l'adattamento della matrice di covarianza per modellare la distribuzione delle soluzioni promettenti.

Per analizzare simultaneamente le principali metriche di prestazione, la figura ancora di seguito riporta un confronto multidimensionale tra **baseline**, **CMA-ES** e **PSO**, considerando:

- tempo mediano di missione
- distanza minima dagli ostacoli d_{min}
- accelerazione RMS a_{rms}



Confronto multidimensionale scenario 1 PSO vs CMA-ES

Nel grafico il tempo di missione è rappresentato sull'asse orizzontale (minore è migliore), mentre la distanza dagli ostacoli è rappresentata sull'asse verticale (maggiore è migliore). La dimensione delle bolle rappresenta invece il livello di accelerazione RMS.

L'analisi evidenzia che entrambi i metodi migliorano significativamente il tempo di navigazione rispetto alla baseline. Tuttavia, emergono alcune differenze qualitative tra i due approcci:

CMA-ES

- mantiene una distanza maggiore dagli ostacoli
- presenta accelerazioni RMS leggermente inferiori
- mostra convergenza più stabile

PSO

- ottiene il tempo di missione leggermente più basso
- tende a produrre una navigazione più aggressiva, con minore distanza dagli ostacoli e accelerazioni leggermente più elevate.

In altre parole, PSO privilegia la velocità, mentre CMA-ES produce una soluzione più equilibrata tra velocità, sicurezza e stabilità del moto.

Sintesi tecnica del confronto

Dall'analisi complessiva emerge che entrambi gli algoritmi di ottimizzazione migliorano significativamente le prestazioni di navigazione rispetto alla configurazione baseline.

Il metodo **PSO** ottiene il **tempo di missione leggermente inferiore**, risultando quindi il metodo più veloce nello scenario considerato. Tuttavia, questo miglioramento è accompagnato da una **riduzione della distanza minima dagli ostacoli** e da **accelerazioni RMS leggermente superiori**, indicando una dinamica di movimento più aggressiva.

Il metodo **CMA-ES**, invece, produce una soluzione più bilanciata, caratterizzata da **tempo di missione comparabile**, **maggiore distanza dagli ostacoli** e **movimento più fluido**, oltre a una **maggiore stabilità durante la fase di ottimizzazione**.

Nel complesso, i risultati suggeriscono che **CMA-ES rappresenta l'approccio più equilibrato**, mentre **PSO può risultare preferibile quando l'obiettivo principale è la minimizzazione del tempo di missione**.

Metodo	Tempo mediano missione (s)	Miglioramento vs baseline	d_min mediano (m)	a_rms mediano (m/s²)	Success rate ottimizzazione	Success rate replay
Baseline	57.93	–	0.225	0.190	100%	100%
CMA-ES	52.68	−9.1 %	0.235	0.180	100%	100%
PSO	52.56	−9.3 %	0.185	0.200	98.75 %	100%

6.2 Scenario 2

Per approfondire la valutazione delle prestazioni del sistema di navigazione e degli algoritmi di ottimizzazione, è stato definito un secondo scenario di testing caratterizzato da condizioni operative più critiche rispetto allo scenario precedente. L'obiettivo principale di questo scenario è stato quello di analizzare il comportamento del robot in presenza di vincoli geometrici più stringenti e di ostacoli non previsti nella mappa globale, al fine di verificare la robustezza dell'architettura di navigazione e l'efficacia degli algoritmi di ottimizzazione nel migliorare le prestazioni del sistema.

In particolare, nello Scenario 2 è stato progettato un percorso che richiede al robot di attraversare corridoi significativamente più stretti rispetto a quelli utilizzati nello Scenario 1. Questo tipo di configurazione rappresenta una condizione

particolarmente sfidante per il sistema di navigazione basato su *move_base*, poiché piccoli errori di localizzazione o variazioni nella stima della posa del robot possono portare a comportamenti subottimali. Tali errori possono derivare, ad esempio, da imprecisioni nell'odometria, da rumore nei sensori oppure da discrepanze tra il modello simulato e la dinamica effettiva del robot. In presenza di corridoi molto stretti, anche deviazioni di pochi centimetri rispetto alla traiettoria ideale possono causare una riduzione significativa della distanza dagli ostacoli laterali, generando situazioni di stallo, oscillazioni o attivazioni frequenti delle procedure di recovery interne al planner.

Oltre alla presenza di passaggi stretti, nello scenario è stato introdotto un ulteriore elemento di complessità: un ostacolo imprevisto non presente nella mappa globale statica dell'ambiente. In particolare, nella simulazione è stata inserita una figura rappresentante un essere umano collocata lungo il percorso del robot. Poiché questo oggetto non è incluso nella mappa globale utilizzata dal planner globale, esso rappresenta un ostacolo dinamico o comunque non noto a priori dal sistema di pianificazione. In queste condizioni, il planner globale continua a considerare il percorso come libero, mentre il planner locale deve reagire esclusivamente sulla base delle informazioni provenienti dai sensori, in particolare dal sensore lidar simulato.

La combinazione di corridoi stretti e ostacoli non presenti nella mappa globale rappresenta quindi un caso d'uso realistico in cui il sistema di navigazione deve essere in grado di adattarsi a condizioni non perfettamente modellate nella rappresentazione dell'ambiente. Per gestire tali criticità, nello Scenario 2 è stato attivato l'algoritmo di recovery basato su Artificial Potential Field (APF) descritto nel Capitolo 5. Questo algoritmo viene richiamato quando il sistema di monitoraggio rileva condizioni critiche durante la navigazione, come prossimità eccessiva agli ostacoli, stallo del robot o assenza di progresso verso il goal. In tali situazioni, il planner corrente viene temporaneamente interrotto e il robot passa a una modalità reattiva basata sul calcolo di forze attrattive verso il goal e repulsive

rispetto agli ostacoli rilevati dal sensore lidar. Lo scopo di questa procedura è consentire al robot di superare la configurazione critica e di ristabilire condizioni di navigazione favorevoli, dopodiché il controllo viene nuovamente restituito al sistema *move_base*.

Questo scenario è stato quindi progettato per valutare le prestazioni degli algoritmi di ottimizzazione in presenza sia di ostacoli noti (presenti nella mappa globale) sia di ostacoli non noti, gestiti dinamicamente dal planner locale e dall'algoritmo APF. In questo modo è stato possibile analizzare la capacità dei parametri ottimizzati di garantire una navigazione efficiente e robusta anche in condizioni ambientali più complesse e meno prevedibili.

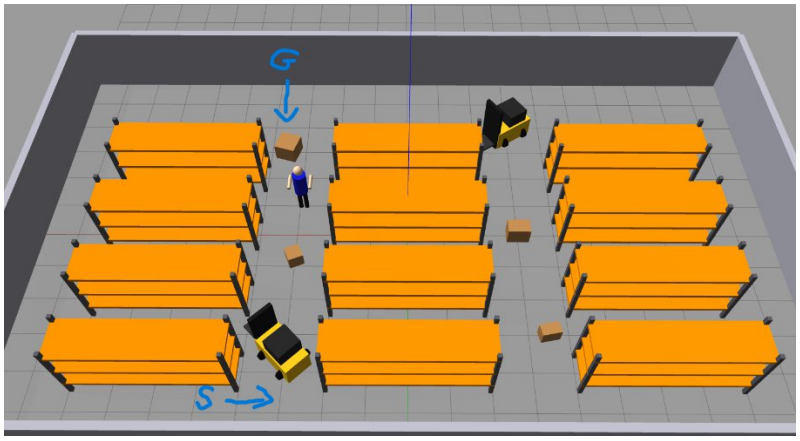
Analogamente allo Scenario 1, anche in questo caso ciascun algoritmo di ottimizzazione è stato eseguito secondo tre modalità operative distinte:

- Baseline, in cui il robot utilizza i parametri di navigazione standard senza alcuna ottimizzazione;
- Optimization, in cui l'algoritmo (PSO o CMA-ES) esplora lo spazio dei parametri del planner locale al fine di minimizzare la funzione di costo definita nel Capitolo 5;
- Best replay, in cui la configurazione di parametri risultata ottimale al termine del processo di ottimizzazione viene testata nuovamente per verificarne la robustezza e la ripetibilità.

Il processo di ottimizzazione è stato eseguito separatamente per i due algoritmi considerati, mantenendo invariato l'ambiente di simulazione e la configurazione sperimentale. Per ciascun algoritmo, l'intero processo – comprendente le fasi di baseline, ottimizzazione e best replay – ha richiesto circa 90 minuti di esecuzione,

includendo i tempi necessari per l'esecuzione dei trial, la raccolta dei dati e la valutazione della funzione di costo.

I risultati ottenuti nello Scenario 2 consentono quindi di confrontare il comportamento dei due algoritmi di ottimizzazione in un contesto più realistico e impegnativo rispetto allo scenario precedente, evidenziando le differenze in termini di robustezza, efficienza e capacità di gestire situazioni di navigazione complesse mediante l'integrazione con l'algoritmo di recovery basato su Artificial Potential Field.



Percorso Scenario 2

I parametri ottimizzati e i relativi bounds, analogamente allo scenario 1, sono i seguenti:

- **max_vel_x**

$$-0.10 \leq max\_vel\_x \leq 0.22 \text{m/s}$$

  Il limite superiore corrisponde alla velocità massima del TurtleBot3 Burger, mentre il limite inferiore consente brevi manovre di retrocessione utili nelle situazioni di disimpegno.

- **acc_lim_x**

$$0.30 \leq acc\_lim\_x \leq 1.20 \text{m/s}^2$$

Il parametro controlla la reattività dinamica del robot. L'intervallo consente di esplorare comportamenti compresi tra guida conservativa e guida più aggressiva.

- **occdist_scale**

$$0.01 \leq occdist\_scale \leq 0.12$$

Determina il peso della distanza dagli ostacoli nella funzione di costo del planner, influenzando il compromesso tra sicurezza e velocità.

- **inflation_radius**

$$0.15 \leq inflation\_radius \leq 0.35 \text{m}$$

Definisce la distanza di sicurezza con cui gli ostacoli vengono inflazionati nella costmap.

- **path_distance_bias**

$$5.0 \leq path\_distance\_bias \leq 60.0$$

Controlla quanto il robot tende a seguire il percorso globale pianificato.

- **goal_distance_bias**

$$5.0 \leq goal\_distance\_bias \leq 40.0$$

Regola quanto il planner privilegia traiettorie che riducono la distanza dal goal.

6.2.1 Ottimizzazione con PSO

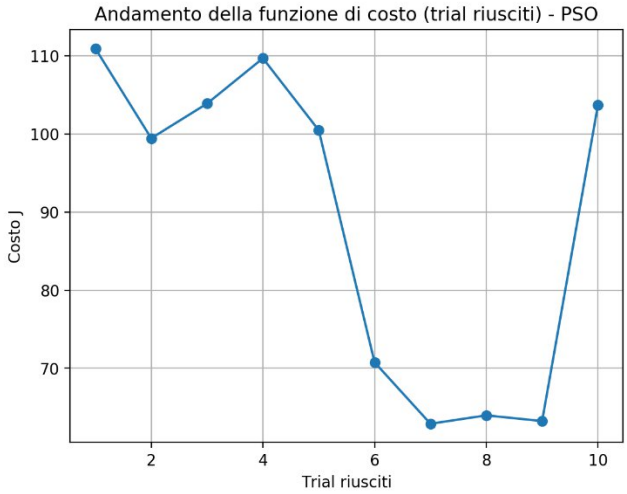
Per valutare l'efficacia dell'algoritmo di Particle Swarm Optimization (PSO) nello scenario caratterizzato da corridoi stretti e dalla presenza di un ostacolo imprevisto, è stato eseguito il processo di ottimizzazione dei parametri del planner locale secondo la stessa procedura descritta nel capitolo precedente. L'obiettivo dell'ottimizzazione è stato quello di individuare una configurazione di parametri in grado di migliorare le prestazioni di navigazione del robot in termini di tempo di missione, stabilità del movimento e distanza minima dagli ostacoli, anche in presenza delle criticità introdotte nello Scenario 2.

Analogamente alla metodologia già adottata nello Scenario 1, l'algoritmo PSO è stato eseguito secondo tre modalità operative distinte: baseline, optimization e best replay. Nella modalità baseline il robot utilizza i parametri standard del planner locale senza alcun intervento di ottimizzazione, consentendo di stabilire un riferimento prestazionale iniziale. Nella modalità optimization, invece, il PSO esplora lo spazio dei parametri del planner al fine di minimizzare la funzione di costo definita nel Capitolo 5. Infine, nella modalità best replay viene testata la configurazione di parametri risultata ottimale al termine del processo di ottimizzazione, al fine di verificarne la stabilità e la ripetibilità delle prestazioni.

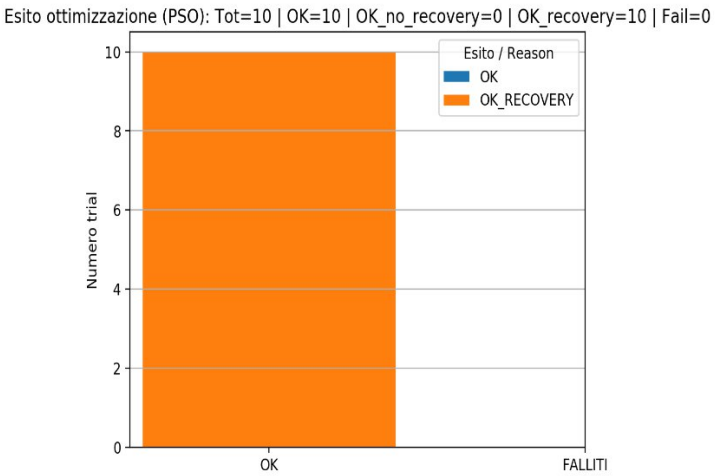
Nei paragrafi successivi vengono analizzati i risultati ottenuti nelle tre modalità operative, accompagnando i grafici generati durante gli esperimenti con un'interpretazione delle principali tendenze osservate nei dati sperimentali.

Modalità baseline

Di seguito i grafici relativi all'andamento dei trial e della funzione di costo J per la modalità baseline:



Scenario 2 PSO baseline andamento J



Scenario 2 PSO baseline andamento trial

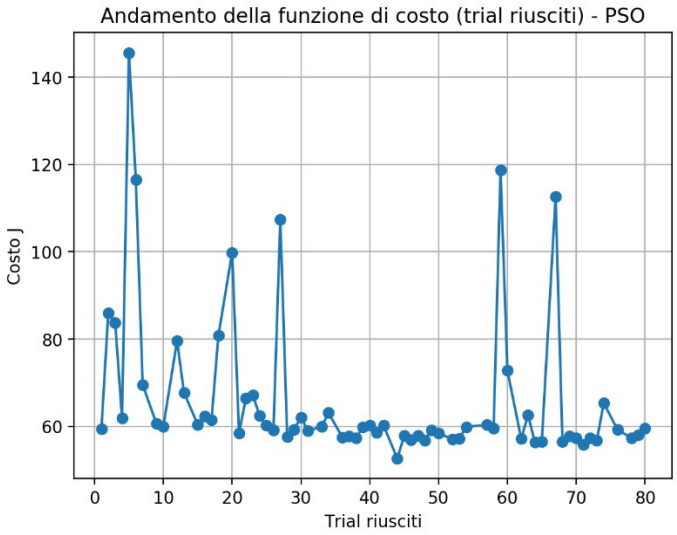
Dal grafico relativo all'**andamento della funzione di costo** J nei trial riusciti, si osserva che i valori della funzione di costo risultano mediamente elevati e presentano una certa variabilità tra le diverse esecuzioni. In particolare, nelle prime esecuzioni si registrano valori superiori a 100, indicativi di prestazioni di navigazione non ottimali, dovute principalmente ai tempi di missione più elevati e alle difficoltà di attraversamento dei corridoi stretti introdotti nello Scenario 2.

Il grafico relativo agli **esiti dei trial** evidenzia che tutte le esecuzioni sono state completate con successo (**10 trial su 10**), tuttavia in tutti i casi è stato necessario l'intervento della procedura di **recovery basata su Artificial Potential Field**, attivata per gestire l'ostacolo imprevisto presente lungo il percorso. Questo comportamento conferma come, nello Scenario 2, il planner locale con i parametri di default non sia in grado di gestire autonomamente tutte le situazioni critiche, rendendo necessario l'utilizzo del meccanismo di recovery per consentire al robot di completare la missione.

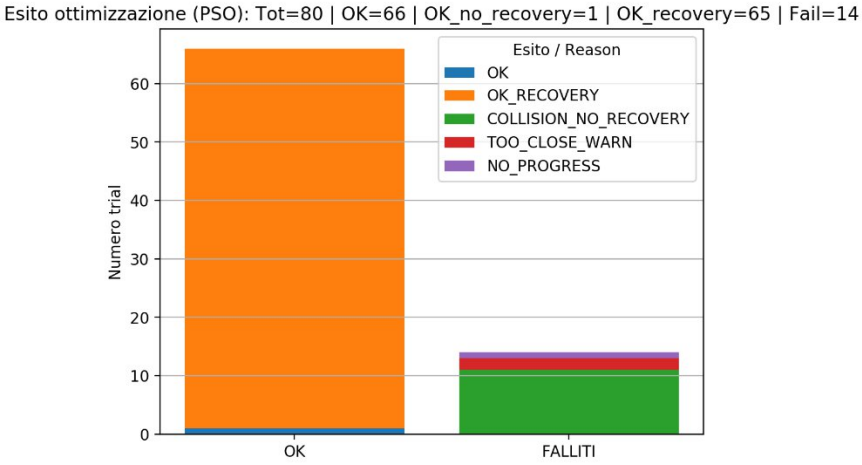
Nel complesso, i risultati della baseline mostrano che il sistema di navigazione è in grado di portare a termine il percorso anche in presenza delle criticità introdotte nello scenario, ma con prestazioni non ottimali e con una forte dipendenza dalle procedure di recupero. Questa configurazione rappresenta quindi il punto di riferimento rispetto al quale verranno confrontati i risultati ottenuti mediante l'ottimizzazione con PSO.

Modalità Optimization

Nella modalità optimization, l'algoritmo Particle Swarm Optimization esplora lo spazio dei parametri del planner locale con l'obiettivo di minimizzare la funzione di costo J. Durante questa fase sono stati eseguiti 80 trial, nei quali l'algoritmo ha progressivamente aggiornato le configurazioni dei parametri sulla base delle prestazioni osservate. Di seguito i grafici dell'andamento dei trial e della funzione di costo:



Scenario 2 PSO optimization andamento J



Scenario 2 PSO optimization andamento trial

Dal grafico dell'**andamento della funzione di costo** si osserva come, nelle prime iterazioni, i valori di Jrisultino relativamente elevati e caratterizzati da una forte variabilità. Questo comportamento è tipico delle fasi iniziali dell'algoritmo PSO, in cui le particelle esplorano ampie regioni dello spazio dei parametri alla ricerca di configurazioni promettenti.

Con il progredire dei trial, i valori della funzione di costo tendono progressivamente a stabilizzarsi su valori più bassi, generalmente compresi

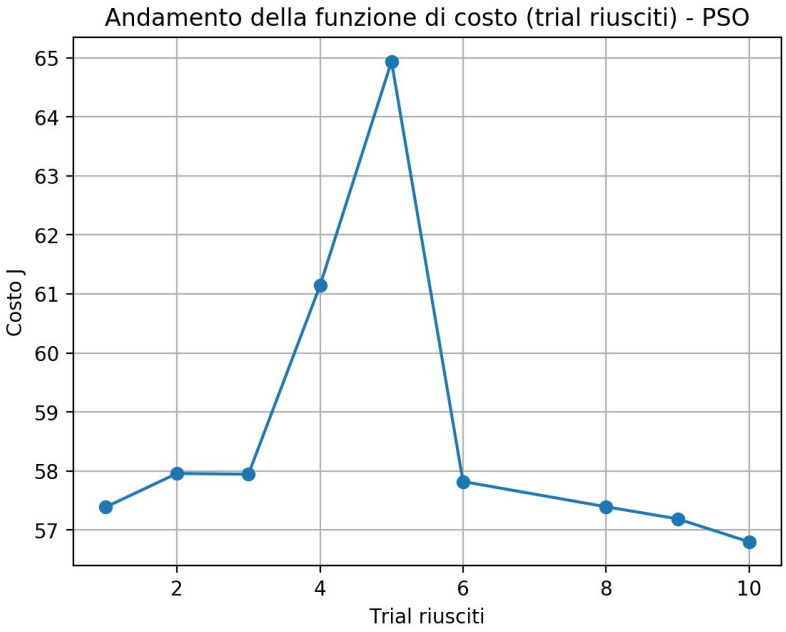
nell'intervallo **55–60**, indicando che l'algoritmo ha individuato configurazioni di parametri più efficienti per la navigazione nello scenario considerato.

Il grafico relativo agli **esiti dei trial** mostra che, su un totale di **80 esecuzioni**, **66 missioni sono state completate con successo**, mentre **14 trial sono falliti**. Tra i successi, **65 missioni hanno richiesto l'intervento della procedura di recovery basata su APF**, mentre solo **un trial è stato completato senza l'attivazione del recovery**. Questo risultato evidenzia come, nello Scenario 2, la presenza dell'ostacolo imprevisto renda frequentemente necessario l'intervento del meccanismo di recupero, anche quando i parametri del planner vengono ottimizzati.

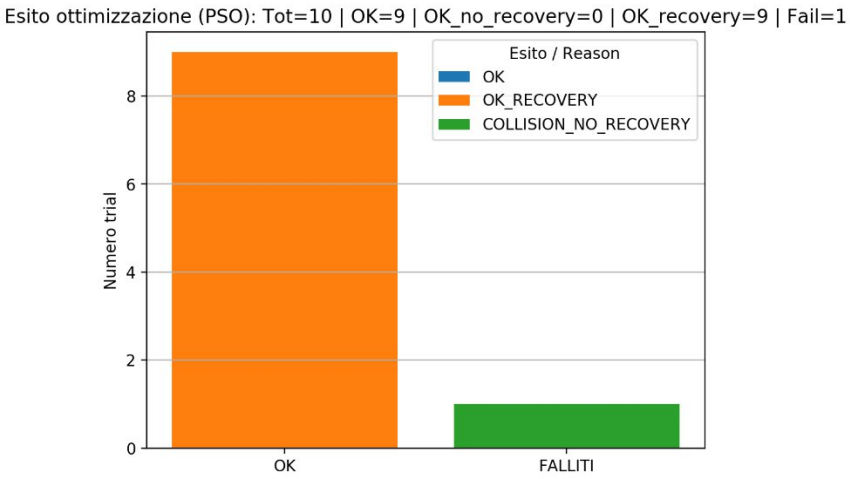
Al termine del processo di ottimizzazione, la migliore configurazione individuata dall'algoritmo PSO ha prodotto un **tempo di missione pari a circa 49.47 s** e un valore minimo della funzione di costo $J = 52.65$, rappresentando un miglioramento significativo rispetto alle prestazioni osservate nella configurazione baseline.

Modalità Best replay

Nella modalità **best replay**, di nuovo la configurazione di parametri risultata ottimale al termine della fase di ottimizzazione viene rieseguita più volte al fine di verificarne la stabilità e la ripetibilità delle prestazioni.



Scenario 2 PSO best replay andamento J



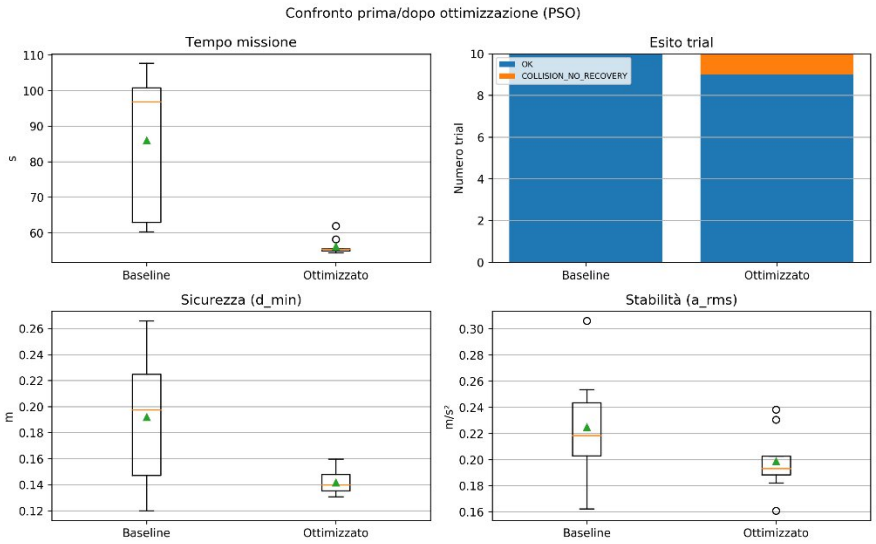
Scenario 2 PSO best replay andamento trial

Dal grafico dell'**andamento della funzione di costo** J si osserva che i valori rimangono relativamente stabili durante le diverse esecuzioni, con valori generalmente compresi tra **57 e 58**, ad eccezione di un singolo picco intorno a **65**, dovuto a una missione con prestazioni leggermente peggiori. Questo comportamento indica che la configurazione di parametri individuata

dall'algoritmo PSO produce risultati complessivamente consistenti tra diverse esecuzioni.

Il grafico relativo agli **esiti dei trial** mostra che **9 missioni su 10 sono state completate con successo**, mentre **un trial è fallito** a causa di collisione senza recupero. In tutti i casi di successo è stato necessario l'intervento della procedura di **recovery basata su APF**, evidenziando come, anche con i parametri ottimizzati, la presenza dell'ostacolo imprevisto nello Scenario 2 renda spesso indispensabile l'attivazione del meccanismo di recupero.

Nel complesso, i risultati del best replay confermano che i parametri individuati tramite PSO consentono di ottenere prestazioni di navigazione più efficienti rispetto alla baseline, mantenendo al contempo una buona stabilità tra diverse esecuzioni della missione.



Scenario 2 PSO baseline vs best replay

Il confronto tra la configurazione **baseline** e la configurazione **ottimizzata** ottenuta tramite PSO evidenzia un miglioramento significativo delle prestazioni di navigazione nello Scenario 2.

Dal boxplot relativo al **tempo di missione** si osserva una riduzione marcata dei tempi necessari per completare il percorso. In particolare, il tempo mediano passa da circa **96.8 s nella configurazione baseline** a circa **55.3 s nella configurazione ottimizzata**, indicando un miglioramento sostanziale dell'efficienza della navigazione.

Per quanto riguarda la **sicurezza**, espressa tramite la distanza minima dagli ostacoli d_{min}, la configurazione ottimizzata mostra valori medi leggermente inferiori rispetto alla baseline. Questo comportamento è coerente con l'obiettivo dell'ottimizzazione, che privilegia una navigazione più rapida anche a costo di una minore distanza dagli ostacoli, pur rimanendo entro margini di sicurezza accettabili.

Dal punto di vista della **stabilità della traiettoria**, valutata tramite il parametro a_{rms}, si osserva una riduzione della variabilità nella configurazione ottimizzata, indicando un comportamento di navigazione più regolare e meno soggetto a oscillazioni.

Infine, l'analisi degli **esiti dei trial** mostra che la configurazione baseline completa tutte le missioni con successo, mentre la configurazione ottimizzata registra **9 missioni riuscite su 10**, con un singolo fallimento. Questo risultato evidenzia come l'ottimizzazione migliori significativamente l'efficienza della navigazione, mantenendo comunque un livello di robustezza complessivamente elevato.

Nel complesso, i risultati ottenuti mostrano che l'algoritmo **PSO è in grado di individuare configurazioni di parametri che riducono sensibilmente il tempo di missione e migliorano la stabilità del movimento**, rendendo il robot più efficiente nell'attraversamento dei corridoi stretti e nella gestione dell'ostacolo imprevisto presente nello Scenario 2, mantenendo sempre un approccio aggressivo ma rimanendo entro margini di sicurezza accettabili.

Metrica	Baseline	Ottimizzato (PSO)	Miglioramento
Tempo missione mediano	96.80 s	55.29 s	−42.9 %
Tempo missione minimo	~60 s	49.47 s	−17 %
Success rate	100 %	90 %	−10 %
Distanza minima ostacoli (d_{min})	~0.19 m	~0.14 m	traiettoria più aggressiva
Iterazioni ottimizzazione	–	80	–
Trial riusciti	–	66	–
Trial falliti	–	14	–
OK con recovery	–	65	–
OK senza recovery	–	1	–
Convergenza algoritmo	–	~30–40 iterazioni	–

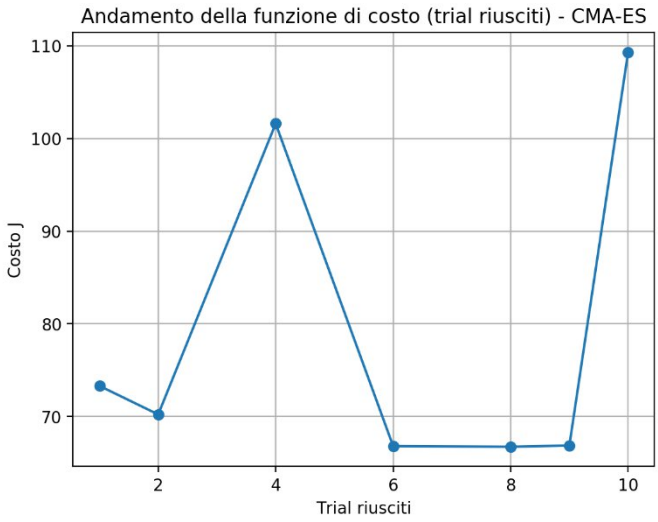
6.2.2 Ottimizzazione con CMA-ES

Analogamente a quanto fatto per PSO, anche per CMA-ES il processo sperimentale è stato suddiviso nelle tre modalità operative **baseline**, **optimization** e **best replay**, al fine di valutare rispettivamente le prestazioni del sistema con i parametri standard, il comportamento durante la fase di ottimizzazione e la stabilità della configurazione finale individuata dall'algoritmo.

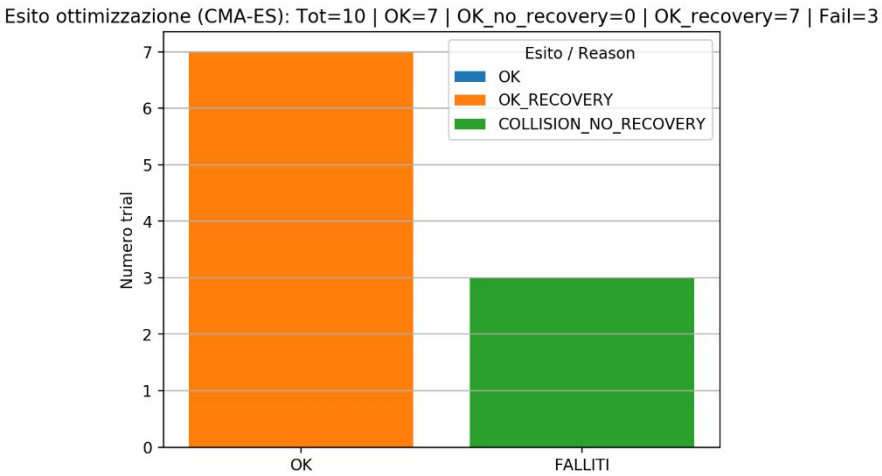
Nei paragrafi seguenti vengono analizzati i risultati ottenuti nello scenario 2, caratterizzato da corridoi stretti e dalla presenza di un ostacolo imprevisto, condizione in cui il sistema di navigazione utilizza il meccanismo di recovery basato su **Artificial Potential Field (APF)** per superare le configurazioni critiche.

Modalità baseline

Nella modalità **baseline**, il robot utilizza i parametri standard del planner locale senza alcuna ottimizzazione. Questa fase ha lo scopo di fornire un riferimento prestazionale per valutare i miglioramenti ottenuti successivamente tramite l'algoritmo CMA-ES. Di seguito i consueti grafici:



Scenario 2 CMA-ES baseline andamento J



Scenario 2 CMA-ES baseline andamento trial

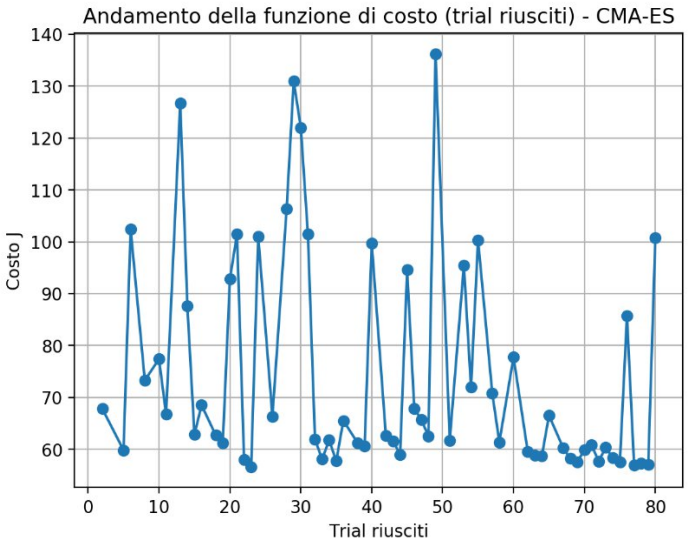
Dal grafico dell'**andamento della funzione di costo** J nei trial riusciti si osserva una certa variabilità tra le diverse esecuzioni, con valori generalmente compresi tra **70 e 110**. Questo comportamento evidenzia le difficoltà del robot nel navigare nello Scenario 2 con i parametri di default, in particolare a causa dei corridoi stretti e della presenza dell'ostacolo imprevisto lungo il percorso.

Il grafico relativo agli **esiti dei trial** mostra che **7 missioni su 10 sono state completate con successo**, mentre **3 trial sono falliti** a causa di collisioni senza recupero. In tutti i casi di successo è stato necessario l'intervento della procedura di **recovery basata su APF**, confermando che nello Scenario 2 il planner locale con i parametri standard non è in grado di gestire autonomamente tutte le criticità dell'ambiente.

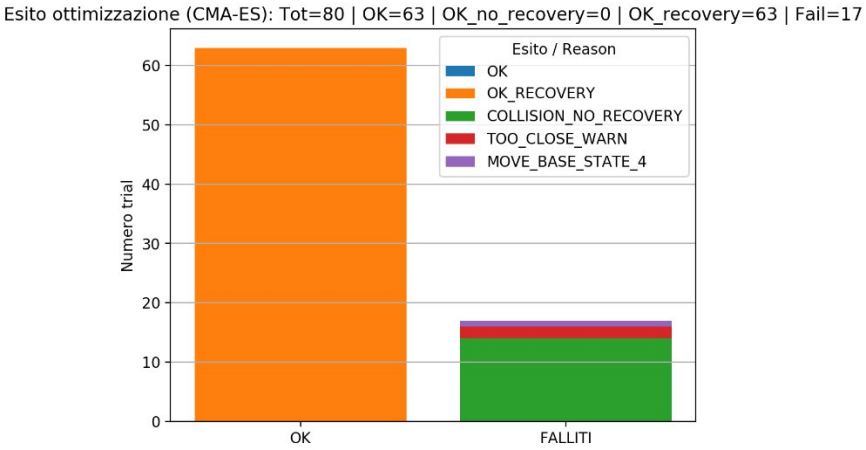
Nel complesso, la configurazione baseline evidenzia prestazioni meno stabili rispetto allo scenario precedente, rappresentando quindi un punto di riferimento utile per valutare i miglioramenti ottenuti tramite l'ottimizzazione con CMA-ES.

Modalità optimization

Nella fase di optimization l'algoritmo CMA-ES ha eseguito 80 iterazioni di ottimizzazione al fine di individuare una configurazione dei parametri del planner locale in grado di migliorare le prestazioni di navigazione nello Scenario 2. Di seguito i grafici :



Scenario 2 CMA-ES optimization andamento J



Scenario 2 CMA-ES optimization andamento trial

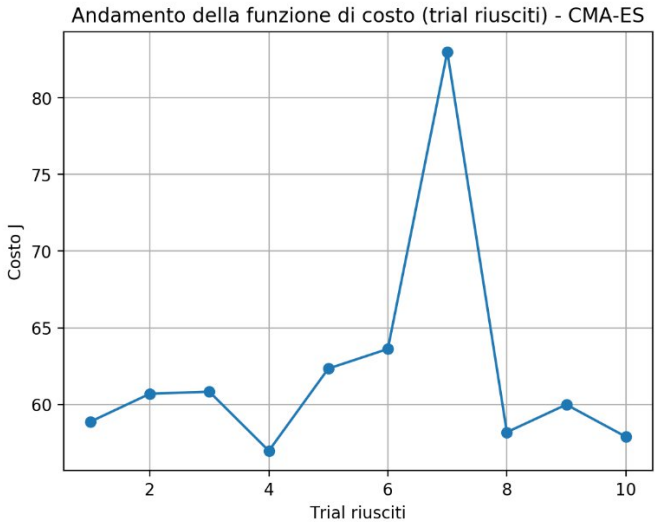
Dal grafico dell'**andamento della funzione di costo** J si osserva un comportamento inizialmente molto variabile, con diversi picchi superiori a **100**, dovuti principalmente a traiettorie inefficienti o a situazioni critiche generate dall'ostacolo imprevisto. Con il progredire delle iterazioni, tuttavia, i valori della funzione di costo tendono progressivamente a stabilizzarsi in un intervallo compreso tra **57 e 65**, indicando una graduale convergenza dell'algoritmo verso soluzioni più efficienti.

Per quanto riguarda gli **esiti dei trial**, su **80 esecuzioni totali** si registrano **63 missioni completate con successo** e **17 fallimenti**. Tutti i successi sono avvenuti con l'attivazione del meccanismo di **recovery basato su APF**, segno che nello Scenario 2 la presenza dell'ostacolo imprevisto richiede frequentemente l'intervento della procedura di recupero.

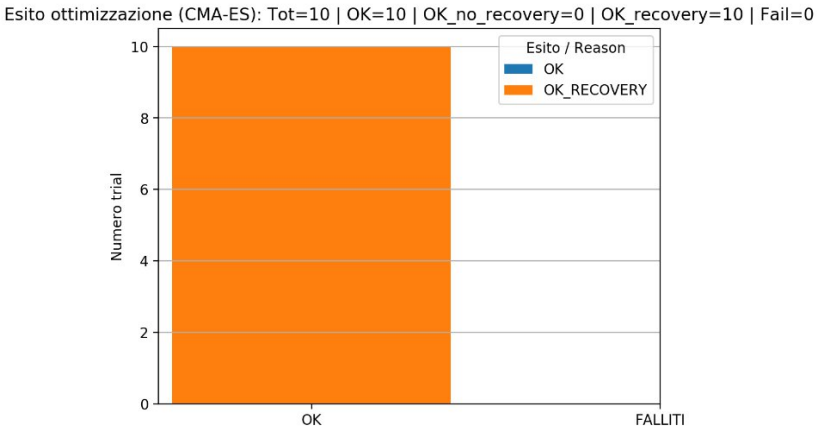
Il miglior risultato individuato dall'algoritmo corrisponde a un **tempo di missione pari a 54.04 s**, con un valore della funzione di costo $J = 56.62$. I parametri associati a questa configurazione rappresentano la soluzione ottimizzata che verrà successivamente validata nella modalità **best replay**.

Modalità best replay

Nella modalità best replay viene testata la configurazione di parametri individuata come migliore durante la fase di ottimizzazione, al fine di verificarne la stabilità e la riproducibilità delle prestazioni nello Scenario 2.



Scenario 2 CMA-ES best replay andamento J



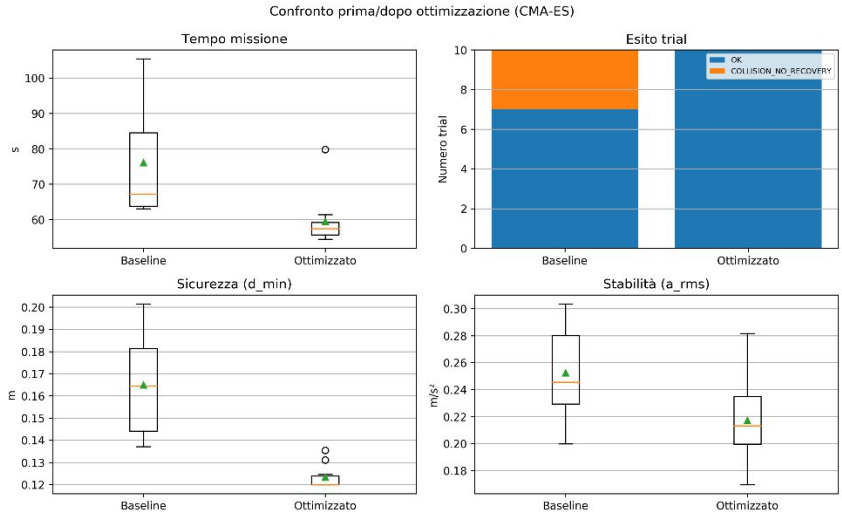
Scenario 2 CMA-ES best replay andamento trial

Dal grafico dell'**andamento della funzione di costo** J si osserva che i valori rimangono generalmente compresi tra **57 e 63**, con un solo picco isolato attorno a **83**. Questo comportamento indica una buona stabilità della soluzione individuata da CMA-ES, con prestazioni complessivamente costanti nelle diverse esecuzioni.

Il grafico relativo agli **esiti dei trial** mostra che **tutte le 10 missioni sono state completate con successo**, senza alcun fallimento. Analogamente alle fasi precedenti, tutti i successi sono stati ottenuti con l'attivazione della procedura di **recovery basata su APF**, confermando il ruolo fondamentale di questo meccanismo nella gestione dell'ostacolo imprevisto presente nello Scenario 2.

Nel complesso, la fase di best replay conferma che la configurazione di parametri individuata da CMA-ES consente al robot di mantenere prestazioni stabili e affidabili anche in presenza delle criticità introdotte nello scenario sperimentale.

Il confronto tra le configurazioni **baseline** e **best replay** permette di valutare l'effettivo miglioramento introdotto dall'ottimizzazione CMA-ES nello Scenario 2.



Scenario 2 CMA-ES baseline vs best replay

Dal grafico relativo al **tempo di missione** emerge una riduzione significativa delle prestazioni temporali. In configurazione baseline il tempo mediano di completamento della missione è pari a **67.22 s**, mentre nella configurazione ottimizzata scende a **57.44 s**, con una riduzione complessiva di circa **14.5%**. Inoltre la distribuzione dei tempi appare più compatta nella configurazione ottimizzata, indicando una maggiore **consistenza delle prestazioni** tra i diversi trial.

Dal punto di vista dell'**affidabilità**, il sistema mostra un miglioramento ancora più evidente. Nella configurazione baseline il **success rate** è pari al **70%**, con **3 missioni fallite su 10**, principalmente dovute a collisioni senza recupero durante l'attraversamento dei corridoi stretti o nelle fasi di interazione con l'ostacolo imprevisto. Dopo l'ottimizzazione, invece, il sistema raggiunge un **success rate del 100%**, completando con successo tutte le missioni nel test di best replay.

Analizzando la metrica di **sicurezza**, rappresentata dalla distanza minima dagli ostacoli d_{min}, si osserva una leggera riduzione nella configurazione ottimizzata. Questo comportamento indica che il robot tende ad adottare traiettorie **più aggressive e dirette**, riducendo la distanza dagli ostacoli per ottenere tempi di percorrenza inferiori. Tale comportamento è coerente con l'obiettivo della

funzione di costo utilizzata durante l'ottimizzazione, che privilegia il tempo di missione pur mantenendo un margine minimo di sicurezza.

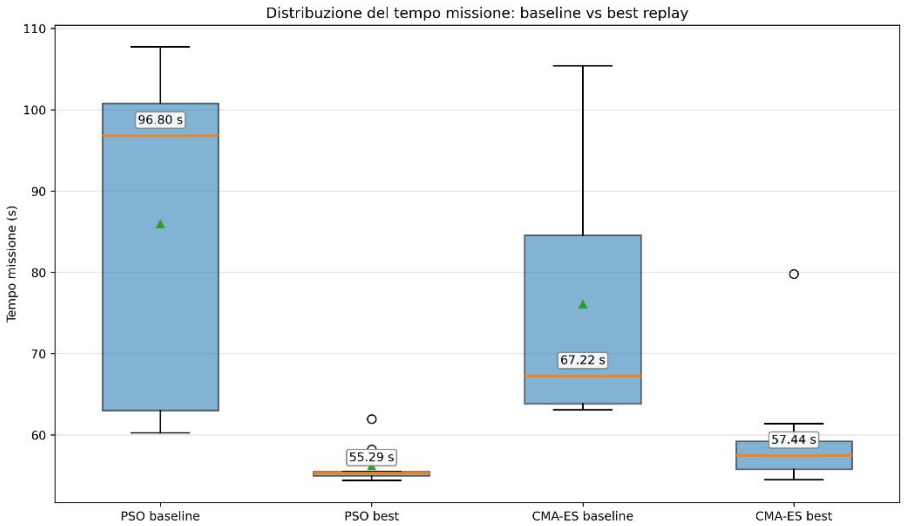
Anche la metrica di **stabilità dinamica** a_{rms}, associata alle accelerazioni del robot durante il movimento, evidenzia un miglioramento nella configurazione ottimizzata. I valori risultano mediamente inferiori e meno dispersi rispetto alla baseline, indicando traiettorie più regolari e un comportamento dinamico più stabile.

Nel complesso, l'ottimizzazione tramite CMA-ES consente quindi di ottenere una configurazione dei parametri del planner locale che migliora simultaneamente **tempo di missione, affidabilità e stabilità**, pur introducendo traiettorie leggermente più aggressive nei confronti degli ostacoli.

Metrica	Baseline	Configurazione ottimizzata	Miglioramento
Success rate	70 %	100 %	+30 %
Tempo missione mediano	67.22 s	57.44 s	−14.5 %
Tempo missione minimo	~63 s	54.04 s	migliorato
Stabilità dinamica (a_rms)	più variabile	più stabile	migliorata
Distanza minima ostacoli (d_min)	~0.16 m	~0.12 m	traiettorie più aggressive
Robustezza sistema	fallimenti presenti	nessun fallimento	migliorata

6.2.3 Confronto tra i metodi di ottimizzazione

Il confronto tra **Particle Swarm Optimization (PSO)** e **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)** nello Scenario 2 consente di valutare in modo diretto il comportamento dei due algoritmi in un contesto sperimentale più complesso rispetto allo scenario precedente. In questo caso, infatti, il robot è chiamato a operare in un ambiente caratterizzato da corridoi più stretti e dalla presenza di un ostacolo imprevisto non presente nella mappa globale, la cui gestione richiede l'intervento della procedura di recovery basata su **Artificial Potential Field (APF)**. Ne consegue che il confronto non riguarda soltanto la capacità dei due algoritmi di ridurre il tempo di missione, ma anche la loro efficacia nel produrre configurazioni di parametri robuste, stabili e sufficientemente sicure in condizioni di navigazione più critiche.

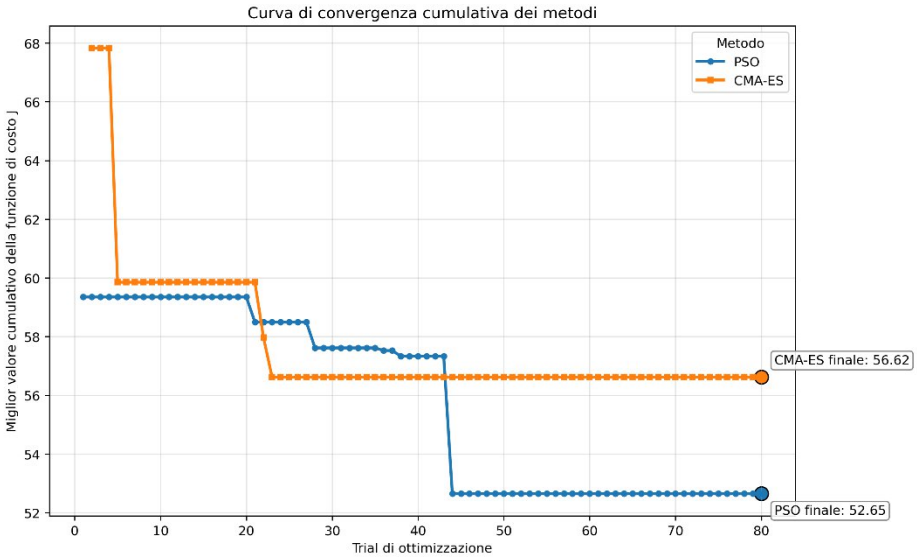


Distribuzione tempo missione scenario 2 PSO vs CMA-ES

Un primo elemento di confronto riguarda il **tempo di missione**. Dal boxplot comparativo emerge con chiarezza che entrambi gli algoritmi sono in grado di ridurre sensibilmente i tempi rispetto alle rispettive configurazioni baseline. Tuttavia, il miglioramento ottenuto con **PSO** risulta più marcato. In particolare, il tempo mediano passa da **96.80 s** nella baseline a **55.29 s** nel best replay, con una riduzione di circa **42.9%**. Nel caso di **CMA-ES**, il tempo mediano si riduce invece da

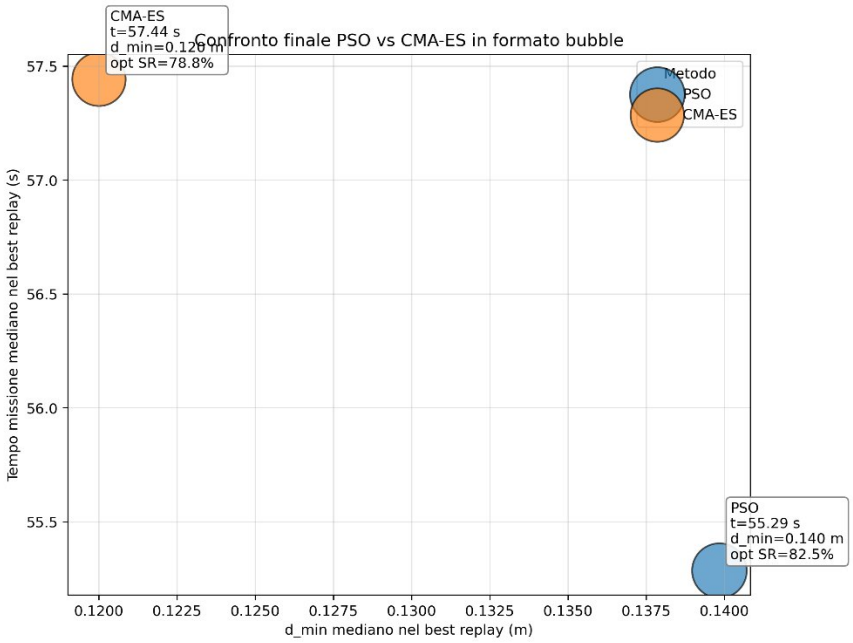
67.22 s a **57.44 s**, con un miglioramento pari a circa **14.5%**. Il risultato finale mostra quindi un lieve vantaggio di PSO sul tempo di missione, con una differenza di circa **2.15 s** rispetto a CMA-ES nel best replay. Questa differenza, pur non essendo enorme in valore assoluto, è significativa perché ottenuta in uno scenario in cui anche piccoli miglioramenti nella fluidità della traiettoria o nella gestione dei passaggi stretti possono produrre effetti apprezzabili sul tempo totale di percorrenza.

Accanto alla metrica temporale, è necessario considerare la **robustezza del processo di ottimizzazione**. In questa prospettiva, il confronto va effettuato sulla fase di optimization, cioè sulle 80 iterazioni eseguite da ciascun algoritmo. PSO ha completato con successo **66 trial su 80**, corrispondenti a un success rate dell'**82.5%**, mentre CMA-ES ha completato con successo **63 trial su 80**, corrispondenti a un success rate del **78.7%**. Entrambi gli algoritmi mostrano dunque una buona capacità di esplorazione dello spazio dei parametri, ma PSO appare leggermente più robusto nel mantenere una quota maggiore di missioni riuscite durante il processo di ricerca. Questo aspetto è importante perché indica che, nello scenario considerato, PSO ha esplorato lo spazio dei parametri in modo sufficientemente efficace senza degradare eccessivamente l'affidabilità del sistema.



Curva di convergenza cumulativa scenario 2 PSO vs CMA-ES

L'analisi della **curva di convergenza cumulativa** conferma ulteriormente questa differenza di comportamento. PSO parte da valori iniziali della funzione di costo inferiori rispetto a CMA-ES e mostra una convergenza progressiva verso la soluzione migliore, con un miglioramento netto che si manifesta soprattutto nella seconda metà delle iterazioni, fino a raggiungere il valore finale $J = 52.65$. CMA-ES, invece, presenta una discesa più rapida nelle prime iterazioni, passando da valori iniziali più elevati a una zona di prestazioni più stabili attorno a $J \approx 56.62$, ma successivamente tende a stabilizzarsi senza ulteriori miglioramenti significativi. Questo comportamento suggerisce che CMA-ES abbia trovato abbastanza rapidamente una regione promettente dello spazio dei parametri, ma che non sia poi riuscito a superarla in termini di qualità finale. PSO, al contrario, ha avuto una convergenza meno regolare ma più produttiva sul lungo periodo, riuscendo a individuare una configurazione finale migliore in termini di funzione di costo complessiva.



Confronto multidimensionale scenario 2 PSO vs CMA-ES

Un altro aspetto essenziale riguarda la **sicurezza della navigazione**, rappresentata in questo lavoro dalla distanza minima dagli ostacoli d_{min}. Dal confronto finale in formato bubble si osserva che la configurazione ottimizzata di **PSO** è associata a un valore mediano di $d_{min} \approx 0.140\, m$, mentre **CMA-ES** si attesta su un valore inferiore, pari a circa $0.120\, m$. Ciò significa che, nello Scenario 2, la soluzione ottenuta da PSO mantiene un margine di sicurezza leggermente maggiore rispetto a quella individuata da CMA-ES. Questo risultato è interessante perché smentisce l'idea, apparentemente intuitiva, secondo cui l'algoritmo più veloce dovrebbe essere anche quello più aggressivo. In realtà, nel contesto sperimentale considerato, PSO non solo produce tempi migliori, ma riesce anche a mantenere una distanza minima dagli ostacoli superiore a quella osservata per CMA-ES. La configurazione ottenuta da CMA-ES appare quindi più "tirata" nei confronti degli ostacoli, pur non restituendo un corrispondente vantaggio sul tempo di missione.

Dal punto di vista della **stabilità dinamica**, entrambi gli algoritmi mostrano un miglioramento rispetto alla baseline, ma il comportamento finale appare sostanzialmente comparabile. I valori di a_{rms}, che sintetizzano il livello di accelerazioni e quindi la regolarità del movimento, risultano più contenuti e meno dispersi nelle configurazioni ottimizzate rispetto alle baseline. In particolare, PSO mostra nel best replay una distribuzione piuttosto raccolta attorno a valori inferiori rispetto alla baseline, mentre CMA-ES evidenzia a sua volta un miglioramento della stabilità e una riduzione della dispersione. La differenza tra i due metodi su questa metrica è però meno marcata rispetto a quanto osservato per il tempo di missione e per il successo complessivo. Si può quindi affermare che, nello Scenario 2, entrambi gli algoritmi siano riusciti a produrre configurazioni dinamicamente più regolari rispetto ai parametri standard, senza che emerga una netta superiorità di uno dei due sul piano della sola stabilità.

Un ruolo centrale in questo scenario è stato svolto dal **recovery basato su APF**, e anche su questo punto il confronto offre indicazioni utili. In entrambe le ottimizzazioni, la quasi totalità delle missioni riuscite richiede l'intervento del recovery, il che conferma la natura fortemente critica dello scenario progettato. Per PSO, durante la fase di optimization, **65 successi su 66** sono avvenuti con recovery, mentre nel best replay **9 successi su 9** sono avvenuti con recovery. Per CMA-ES, durante la fase di optimization, **tutti i 63 successi** hanno richiesto recovery, e nel best replay **10 successi su 10** sono avvenuti con recovery. Questo dato mostra che, in presenza dell'ostacolo imprevisto e dei corridoi stretti, il livello reattivo introdotto con l'APF non rappresenta un supporto marginale, ma una componente strutturale della navigazione nello Scenario 2. Tuttavia, si nota che PSO è riuscito a completare almeno un trial di ottimizzazione senza recovery, mentre CMA-ES no. Anche se si tratta di un singolo episodio, esso suggerisce che le configurazioni individuate da PSO possano in alcuni casi integrarsi meglio con la logica di move_base, riducendo lievemente la dipendenza dal recovery.

L'analisi delle **cause di fallimento** supporta ulteriormente questa lettura. Nel caso di PSO, i fallimenti durante l'ottimizzazione sono pari a **14**, mentre per CMA-ES sono **17**. Per entrambi i metodi la causa dominante è la **collisione senza recovery**, ma CMA-ES mostra una frequenza leggermente maggiore di failure complessivi e una maggiore presenza di condizioni critiche legate alla gestione dei passaggi stretti e dell'ostacolo imprevisto. Ciò conferma che, nello scenario considerato, PSO è risultato non solo più efficace nel migliorare la funzione di costo, ma anche leggermente più robusto nell'esplorazione dello spazio dei parametri.

Nel complesso, il confronto tra i due algoritmi mostra che **PSO rappresenta la soluzione più vantaggiosa nello Scenario 2**. Esso ottiene infatti il miglior valore finale della funzione di costo, il minor tempo mediano di missione nel best replay, una robustezza leggermente superiore durante la fase di ottimizzazione e un margine di sicurezza finale più elevato rispetto a CMA-ES. Quest'ultimo, pur mostrando buone capacità di convergenza e una notevole affidabilità nel best replay, si arresta su una soluzione finale meno performante e più vicina agli ostacoli. Si può dunque concludere che, nelle condizioni sperimentali considerate, PSO riesca a bilanciare meglio efficienza, robustezza e sicurezza, risultando il metodo complessivamente più adatto per l'ottimizzazione dei parametri di navigazione del robot mobile nello Scenario 2.

Metrica	PSO	CMA-ES	Metodo migliore
Trial di ottimizzazione	80	80	Pari
Trial riusciti in optimization	66	63	PSO
Trial falliti in optimization	14	17	PSO
Success rate in optimization	82.5 %	78.7 %	PSO
Success rate baseline	100 %	70 %	PSO

Metrica	PSO	CMA-ES	Metodo migliore
Success rate best replay	90 %	100 %	CMA-ES
Tempo mediano baseline	96.80 s	67.22 s	CMA-ES
Tempo mediano best replay	**55.29 s**	57.44 s	**PSO**
Miglior tempo osservato	**49.47 s**	54.04 s	**PSO**
Miglior valore finale di (J)	**52.65**	56.62	**PSO**
(d_{min}) mediano best replay	**0.140 m**	0.120 m	**PSO**
Stabilità dinamica (a_{rms})	migliorata	migliorata	Pari / differenza contenuta
Successi con recovery in optimization	65	63	PSO
Successi senza recovery in optimization	1	0	PSO
Convergenza	più lenta ma più efficace	più rapida ma meno migliorativa	PSO

I risultati ottenuti nello Scenario 2 mostrano che entrambi gli algoritmi sono in grado di migliorare le prestazioni del sistema rispetto ai parametri standard, ma con caratteristiche differenti. **CMA-ES** raggiunge una soluzione stabile e affidabile, riuscendo a garantire un best replay privo di fallimenti. **PSO**, però, ottiene un risultato complessivamente migliore sotto il profilo dell'ottimizzazione: convergenza finale più favorevole, tempo di missione più basso, minore numero di fallimenti in optimization e un margine di sicurezza finale superiore. Per questo motivo, nello scenario considerato, **PSO può essere considerato l'algoritmo più efficace per l'ottimizzazione della navigazione del robot mobile in presenza di corridoi stretti e ostacoli imprevisti**.

Capitolo 7

Sintesi, conclusioni, limiti del lavoro e sviluppi futuri

7.1 Sintesi dell'implementazione degli algoritmi di ottimizzazione e dell'APF

Dal punto di vista implementativo, il lavoro sperimentale è stato realizzato attraverso lo sviluppo di un **Digital Twin del robot TurtleBot3 in ambiente ROS–Gazebo**, all'interno del quale è stato possibile eseguire in modo automatico centinaia di missioni di navigazione variando i parametri del planner locale. L'intero framework di sperimentazione è stato sviluppato in **Python**, utilizzando le interfacce ROS per il controllo della navigazione e per l'acquisizione dei dati di esecuzione.

Il sistema realizzato automatizza il ciclo sperimentale costituito dalle seguenti fasi:

1. reset dello stato del robot nella simulazione Gazebo;
2. configurazione dei parametri del planner di navigazione tramite **dynamic_reconfigure**;
3. invio del goal di navigazione al nodo **move_base**;
4. monitoraggio dell'esecuzione della missione tramite i topic ROS (/odom, /scan, /cmd_vel);
5. calcolo delle metriche di prestazione e della funzione costo utilizzata dagli algoritmi di ottimizzazione.

Gran parte dell'infrastruttura di navigazione è fornita direttamente dal framework ROS. In particolare, il pacchetto **move_base** gestisce la pianificazione globale e locale del percorso, mentre il planner locale **DWAPlannerROS** genera i comandi di velocità del robot. Nel lavoro presentato non è stato modificato il codice interno di questi moduli, ma sono stati ottimizzati alcuni dei loro parametri principali (tra cui velocità massima, accelerazione, peso dell'evitamento ostacoli e raggio di inflazione delle costmap) tramite l'interfaccia **dynamic_reconfigure**, che consente di modificarli a runtime durante l'esecuzione delle simulazioni.

Gli algoritmi di ottimizzazione sono stati integrati all'interno del Digital Twin tramite script Python dedicati. Nel caso di **CMA-ES**, l'algoritmo è stato utilizzato tramite la libreria open-source cma, che implementa l'evoluzione della matrice di covarianza per problemi di ottimizzazione continua. Lo script sviluppato gestisce la valutazione delle soluzioni candidate: per ogni vettore di parametri generato dall'algoritmo, viene eseguita una missione di navigazione completa e viene calcolato il valore della funzione costo associata. Analogamente, per l'algoritmo **PSO (Particle Swarm Optimization)** è stata implementata la logica di aggiornamento delle particelle e del miglior valore globale, mentre la valutazione delle particelle viene effettuata tramite lo stesso ciclo di simulazione del Digital Twin. In entrambi i casi, la logica di esecuzione dei trial, il logging dei risultati e la gestione del reset della simulazione sono stati sviluppati appositamente per questo lavoro.

Per migliorare la robustezza della navigazione in presenza di ostacoli imprevisti, è stato inoltre implementato un **modulo di recovery basato su Artificial Potential Fields (APF)**. Questo componente entra in funzione quando il sistema di monitoraggio rileva condizioni critiche (ad esempio stallo del robot o eccessiva vicinanza a un ostacolo). L'algoritmo calcola una forza attrattiva verso il goal e una forza repulsiva generata dalle misure del sensore laser, producendo direttamente i comandi di velocità del robot fino al superamento della situazione critica.

L’implementazione è stata sviluppata interamente in Python e integrata nel ciclo di controllo della navigazione, consentendo di riprendere successivamente l’esecuzione normale del planner di ROS.

In sintesi, l’architettura finale del sistema combina componenti esistenti del framework ROS (move_base, costmap, planner locali) con moduli software sviluppati nel corso di questo lavoro, che includono il **Digital Twin sperimentale, il sistema di monitoraggio delle missioni, l’integrazione degli algoritmi di ottimizzazione PSO e CMA-ES e il modulo di recovery basato su Artificial Potential Fields**. Questa integrazione ha permesso di realizzare un ambiente di sperimentazione automatizzato, all’interno del quale è stato possibile confrontare in modo sistematico le prestazioni dei diversi approcci di ottimizzazione analizzati.

7.2 Conclusioni, limiti e possibili sviluppi

Conclusioni

Il lavoro di tesi ha avuto come obiettivo lo sviluppo e la valutazione di un approccio basato su **Digital Twin per l’ottimizzazione dei parametri di navigazione di un robot mobile**, con particolare riferimento al planner locale DWA utilizzato nel

framework ROS. L'ambiente simulato realizzato in Gazebo ha permesso di costruire una piattaforma sperimentale automatizzata in grado di eseguire numerosi test di navigazione, raccogliere metriche di prestazione e confrontare differenti strategie di ottimizzazione.

All'interno di questo contesto sono stati analizzati e confrontati due algoritmi metaeuristici di ottimizzazione continua: **Particle Swarm Optimization (PSO)** e **Covariance Matrix Adaptation Evolution Strategy (CMA-ES)**. Entrambi gli algoritmi sono stati integrati nel Digital Twin sviluppato durante il lavoro di tesi, consentendo di valutare automaticamente numerose configurazioni dei parametri del planner locale e di identificare le soluzioni che minimizzano una funzione di costo definita in termini di **tempo di missione, sicurezza rispetto agli ostacoli e stabilità dinamica del robot**.

I risultati sperimentali mostrano che **entrambi gli algoritmi sono in grado di migliorare significativamente le prestazioni rispetto alla configurazione baseline**, dimostrando l'efficacia dell'approccio di ottimizzazione dei parametri del sistema di navigazione. In particolare, l'utilizzo del Digital Twin ha consentito di esplorare automaticamente lo spazio dei parametri del planner, individuando configurazioni che producono traiettorie più efficienti e un comportamento complessivamente più stabile del robot.

L'analisi comparativa dei due metodi ha evidenziato alcune differenze interessanti nel loro comportamento. Nello **Scenario 1**, caratterizzato da un ambiente relativamente regolare e privo di ostacoli dinamici imprevisti, l'algoritmo **CMA-ES** ha mostrato prestazioni leggermente superiori in termini di convergenza e qualità della soluzione finale. Questo risultato è coerente con le proprietà teoriche dell'algoritmo, che utilizza l'adattamento della matrice di covarianza per esplorare in modo efficiente lo spazio delle soluzioni e risulta particolarmente efficace in problemi di ottimizzazione continui con paesaggi di costo relativamente regolari.

Nel **secondo scenario sperimentale**, invece, sono state introdotte condizioni più complesse, tra cui corridoi più stretti e la presenza di un ostacolo imprevisto non presente nella mappa statica globale. In questo contesto è stato integrato un **modulo di recovery basato su Artificial Potential Fields (APF)**, progettato per consentire al robot di superare situazioni di stallo o di eccessiva vicinanza agli ostacoli. I risultati ottenuti mostrano che in questo scenario più dinamico l'algoritmo **PSO** ha dimostrato una maggiore robustezza complessiva, ottenendo prestazioni migliori rispetto a CMA-ES in termini di successo delle missioni e stabilità della navigazione.

Questo comportamento evidenzia come **non esista un algoritmo universalmente migliore in ogni contesto**, ma piuttosto come le prestazioni dipendano dalle caratteristiche specifiche dell'ambiente e del problema di navigazione. In ambienti relativamente regolari e prevedibili, CMA-ES può offrire una convergenza più efficiente verso soluzioni ottimali; al contrario, in scenari più complessi o dinamici, l'approccio di esplorazione distribuita tipico di PSO può risultare più robusto e adattabile.

Nel complesso, il lavoro svolto dimostra che l'utilizzo di un **Digital Twin per l'ottimizzazione automatica dei parametri di navigazione** rappresenta uno strumento efficace per migliorare le prestazioni dei sistemi robotici mobili. L'integrazione tra simulazione, algoritmi di ottimizzazione e moduli di gestione degli ostacoli imprevisti consente infatti di analizzare e migliorare il comportamento del robot in modo sistematico, riducendo il numero di prove necessarie su robot reale e facilitando il processo di progettazione e tuning dei sistemi di navigazione autonoma.

Limiti del lavoro

Nonostante i risultati ottenuti evidenzino l'efficacia dell'approccio proposto, è opportuno evidenziare alcuni limiti del lavoro svolto. In primo luogo, le sperimentazioni sono state condotte esclusivamente in ambiente simulato mediante il simulatore Gazebo. Sebbene l'utilizzo di un Digital Twin consenta di riprodurre in modo realistico il comportamento del robot e di eseguire un elevato numero di test in modo automatizzato, la simulazione non è in grado di rappresentare completamente tutte le complessità presenti in un sistema reale, come il rumore dei sensori, le incertezze nella dinamica del robot e le possibili variazioni dell'ambiente operativo. Di conseguenza, le prestazioni osservate in simulazione potrebbero differire parzialmente da quelle ottenibili su una piattaforma robotica fisica.

Un secondo limite riguarda la modalità con cui è stata effettuata l'ottimizzazione dei parametri del sistema di navigazione. Gli algoritmi PSO e CMA-ES sono stati utilizzati in modalità **offline**, all'interno del Digital Twin, al fine di individuare configurazioni di parametri efficaci prima dell'esecuzione delle missioni. In questo lavoro non è stata quindi implementata una strategia di **adattamento online dei parametri**, che permetterebbe al robot di modificare dinamicamente il proprio comportamento durante l'esecuzione della navigazione in risposta a variazioni dell'ambiente o delle condizioni operative.

Possibili Sviluppi futuri

Un possibile sviluppo del lavoro riguarda l'estensione dell'approccio di ottimizzazione verso modalità **online e adattive**. Nel lavoro presentato gli algoritmi PSO e CMA-ES sono stati utilizzati in modalità offline all'interno del Digital Twin, al fine di individuare configurazioni di parametri efficaci prima dell'esecuzione delle missioni. In prospettiva, sarebbe interessante sviluppare strategie di ottimizzazione **online**, in cui il robot possa adattare dinamicamente i

parametri del planner durante la navigazione, reagendo in tempo reale a cambiamenti dell'ambiente o delle condizioni operative.

Un ulteriore sviluppo riguarda la possibile **migrazione dell'architettura software verso ROS2**. Negli ultimi anni la comunità robotica sta progressivamente passando da ROS1 a ROS2, che introduce numerosi miglioramenti a livello di architettura middleware. ROS2 è basato su **DDS (Data Distribution Service)**, che consente comunicazioni più robuste, distribuite e scalabili tra i nodi del sistema, oltre a garantire migliori meccanismi di gestione della qualità del servizio (QoS). Queste caratteristiche rendono ROS2 particolarmente adatto allo sviluppo di sistemi robotici complessi e distribuiti.

La migrazione verso ROS2 permetterebbe inoltre di sfruttare le potenzialità delle nuove versioni del simulatore **Gazebo (Ignition/Gazebo Garden–Harmonic)**, progettate con un'architettura modulare e maggiormente integrate con ROS2. Le versioni più recenti del simulatore offrono miglioramenti significativi in termini di realismo fisico, gestione dei sensori, prestazioni di simulazione e possibilità di eseguire simulazioni distribuite. L'integrazione tra ROS2 e le nuove versioni di Gazebo potrebbe quindi rendere il Digital Twin ancora più realistico ed efficiente, permettendo di eseguire sperimentazioni su scenari più complessi e su sistemi robotici più articolati.

Infine, il framework sviluppato in questo lavoro potrebbe essere esteso per includere **ulteriori algoritmi di ottimizzazione** o tecniche di apprendimento automatico, al fine di confrontare differenti strategie di tuning automatico dei parametri dei sistemi di navigazione robotica.

Capitolo 8

Appendice

In questa appendice conclusiva verranno inseriti i blocchi di codice di rilievo utilizzati per la creazione dell'ambiente di lavoro, la generazione ed il controllo delle traiettorie attraverso il digital twin negli scenari proposti ed ampiamente

illustrati nei capitoli precedenti

8.1 Lemniscata

Generazione traiettoria

```
mkdir -p ~/catkin_ws/src/twin_tools/scripts

cat > ~/catkin_ws/src/twin_tools/scripts/ref_lemniscate.py << 'PY'
#!/usr/bin/env python3
import rospy, math
from geometry_msgs.msg import PointStamped

rospy.init_node("ref_lemniscate_publisher")
pub = rospy.Publisher("/ref_point", PointStamped, queue_size=10)

a = rospy.get_param("~a", 1.0)          # ampiezza [m]
omega = rospy.get_param("~omega", 0.2)  # [rad/s]
frame_id = rospy.get_param("~frame_id", "map")
rate = rospy.Rate(rospy.get_param("~rate", 30))

t0 = rospy.Time.now().to_sec()
while not rospy.is_shutdown():
    t = rospy.Time.now().to_sec() - t0
    x = a * math.sin(omega * t)
    y = a * math.sin(omega * t) * math.cos(omega * t)

    msg = PointStamped()
    msg.header.stamp = rospy.Time.now()
    msg.header.frame_id = frame_id
    msg.point.x = x
    msg.point.y = y
    msg.point.z = 0.0

    pub.publish(msg)
    rate.sleep()
PY

chmod +x ~/catkin_ws/src/twin_tools/scripts/ref_lemniscate.py
python3 ~/catkin_ws/src/twin_tools/scripts/ref_lemniscate.py
```

Digital twin che insegue traiettoria ed evita ostacoli

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
track_ref_point.py

Nodo ROS che comanda il TurtleBot3 per inseguire un punto di riferimento (/ref_point)
utilizzando un controllo proporzionale su distanza e errore angolare.

In presenza di ostacoli (da LIDAR /scan) integra:
- Modalità AVOID con isteresi e "hold" temporale (tempo minimo in avoidance).
- Safety shaping anche in TRACK: rallenta e devia se troppo vicino davanti o lateralmente.
- Anti-stallo: se la safety porta v≈0, forza una rotazione minima per sbloccarsi.
- Virtual reference (riferimento virtuale tangenziale) vicino agli ostacoli, per evitare
  che un riferimento "non feasible" (es. dietro un ostacolo) generi strusciamenti.

Autore: (tuo nome)
"""

import math
import rospy

from nav_msgs.msg import Odometry
from geometry_msgs.msg import PointStamped, Twist
from sensor_msgs.msg import LaserScan
from tf.transformations import euler_from_quaternion


class Tracker:
    def __init__(self):
        rospy.init_node("ref_point_tracker")

        # ---------------------------
        # Parametri sicurezza / LIDAR
        # ---------------------------
        self.d_safe  = rospy.get_param("~d_safe", 0.35)   # [m] soglia ingresso in avoidance (hard safety)
        self.d_clear = rospy.get_param("~d_clear", 0.45)  # [m] soglia uscita da avoidance (isteresi)
        self.v_avoid = rospy.get_param("~v_avoid", 0.08)  # [m/s] (non usata se si ruota sul posto)
        self.w_avoid = rospy.get_param("~w_avoid", 0.70)  # [rad/s] velocità angolare in AVOID

        # Hold temporale: evita ping-pong TRACK/AVOID
        self.avoid_hold  = rospy.get_param("~avoid_hold", 0.8)  # [s]
        self.avoid_until = rospy.Time(0)

        # Parametri per safety shaping in TRACK
        self.d_slow      = rospy.get_param("~d_slow", 0.80)         # [m] inizia a rallentare/deflettere
        self.d_side_safe = rospy.get_param("~d_side_safe", 0.35)    # [m] distanza minima laterale desiderata
        self.k_avoid_w   = rospy.get_param("~k_avoid_w", 1.5)       # intensità deviazione angolare in TRACK
        self.w_min_track = rospy.get_param("~w_min_track", 0.45)    # [rad/s] rotazione minima anti-stallo

        # Virtual reference: distanza del punto tangenziale
        self.d_virtual = rospy.get_param("~d_virtual", 0.50)        # [m]
        self.virtual_enable = rospy.get_param("~virtual_enable", True)

        # Settori LIDAR (in gradi) convertiti in radianti
        self.front_half_deg = rospy.get_param("~front_half_deg", 20.0)  # ±20°
        self.side_min_deg   = rospy.get_param("~side_min_deg", 20.0)    # 20° ...
        self.side_max_deg   = rospy.get_param("~side_max_deg", 80.0)    # ... 80°

        # Stato modalità
        self.mode = "TRACK"   # "TRACK" o "AVOID"
        self.turn_dir = 0     # +1 = gira a sinistra, -1 = gira a destra (latch in AVOID)

        rospy.loginfo("Hard constraint: d_safe = %.3f m", self.d_safe)

        # Distanze calcolate dal LIDAR
        self.d_min   = float("inf")  # minimo globale
        self.d_front = float("inf")  # minimo nel settore frontale
        self.min_left  = float("inf")  # minimo settore sinistra
        self.min_right = float("inf")  # minimo settore destra

        rospy.Subscriber("/scan", LaserScan, self.scan_cb, queue_size=1)

        # ---------------------------
```

```
    # Parametri tracking / limiti
    # ---------------------------
    self.k_v = rospy.get_param("~k_v", 0.6)          # guadagno velocità lineare
    self.k_w = rospy.get_param("~k_w", 1.8)          # guadagno velocità angolare
    self.v_max = rospy.get_param("~v_max", 0.20)     # [m/s]
    self.w_max = rospy.get_param("~w_max", 1.5)      # [rad/s]
    self.stop_radius = rospy.get_param("~stop_radius", 0.05)  # [m]

    self.speed_scale = rospy.get_param("~speed_scale", 1.0)

    # ---------------------------
    # Stato: riferimento e posa
    # ---------------------------
    self.ref  = None   # (rx, ry) in frame "map"
    self.pose = None   # (x, y, yaw) in frame "odom" o "map" (dipende dal setup)

    # ---------------------------
    # I/O ROS
    # ---------------------------
    self.pub = rospy.Publisher("/cmd_vel", Twist, queue_size=10)
    rospy.Subscriber("/ref_point", PointStamped, self.cb_ref, queue_size=1)
    rospy.Subscriber("/odom", Odometry, self.cb_odom, queue_size=1)

    self.rate = rospy.Rate(30)

# ------------------------------------------------------------------------------
# Callback LIDAR: calcola minimi su settori usando gli angoli del LaserScan
# ------------------------------------------------------------------------------
def scan_cb(self, msg: LaserScan):
    n = len(msg.ranges)
    if n == 0:
        self.d_min = float("inf")
        self.d_front = float("inf")
        self.min_left = float("inf")
        self.min_right = float("inf")
        return

    front_half = math.radians(self.front_half_deg)
    side_min   = math.radians(self.side_min_deg)
    side_max   = math.radians(self.side_max_deg)

    d_min   = float("inf")
    d_front = float("inf")
    d_left  = float("inf")
    d_right = float("inf")

    angle = msg.angle_min
    for r in msg.ranges:
        if math.isfinite(r) and r > 0.0:
            # minimo globale
            if r < d_min:
                d_min = r

            a = angle

            # fronte: |a| <= front_half
            if abs(a) <= front_half and r < d_front:
                d_front = r

            # sinistra: side_min <= a <= side_max (angoli positivi)
            if side_min <= a <= side_max and r < d_left:
                d_left = r

            # destra: -side_max <= a <= -side_min (angoli negativi)
            if -side_max <= a <= -side_min and r < d_right:
                d_right = r

        angle += msg.angle_increment

    self.d_min = d_min
    self.d_front = d_front
```

```
    self.min_left = d_left
    self.min_right = d_right

# ----------------------------------------
# Callback riferimento (ref point)
# ----------------------------------------
def cb_ref(self, msg: PointStamped):
    self.ref = (msg.point.x, msg.point.y)

# ----------------------------------------
# Callback odometria
# ----------------------------------------
def cb_odom(self, msg: Odometry):
    p = msg.pose.pose.position
    q = msg.pose.pose.orientation
    yaw = euler_from_quaternion([q.x, q.y, q.z, q.w])[2]
    self.pose = (p.x, p.y, yaw)

@staticmethod
def clamp(x, lo, hi):
    return max(lo, min(hi, x))

# ----------------------------------------------------------------------------
# Loop principale
# ----------------------------------------------------------------------------
def spin(self):
    while not rospy.is_shutdown():
        cmd = Twist()

        # Se mancano dati, fermo per sicurezza
        if self.ref is None or self.pose is None:
            self.pub.publish(cmd)
            self.rate.sleep()
            continue

        # Stato attuale
        rx, ry = self.ref
        x, y, yaw = self.pose

        # ------------------------------------------------------------------
        # 1) Virtual reference (solo in TRACK e vicino a ostacoli)
        #    Serve a evitare che il riferimento reale "tiri" dentro l'ostacolo.
        # ------------------------------------------------------------------
        rx_safe, ry_safe = rx, ry
        if self.virtual_enable and self.mode == "TRACK":
            # Attiva virtual ref se si è "vicini" a qualcosa davanti o lateralmente
            d_side = min(self.min_left, self.min_right)
            if (math.isfinite(self.d_front) and self.d_front < self.d_slow) or \
               (math.isfinite(d_side) and d_side < self.d_side_safe):
                angle_to_ref = math.atan2(ry - y, rx - x)

                # Scegli direzione tangenziale verso il lato più libero
                if self.min_left >= self.min_right:
                    tangential_angle = angle_to_ref + math.pi / 2.0
                else:
                    tangential_angle = angle_to_ref - math.pi / 2.0

                rx_safe = x + self.d_virtual * math.cos(tangential_angle)
                ry_safe = y + self.d_virtual * math.sin(tangential_angle)

        # ------------------------------------------------------------------
        # 2) Tracking proporzionale verso (rx_safe, ry_safe)
        # ------------------------------------------------------------------
        dx = rx_safe - x
        dy = ry_safe - y
        dist = math.hypot(dx, dy)

        target_ang = math.atan2(dy, dx)
        ang_err = target_ang - yaw
        # wrap in [-pi, pi]
        ang_err = (ang_err + math.pi) % (2.0 * math.pi) - math.pi
```

```
# Se molto vicino al punto, fermo (nota: se il ref si muove, poi riparte)
if dist < self.stop_radius:
   self.pub.publish(Twist())
   self.rate.sleep()
   continue

# Comando base di tracking (P)
v = self.k_v * dist
w = self.k_w * ang_err

cmd.linear.x  = self.clamp(v, -self.v_max, self.v_max) * self.speed_scale
cmd.angular.z = self.clamp(w, -self.w_max, self.w_max) * self.speed_scale

# ---------------------------------------------------------------------
# 3) Logica modalità TRACK / AVOID (usa d_front con isteresi + hold)
# ---------------------------------------------------------------------
if self.mode == "TRACK":
   # Entra in AVOID solo se davanti è sotto d_safe (e misura finita)
   if math.isfinite(self.d_front) and (self.d_front <= self.d_safe):
      self.mode = "AVOID"
      self.avoid_until = rospy.Time.now() + rospy.Duration.from_sec(self.avoid_hold)

      # Latch direzione: gira verso lato più libero
      if (not math.isfinite(self.min_left)) and (not math.isfinite(self.min_right)):
         self.turn_dir = +1
      else:
         self.turn_dir = (+1 if self.min_left >= self.min_right else -1)

elif self.mode == "AVOID":
   # Esci solo dopo avoid_hold e quando davanti è sufficientemente libero
   if rospy.Time.now() >= self.avoid_until:
      if math.isfinite(self.d_front) and (self.d_front >= self.d_clear):
         self.mode = "TRACK"
         self.turn_dir = 0

# ---------------------------------------------------------------------
# 4) Safety shaping in TRACK (anti-struscio)
#    Se vicino davanti o lateralmente, rallenta e devia via dall'ostacolo.
# ---------------------------------------------------------------------
if self.mode == "TRACK":
   d_side = min(self.min_left, self.min_right)

   cond_front = math.isfinite(self.d_front) and (self.d_front < self.d_slow)
   cond_side  = math.isfinite(d_side) and (d_side < self.d_side_safe)

   if cond_front or cond_side:
      # distanza efficace: la più critica tra fronte e lato
      d_eff = min(self.d_front if math.isfinite(self.d_front) else float("inf"),
                  d_side if math.isfinite(d_side) else float("inf"))

      # alpha in [0,1]: 0 = molto vicino, 1 = lontano
      alpha = self.clamp(
         (d_eff - self.d_safe) / max(1e-3, (self.d_slow - self.d_safe)),
         0.0, 1.0
      )

      # rallentamento progressivo
      cmd.linear.x *= alpha

      # sterza via dal lato più vicino (non dal lato più libero)
      if self.min_left < self.min_right:
         sgn = -1.0   # ostacolo più vicino a sinistra -> gira a destra
      else:
         sgn = +1.0   # ostacolo più vicino a destra -> gira a sinistra

      cmd.angular.z += sgn * self.k_avoid_w * (1.0 - alpha)
      cmd.angular.z = self.clamp(cmd.angular.z, -self.w_max, self.w_max)

      # anti-stallo: se v è stata annullata, imponi una rotazione minima
      if cmd.linear.x < 1e-3:
```

```
                if abs(cmd.angular.z) < self.w_min_track:
                    cmd.angular.z = (self.w_min_track if sgn > 0 else -self.w_min_track)

            # ------------------------------------------------------------------
            # 5) Comando in AVOID (priorità massima)
            #    Scelta conservativa: ruota sul posto nel verso latched.
            # ------------------------------------------------------------------
            if self.mode == "AVOID":
                cmd.linear.x = 0.0
                cmd.angular.z = self.turn_dir * self.w_avoid

            # Log (throttle 1 Hz)
            rospy.loginfo_throttle(
                1.0,
                "mode=%s d_front=%.3f d_min=%.3f d_safe=%.3f left=%.3f right=%.3f cmd=(%.2f, %.2f)",
                self.mode, self.d_front, self.d_min, self.d_safe,
                self.min_left, self.min_right, cmd.linear.x, cmd.angular.z
            )

            self.pub.publish(cmd)
            self.rate.sleep()


if __name__ == "__main__":
    try:
        Tracker().spin()
    except rospy.ROSInterruptException:
        pass
```

Grafico velocità lineare e angolare

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

odom = pd.read_csv("odom_T1.csv")

t = odom["%time"].values * 1e-9
t = t - t[0]

v_lin = odom["field.twist.twist.linear.x"].values
v_ang = odom["field.twist.twist.angular.z"].values

plt.figure()
plt.plot(t, v_lin)
plt.xlabel("Tempo [s]")
plt.ylabel("Velocità lineare [m/s]")
plt.title("Velocità lineare nel tempo - Test 1")
plt.grid()
plt.show()

plt.figure()
plt.plot(t, v_ang)
plt.xlabel("Tempo [s]")
plt.ylabel("Velocità angolare [rad/s]")
plt.title("Velocità angolare nel tempo - Test 1")
```

```
plt.grid()
plt.show()
```

Errore medio ed RMS

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Caricamento dati
odom = pd.read_csv("odom_T1.csv")
ref = pd.read_csv("ref_T1.csv")

# Tempo in secondi
t_odom = odom["%time"].values * 1e-9
t_ref  = ref["%time"].values * 1e-9

# Posizioni reali
x_real = odom["field.pose.pose.position.x"].values
y_real = odom["field.pose.pose.position.y"].values

# Riferimento
x_ref = ref["field.point.x"].values
y_ref = ref["field.point.y"].values

# Interpolazione riferimento sui tempi odom
x_ref_interp = np.interp(t_odom, t_ref, x_ref)
y_ref_interp = np.interp(t_odom, t_ref, y_ref)

# Errore euclideo
error = np.sqrt((x_real - x_ref_interp)**2 +
                (y_real - y_ref_interp)**2)

# Metriche
error_mean = np.mean(error)
error_rms  = np.sqrt(np.mean(error**2))
error_max  = np.max(error)

print(f"Errore medio = {error_mean:.4f} m")
print(f"Errore RMS   = {error_rms:.4f} m")
print(f"Errore max   = {error_max:.4f} m")

# Grafico errore nel tempo
plt.figure()
plt.plot(t_odom - t_odom[0], error)
plt.xlabel("Tempo [s]")
plt.ylabel("Errore [m]")
plt.title("Errore di tracking nel tempo - Test 1")
plt.grid()
plt.show()
```

Confronto grafico tra riferimento e traiettoria eseguita

```
import pandas as pd
import matplotlib.pyplot as plt

odom = pd.read_csv("odom_T1.csv")
ref = pd.read_csv("ref_T1.csv")

x_real = odom["field.pose.pose.position.x"]
```

```
y_real = odom["field.pose.pose.position.y"]

x_ref = ref["field.point.x"]
y_ref = ref["field.point.y"]

plt.figure()
plt.plot(x_ref, y_ref, label="Riferimento")
plt.plot(x_real, y_real, label="Reale")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.title("Confronto Traiettoria - Test 1")
plt.axis("equal")
plt.legend()
plt.grid()
plt.show()
```

8.2 Riproduzione stabilimento logistico industriale

Fisica del mondo + mura perimetrali

```
<?xml version="1.0" ?>
<sdf version="1.6">
  <world name="warehouse_simple">

    <!-- luce + gravità -->
    <include>
      <uri>model://sun</uri>
    </include>

    <!-- pavimento -->
    <include>
      <uri>model://ground_plane</uri>
    </include>

    <!-- (opzionale ma utile): fisica un po' più stabile -->
    <physics type="ode">
      <real_time_update_rate>1000</real_time_update_rate>
      <max_step_size>0.001</max_step_size>
    </physics>

    <!-- muri perimetrali: box semplici -->
    <model name="wall_north">
      <static>true</static>
      <pose>0 6 1 0 0 0</pose>
      <link name="link">
        <collision name="c"><geometry><box><size>20 0.2 2</size></box></geometry></collision>
        <visual name="v"><geometry><box><size>20 0.2 2</size></box></geometry></visual>
      </link>
    </model>

    <model name="wall_south">
      <static>true</static>
      <pose>0 -6 1 0 0 0</pose>
      <link name="link">
        <collision name="c"><geometry><box><size>20 0.2 2</size></box></geometry></collision>
        <visual name="v"><geometry><box><size>20 0.2 2</size></box></geometry></visual>
      </link>
```

```
  </model>

  <model name="wall_east">
    <static>true</static>
    <pose>10 0 1 0 0 0</pose>
    <link name="link">
      <collision name="c"><geometry><box><size>0.2 12 2</size></box></geometry></collision>
      <visual name="v"><geometry><box><size>0.2 12 2</size></box></geometry></visual>
    </link>
  </model>

  <model name="wall_west">
    <static>true</static>
    <pose>-10 0 1 0 0 0</pose>
    <link name="link">
      <collision name="c"><geometry><box><size>0.2 12 2</size></box></geometry></collision>
      <visual name="v"><geometry><box><size>0.2 12 2</size></box></geometry></visual>
    </link>
  </model>

 </world>
</sdf>
```

Scaffali (uno solo)

```
  <!-- scaffale 1: struttura industriale (montanti + 3 ripiani) -->
  <model name="rack_1">
    <static>true</static>
    <pose>0 0 0 0 0 0</pose>

    <link name="link">

      <!-- ===== COLLISIONI (servono per urti/laser) ===== -->
      <!-- montanti -->
      <collision name="col_post1">
        <pose>-1.95 -0.45 1.0 0 0 0</pose>
        <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      </collision>
      <collision name="col_post2">
        <pose>-1.95  0.45 1.0 0 0 0</pose>
        <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      </collision>
      <collision name="col_post3">
        <pose> 1.95 -0.45 1.0 0 0 0</pose>
        <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      </collision>
      <collision name="col_post4">
        <pose> 1.95  0.45 1.0 0 0 0</pose>
        <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      </collision>

      <!-- ripiani -->
      <collision name="col_shelf1">
        <pose>0 0 0.45 0 0 0</pose>
        <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      </collision>
      <collision name="col_shelf2">
        <pose>0 0 1.10 0 0 0</pose>
        <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      </collision>
      <collision name="col_shelf3">
        <pose>0 0 1.75 0 0 0</pose>
        <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      </collision>

      <!-- ===== VISUAL (quello che si vede davvero) ===== -->
```

```
    <!-- materiale montanti: grigio metallo -->
    <visual name="vis_post1">
      <pose>-1.95 -0.45 1.0 0 0 0</pose>
      <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      <material>
        <ambient>0.35 0.35 0.38 1</ambient>
        <diffuse>0.35 0.35 0.38 1</diffuse>
        <specular>0.20 0.20 0.20 1</specular>
      </material>
    </visual>
    <visual name="vis_post2">
      <pose>-1.95  0.45 1.0 0 0 0</pose>
      <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      <material>
        <ambient>0.35 0.35 0.38 1</ambient>
        <diffuse>0.35 0.35 0.38 1</diffuse>
        <specular>0.20 0.20 0.20 1</specular>
      </material>
    </visual>
    <visual name="vis_post3">
      <pose> 1.95 -0.45 1.0 0 0 0</pose>
      <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      <material>
        <ambient>0.35 0.35 0.38 1</ambient>
        <diffuse>0.35 0.35 0.38 1</diffuse>
        <specular>0.20 0.20 0.20 1</specular>
      </material>
    </visual>
    <visual name="vis_post4">
      <pose> 1.95  0.45 1.0 0 0 0</pose>
      <geometry><box><size>0.10 0.10 2.00</size></box></geometry>
      <material>
        <ambient>0.35 0.35 0.38 1</ambient>
        <diffuse>0.35 0.35 0.38 1</diffuse>
        <specular>0.20 0.20 0.20 1</specular>
      </material>
    </visual>

    <!-- materiale ripiani: arancio industriale -->
    <visual name="vis_shelf1">
      <pose>0 0 0.45 0 0 0</pose>
      <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      <material>
        <ambient>1.0 0.55 0.0 1</ambient>
        <diffuse>1.0 0.55 0.0 1</diffuse>
        <specular>0.15 0.15 0.15 1</specular>
      </material>
    </visual>
    <visual name="vis_shelf2">
      <pose>0 0 1.10 0 0 0</pose>
      <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      <material>
        <ambient>1.0 0.55 0.0 1</ambient>
        <diffuse>1.0 0.55 0.0 1</diffuse>
        <specular>0.15 0.15 0.15 1</specular>
      </material>
    </visual>
    <visual name="vis_shelf3">
      <pose>0 0 1.75 0 0 0</pose>
      <geometry><box><size>4.00 1.00 0.08</size></box></geometry>
      <material>
        <ambient>1.0 0.55 0.0 1</ambient>
        <diffuse>1.0 0.55 0.0 1</diffuse>
        <specular>0.15 0.15 0.15 1</specular>
      </material>
    </visual>

  </link>
</model>
```

Colli (uno solo)

```
<!-- colli cartone nei corridoi -->

<model name="crate_1">
<static>true</static>
<pose>-3.0 0.0 0.25 0 0 0</pose>
<link name="link">
<collision name="c"><geometry><box><size>0.60 0.40 0.50</size></box></geometry></collision>
<visual name="v">
<geometry><box><size>0.60 0.40 0.50</size></box></geometry>
<material>
<ambient>0.72 0.52 0.30 1</ambient>
<diffuse>0.72 0.52 0.30 1</diffuse>
</material>
</visual>
</link>
</model>
```

Muletti portacarichi (uno solo)

```
  <!-- muletto 1 (ostacolo statico) -->
  <model name="forklift_1">
    <static>true</static>
    <!-- posizione "random-looking" in un corridoio: cambia solo x,y se lo vuoi altrove -->
    <pose>2.5 -3.2 0 0 0 0.6</pose>

    <link name="base">

      <!-- corpo muletto -->
      <collision name="col_body">
        <pose>0 0 0.30 0 0 0</pose>
        <geometry><box><size>1.20 0.70 0.60</size></box></geometry>
      </collision>
      <visual name="vis_body">
        <pose>0 0 0.30 0 0 0</pose>
        <geometry><box><size>1.20 0.70 0.60</size></box></geometry>
        <material>
          <ambient>0.95 0.75 0.10 1</ambient>
          <diffuse>0.95 0.75 0.10 1</diffuse>
          <specular>0.10 0.10 0.10 1</specular>
        </material>
      </visual>

      <!-- tettuccio/cabina -->
      <collision name="col_cabin">
        <pose>-0.10 0 0.85 0 0 0</pose>
        <geometry><box><size>0.60 0.65 0.40</size></box></geometry>
      </collision>
      <visual name="vis_cabin">
        <pose>-0.10 0 0.85 0 0 0</pose>
        <geometry><box><size>0.60 0.65 0.40</size></box></geometry>
        <material>
          <ambient>0.15 0.15 0.15 1</ambient>
          <diffuse>0.15 0.15 0.15 1</diffuse>
          <specular>0.05 0.05 0.05 1</specular>
        </material>
      </visual>

      <!-- montante (mast) -->
      <collision name="col_mast">
        <pose>0.60 0 0.75 0 0 0</pose>
        <geometry><box><size>0.10 0.65 1.50</size></box></geometry>
      </collision>
```

```
    <visual name="vis_mast">
      <pose>0.60 0 0.75 0 0 0</pose>
      <geometry><box><size>0.10 0.65 1.50</size></box></geometry>
      <material>
        <ambient>0.10 0.10 0.10 1</ambient>
        <diffuse>0.10 0.10 0.10 1</diffuse>
      </material>
    </visual>

    <!-- forche -->
    <collision name="col_forks">
      <pose>0.95 0 0.12 0 0 0</pose>
      <geometry><box><size>0.70 0.50 0.06</size></box></geometry>
    </collision>
    <visual name="vis_forks">
      <pose>0.95 0 0.12 0 0 0</pose>
      <geometry><box><size>0.70 0.50 0.06</size></box></geometry>
      <material>
        <ambient>0.20 0.20 0.20 1</ambient>
        <diffuse>0.20 0.20 0.20 1</diffuse>
      </material>
    </visual>

    <!-- ruote (4 cilindri) -->
    <collision name="col_w1">
      <pose>0.35 0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
    </collision>
    <visual name="vis_w1">
      <pose>0.35 0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
      <material><ambient>0.05 0.05 0.05 1</ambient><diffuse>0.05 0.05 0.05 1</diffuse></material>
    </visual>

    <collision name="col_w2">
      <pose>0.35 -0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
    </collision>
    <visual name="vis_w2">
      <pose>0.35 -0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
      <material><ambient>0.05 0.05 0.05 1</ambient><diffuse>0.05 0.05 0.05 1</diffuse></material>
    </visual>

    <collision name="col_w3">
      <pose>-0.35 0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
    </collision>
    <visual name="vis_w3">
      <pose>-0.35 0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
      <material><ambient>0.05 0.05 0.05 1</ambient><diffuse>0.05 0.05 0.05 1</diffuse></material>
    </visual>

    <collision name="col_w4">
      <pose>-0.35 -0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
    </collision>
    <visual name="vis_w4">
      <pose>-0.35 -0.35 0.15 0 1.5708 0</pose>
      <geometry><cylinder><radius>0.15</radius><length>0.10</length></cylinder></geometry>
      <material><ambient>0.05 0.05 0.05 1</ambient><diffuse>0.05 0.05 0.05 1</diffuse></material>
    </visual>

  </link>
</model>
```

8.3 Digital twin (PSO)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
Digital Twin con ottimizzazione PSO (Particle Swarm Optimization) per ROS1 move_base.
- Esegue trial: reset posa -> set parametri (dynamic_reconfigure) -> invio goal -> monitor -> costo J
- PSO aggiorna particelle e converge su best globale
- Persistenza best globale su JSON (solo se migliora)
- Recovery: se STUCK / TOO_CLOSE -> modalità Bug2 (wall-follow) -> ritorno a move_base
- Plot finali:
  (1) J solo trial OK
  (2) conteggio OK/falliti per motivo
"""

import os
import time
import json
import math
import csv
import threading
import argparse
import subprocess
from datetime import datetime
from collections import deque, Counter

import numpy as np

import rospy
import actionlib
import dynamic_reconfigure.client

from gazebo_msgs.srv import SetModelState
from gazebo_msgs.msg import ModelState
from move_base_msgs.msg import MoveBaseAction, MoveBaseGoal
from std_srvs.srv import Empty
from geometry_msgs.msg import PoseWithCovarianceStamped, Twist
from nav_msgs.msg import Odometry
from sensor_msgs.msg import LaserScan


# ============================================================
# CONFIG GENERALE
# ============================================================

# Path log + best persistente
LOG_DIR = os.path.expanduser("~/twin_logs/opt_S1_pso/trials")

# ===== Modalità esperimento =====
BASELINE_TRIALS = 10       # quanti trial per baseline
BEST_REPLAY_TRIALS = 10    # quanti trial per testare il best finale

BEST_FILE = os.path.expanduser("~/twin_logs/opt_S1_pso/best_params.json")
# Etichetta metodo (per titoli plot / report)
METHOD_LABEL = "PSO"

# Timeout missione (trial)
MISSION_TIMEOUT_S = 600.0
BIG_PENALTY = 1e6

# Limiti TB3 Burger (guard-rail)
TB3_MAX_V = 0.22   # m/s
TB3_MAX_W = 2.84   # rad/s

# Scenario START/GOAL (S1) - copia i tuoi valori reali
START_X = 2.6971
START_Y = 5.0313
START_Z = 0.0
```

```
START_QX = 0.0
START_QY = 0.0
START_QZ = -0.7064
START_QW = 0.7079



GOAL_X = 3.1814
GOAL_Y = -4.1758
GOAL_Z = 0.0

GOAL_QX = 0.0
GOAL_QY = 0.0
GOAL_QZ = -0.7133
GOAL_QW = 0.7009


ROBOT_MODEL_NAME = "turtlebot3_burger"

# Namespace dynamic_reconfigure (adatta se i tuoi differiscono)
NS_DWA = "/move_base/DWAPlannerROS"
NS_LOCAL_INFL = "/move_base/local_costmap/inflation_layer"
NS_GLOBAL_INFL = "/move_base/global_costmap/inflation_layer"
# Global planner: spesso è /move_base/GlobalPlanner o /move_base/NavfnROS.
# Qui assumo GlobalPlanner (pacchetto global_planner). Se non ce l'hai, cambia.
NS_GLOBAL_PLANNER = "/move_base/NavfnROS"


# ============================================================
# PARAMETRI OTTIMIZZATI (PSO)
# - includo "leve topologiche": inflation + global planner cost settings
# ============================================================

PARAM_NAMES = [
    "max_vel_x",
    "acc_lim_x",
    "occdist_scale",
    "inflation_radius",
    "path_distance_bias",
    "goal_distance_bias",
]


BOUNDS = [
    (-0.10, TB3_MAX_V),  # max_vel_x
    (0.30, 1.20),       # acc_lim_x
    (0.01, 0.12),       # occdist_scale
    (0.15, 0.35),       # inflation_radius
    (5.0, 60.0),        # path_distance_bias
    (5.0, 40.0),        # goal_distance_bias
]


# Seed iniziale (se non c'è warmstart)
INIT_PARAMS = {
    "max_vel_x": 0.20,
    "acc_lim_x": 0.80,
    "occdist_scale": 0.02,
    "inflation_radius": 0.15,
    "path_distance_bias": 32.0,
    "goal_distance_bias": 24.0,
}

# ============================================================
# FUNZIONE COSTO
# ============================================================
W_T = 1.0     # tempo missione
W_E = 1.0     # energia proxy
W_ARMS = 10.0 # accelerazione RMS
```

```
D_MIN_THRESH = 0.05     # sotto = penalità (vicinanza ostacoli)
A_RMS_THRESH = 0.80     # sopra = penalità (aggressività)

K_DMIN = 2000.0
K_ARMS = 300.0


# ============================================================
# MONITOR ABORT / TRIGGER RECOVERY
# ============================================================
COLLISION_ABORT_DMIN = 0.2
ABORT_DMIN_SOFT = 0.25
ABORT_FRONT_SECTOR_DEG = 40
ABORT_WARN_DIST = 0.3
STALL_WINDOW_S = 5.0
STALL_MIN_TRAVEL_M = 0.05
STALL_GRACE_S = 4.0

SPIN_WINDOW_S = 6.0
SPIN_MAX_LIN = 0.01
SPIN_MIN_ANG = 0.6

MIN_PROGRESS_CHECK_S = 15.0
MIN_PROGRESS_IMPROV_M = 0.50

MAX_TIME_OK_S = 75.0
MAX_EPROXY = 3.0

# Recovery
ENABLE_RECOVERY = True
MAX_RECOVERY_ATTEMPTS = 5


# ============================================================
# LOCAL POTENTIAL FIELD AVOIDER
# ============================================================
PF_RATE_HZ = 12.0
PF_MAX_TIME_S = 10.0

PF_ATTR_GAIN = 0.45
PF_REP_GAIN = 0.20
PF_REP_RADIUS = 1.1

PF_FRONT_SECTOR_DEG = 20.0
PF_SIDE_SECTOR_DEG = 35.0

PF_FRONT_CLEAR_DIST = 0.55
PF_SIDE_CLEAR_DIST = 0.40
PF_DANGER_DIST = 0.35

PF_V_MAX = 0.12
PF_V_MIN = 0.03
PF_W_MAX = 1.2
PF_HEADING_GAIN = 1.8

PF_PROGRESS_EPS = 0.18
PF_GOAL_EPS = 0.25
PF_PROGRESS_EPS = 0.18
PF_BYPASS_MIN_TRAVEL = 0.75
PF_GOAL_EPS = 0.25

PF_STUCK_LIN_THRESH = 0.015
PF_STUCK_TIME_S = 2.0
PF_ESCAPE_ROT_TIME_S = 1.0

# ============================================================
# Utils: clamp + persist best
# ============================================================
def clamp_vec(x):
    y = []
    for v, (lo, hi) in zip(x, BOUNDS):
```

```
        y.append(min(max(float(v), lo), hi))
    return y

def _atomic_write_json(path, data):
    tmp = path + ".tmp"
    os.makedirs(os.path.dirname(path), exist_ok=True)
    with open(tmp, "w") as f:
        json.dump(data, f, indent=2)
    os.replace(tmp, path)

def read_saved_best(best_file):
    if not os.path.exists(best_file):
        return None, None, None
    try:
        with open(best_file, "r") as f:
            d = json.load(f)
        x_best = [float(d[n]) for n in PARAM_NAMES]
        x_best = clamp_vec(x_best)
        J_best = d.get("_best_J")
        t_best = d.get("_best_time_s")
        J_best = float(J_best) if J_best is not None else None
        t_best = float(t_best) if t_best is not None else None
        return x_best, J_best, t_best
    except Exception:
        return None, None, None

def save_best_if_improves(best_file, x_best, J_best, t_best):
    prev_x, prev_J, prev_t = read_saved_best(best_file)
    if prev_J is None or J_best < prev_J:
        payload = {n: float(v) for n, v in zip(PARAM_NAMES, x_best)}
        payload["_saved_at_unix"] = time.time()
        payload["_best_J"] = float(J_best)
        payload["_best_time_s"] = float(t_best)
        _atomic_write_json(best_file, payload)
        return True, prev_J
    return False, prev_J

def load_x0(best_file, default_x0):
    x_best, J_saved, _ = read_saved_best(best_file)
    if x_best is None:
        return default_x0, None
    return x_best, J_saved


# ============================================================
# Reset / setup ROS-Gazebo
# ============================================================
def reset_robot_pose():
    rospy.wait_for_service("/gazebo/set_model_state")
    set_state = rospy.ServiceProxy("/gazebo/set_model_state", SetModelState)

    state_msg = ModelState()
    state_msg.model_name = ROBOT_MODEL_NAME
    state_msg.pose.position.x = START_X
    state_msg.pose.position.y = START_Y
    state_msg.pose.position.z = START_Z
    state_msg.pose.orientation.x = START_QX
    state_msg.pose.orientation.y = START_QY
    state_msg.pose.orientation.z = START_QZ
    state_msg.pose.orientation.w = START_QW

    set_state(state_msg)
    return state_msg.pose

def reset_amcl(initial_pose):
    init_pub = rospy.Publisher("/initialpose", PoseWithCovarianceStamped, queue_size=1, latch=True)

    init = PoseWithCovarianceStamped()
    init.header.frame_id = "map"
    init.pose.pose = initial_pose
    init.pose.covariance = [0.0] * 36
```

```
    init.pose.covariance[0] = 0.05 ** 2
    init.pose.covariance[7] = 0.05 ** 2
    init.pose.covariance[35] = (5.0 * np.pi / 180.0) ** 2

    for _ in range(5):
        init.header.stamp = rospy.Time.now()
        init_pub.publish(init)
        time.sleep(0.1)
    time.sleep(1.0)

def clear_costmaps():
    rospy.wait_for_service("/move_base/clear_costmaps")
    clear_srv = rospy.ServiceProxy("/move_base/clear_costmaps", Empty)
    clear_srv()
    time.sleep(0.25)

def stop_robot_brief():
    pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)
    msg = Twist()
    for _ in range(3):
        pub.publish(msg)
        time.sleep(0.05)


# ============================================================
# Dynamic reconfigure helpers
# ============================================================
def dr_update(client, params: dict, fallback_ns: str = None):
    try:
        client.update_configuration(params)
        return True
    except Exception as e:
        if fallback_ns is None:
            rospy.logwarn(f"[DR] update failed (no fallback): {e}")
            return False
        try:
            payload = "{%s}" % (", ".join([f"{k}: {v}" for k, v in params.items()]))
            subprocess.call(["rosrun", "dynamic_reconfigure", "dynparam", "set", fallback_ns, payload])
            return True
        except Exception as e2:
            rospy.logwarn(f"[DR] fallback failed: {e2}")
            return False

def set_costmap_inflation(dr_local, dr_global, inflation_radius: float):
    ok1 = dr_update(dr_local, {"inflation_radius": float(inflation_radius)}, fallback_ns=NS_LOCAL_INFL)
    ok2 = dr_update(dr_global, {"inflation_radius": float(inflation_radius)}, fallback_ns=NS_GLOBAL_INFL)
    time.sleep(0.1)
    return ok1 and ok2

def set_dwa_params_safe(dr_dwa, max_vel_x: float, acc_lim_x: float, occdist_scale: float,
                        path_distance_bias: float, goal_distance_bias: float):
    max_vel_x = float(max_vel_x)
    acc_lim_x = float(acc_lim_x)
    occdist_scale = float(occdist_scale)
    path_distance_bias = float(path_distance_bias)
    goal_distance_bias = float(goal_distance_bias)

    params = {
        # strutturali (fissi, anti indecisione)
        "min_vel_x": -0.10,
        "vx_samples": 20,
        "vth_samples": 40,
        "sim_time": 2.0,

        # coerenza traslazionale
        "max_vel_trans": max_vel_x,
        "min_vel_trans": 0.0,

        # vincoli angolari
        "max_vel_theta": float(TB3_MAX_W),
        "min_vel_theta": 0.2,
```

```
        "acc_lim_theta": 3.0,
        "yaw_goal_tolerance": 0.35,

        # ottimizzati

        "max_vel_x": max_vel_x,
        "acc_lim_x": acc_lim_x,
        "occdist_scale": occdist_scale,
        "path_distance_bias": path_distance_bias,
        "goal_distance_bias": goal_distance_bias,
    }
    ok = dr_update(dr_dwa, params, fallback_ns=NS_DWA)
    time.sleep(0.1)
    return ok



# ============================================================
# Trial Monitor: metriche + abort reason
# ============================================================
class TrialMonitor:
    def __init__(self):
        self.lock = threading.RLock()
        self.start_wall = time.time()
        self.abort_reason = None

        self.d_goal_start = None
        self.d_goal_best = None

        self.last_v_odom = None
        self.last_t_odom = None
        self.e_proxy_cum = 0.0

        self.t_odom = []
        self.x = []
        self.y = []
        self.v = []

        self.buf = deque()  # (wall_t, x, y, v_cmd, w_cmd)
        self.dmin = float("inf")

        self.last_cmd_lin = 0.0
        self.last_cmd_ang = 0.0

        self.sub_odom = rospy.Subscriber("/odom", Odometry, self._cb_odom, queue_size=100)
        self.sub_cmd = rospy.Subscriber("/cmd_vel", Twist, self._cb_cmd, queue_size=100)
        self.sub_scan = rospy.Subscriber("/scan", LaserScan, self._cb_scan, queue_size=10)

    def stop(self):
        for s in [self.sub_odom, self.sub_cmd, self.sub_scan]:
            try:
                s.unregister()
            except Exception:
                pass

    def _set_abort(self, reason: str):
        if self.abort_reason is None:
            self.abort_reason = reason

    def _goal_distance(self, x, y):
        return math.hypot(GOAL_X - x, GOAL_Y - y)

    def _front_min_from_scan(self, msg: LaserScan):
        m = float("inf")
        a0 = msg.angle_min
        inc = msg.angle_increment

        lo = -math.radians(ABORT_FRONT_SECTOR_DEG)
        hi = +math.radians(ABORT_FRONT_SECTOR_DEG)
```

```
    for i, r in enumerate(msg.ranges):
        ang = a0 + i * inc
        if lo <= ang <= hi and math.isfinite(r) and r > 0.0:
            if r < m:
                m = r

    return m

def _cb_cmd(self, msg: Twist):
    with self.lock:
        self.last_cmd_lin = float(msg.linear.x)
        self.last_cmd_ang = float(msg.angular.z)

def _cb_scan(self, msg: LaserScan):
    m_global = float("inf")
    for r in msg.ranges:
        if math.isfinite(r) and r > 0.0 and r < m_global:
            m_global = r

    m_front = self._front_min_from_scan(msg)

    with self.lock:
        # mantieni il minimo globale come diagnostica generale
        if m_global < self.dmin:
            self.dmin = m_global

        # trigger abort/recovery SOLO su ostacolo frontale
        if math.isfinite(m_front) and m_front <= COLLISION_ABORT_DMIN:
            self.abort_reason = self.abort_reason or "COLLISION"

        elif math.isfinite(m_front) and m_front <= ABORT_DMIN_SOFT:
            self.abort_reason = self.abort_reason or f"TOO_CLOSE_FRONT (dfront={m_front:.2f}m)"

        elif math.isfinite(m_front) and m_front <= ABORT_WARN_DIST:
            self.abort_reason = self.abort_reason or f"TOO_CLOSE_WARN (dfront={m_front:.2f}m)"

def _cb_odom(self, msg: Odometry):
    wall_t = time.time()
    with self.lock:
        t = rospy.Time.now().to_sec()
        x = float(msg.pose.pose.position.x)
        y = float(msg.pose.pose.position.y)
        v = float(msg.twist.twist.linear.x)

        # (A) progresso
        d_goal = self._goal_distance(x, y)
        if self.d_goal_start is None:
            self.d_goal_start = d_goal
            self.d_goal_best = d_goal
        else:
            if d_goal < self.d_goal_best:
                self.d_goal_best = d_goal

        elapsed = wall_t - self.start_wall
        if elapsed >= MIN_PROGRESS_CHECK_S and self.d_goal_start is not None:
            improvement = self.d_goal_start - self.d_goal_best
            if improvement < MIN_PROGRESS_IMPROV_M:
                self._set_abort(f"NO_PROGRESS (improv={improvement:.2f}m in {MIN_PROGRESS_CHECK_S:.0f}s)")

        # (B) energia proxy
        t_now = t
        if self.last_t_odom is not None:
            dt = t_now - self.last_t_odom
            if dt > 1e-4 and self.last_v_odom is not None:
                a_est = (v - self.last_v_odom) / dt
                self.e_proxy_cum += abs(v * a_est) * dt
                if self.e_proxy_cum >= MAX_EPROXY:
                    self._set_abort(f"TOO_MUCH_ENERGY (E={self.e_proxy_cum:.2f} > {MAX_EPROXY:.2f})")
        self.last_t_odom = t_now
        self.last_v_odom = v
```

```
        # (C) troppo lento
        if elapsed >= MAX_TIME_OK_S:
            self._set_abort(f"TOO_SLOW (t={elapsed:.1f}s > {MAX_TIME_OK_S:.1f}s)")

        self.t_odom.append(t)
        self.x.append(x)
        self.y.append(y)
        self.v.append(v)

        self.buf.append((wall_t, x, y, self.last_cmd_lin, self.last_cmd_ang))

        keep_s = max(STALL_WINDOW_S, SPIN_WINDOW_S) + 1.0
        while self.buf and (wall_t - self.buf[0][0]) > keep_s:
            self.buf.popleft()

        if (wall_t - self.start_wall) < STALL_GRACE_S:
            return

        self._check_stall(wall_t)
        self._check_spin(wall_t)

    def _check_stall(self, now_wall):
        pts = [p for p in self.buf if (now_wall - p[0]) <= STALL_WINDOW_S]
        if len(pts) < 3:
            return
        x0, y0 = pts[0][1], pts[0][2]
        x1, y1 = pts[-1][1], pts[-1][2]
        travel = math.hypot(x1 - x0, y1 - y0)
        avg_cmd_lin = sum(p[3] for p in pts) / len(pts)
        if travel < STALL_MIN_TRAVEL_M and abs(avg_cmd_lin) > 0.03:
            self._set_abort("STUCK")

    def _check_spin(self, now_wall):
        pts = [p for p in self.buf if (now_wall - p[0]) <= SPIN_WINDOW_S]
        if len(pts) < 3:
            return
        avg_cmd_lin = sum(abs(p[3]) for p in pts) / len(pts)
        avg_cmd_ang = sum(abs(p[4]) for p in pts) / len(pts)
        if avg_cmd_lin <= SPIN_MAX_LIN and avg_cmd_ang >= SPIN_MIN_ANG:
            self._set_abort("SPIN")

    def get_abort_reason(self):
        with self.lock:
            return self.abort_reason

    def get_dmin(self):
        with self.lock:
            return float(self.dmin)

    def clear_abort_reason(self):
        with self.lock:
            self.abort_reason = None

    def reset_recovery_state(self):
        with self.lock:
            self.abort_reason = None
            self.buf.clear()
            self.dmin = float("inf")
            self.start_wall = time.time()

            if self.x and self.y:
                d_goal = self._goal_distance(self.x[-1], self.y[-1])
                self.d_goal_start = d_goal
                self.d_goal_best = d_goal
            else:
                self.d_goal_start = None
                self.d_goal_best = None

            self.last_v_odom = None
            self.last_t_odom = None
            self.e_proxy_cum = 0.0
```

```
    def get_pose_latest(self):
        with self.lock:
            if not self.x:
                return None
            return float(self.x[-1]), float(self.y[-1])

    def compute_metrics(self):
        with self.lock:
            t = np.array(self.t_odom, dtype=float)
            x = np.array(self.x, dtype=float)
            y = np.array(self.y, dtype=float)
            v = np.array(self.v, dtype=float)
            d_min = float(self.dmin)

        if len(t) < 5:
            return {"distance": float("nan"), "a_rms": float("nan"), "e_proxy": float("nan"), "d_min": d_min}

        idx = np.argsort(t)
        t = t[idx]
        x = x[idx]
        y = y[idx]
        v = v[idx]

        dt = np.diff(t)
        valid = dt > 1e-4
        if np.sum(valid) < 3:
            return {"distance": float("nan"), "a_rms": float("nan"), "e_proxy": float("nan"), "d_min": d_min}

        dx = np.diff(x)
        dy = np.diff(y)
        dist = float(np.sum(np.sqrt(dx * dx + dy * dy)))

        dv = np.diff(v)
        a = np.zeros_like(dv)
        a[valid] = dv[valid] / dt[valid]
        a_rms = float(np.sqrt(np.mean(a[valid] ** 2)))

        v_mid = 0.5 * (v[:-1] + v[1:])
        e_proxy = float(np.sum(np.abs(v_mid[valid] * a[valid]) * dt[valid]))

        return {"distance": dist, "a_rms": a_rms, "e_proxy": e_proxy, "d_min": d_min}


# ============================================================
# Potential field algorithm
# ============================================================
class LocalPotentialFieldAvoider:

    def __init__(self):

        self.goal = np.array([GOAL_X, GOAL_Y])

        self.scan = None
        self.pose = None

        self.sub_scan = rospy.Subscriber("/scan", LaserScan, self._scan_cb, queue_size=1)
        self.sub_odom = rospy.Subscriber("/odom", Odometry, self._odom_cb, queue_size=1)

        self.cmd_pub = rospy.Publisher("/cmd_vel", Twist, queue_size=1)

    def stop(self):
        try:
            self.sub_scan.unregister()
            self.sub_odom.unregister()
        except Exception:
            pass

    def _scan_cb(self, msg):
        self.scan = msg
```

```
def _odom_cb(self, msg):

    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y

    q = msg.pose.pose.orientation
    yaw = 2 * math.atan2(q.z, q.w)

    self.pose = np.array([x, y, yaw])

def _compute_repulsive(self):

    if self.scan is None:
        return np.array([0.0, 0.0])

    force = np.array([0.0, 0.0])

    angle = self.scan.angle_min

    for r in self.scan.ranges:

        if not math.isfinite(r):
            angle += self.scan.angle_increment
            continue

        if r < PF_REP_RADIUS:

            strength = PF_REP_GAIN * (1.0/r - 1.0/PF_REP_RADIUS)

            fx = -strength * math.cos(angle)
            fy = -strength * math.sin(angle)

            force += np.array([fx, fy])

        angle += self.scan.angle_increment

    return force

def _compute_attractive(self):

    if self.pose is None:
        return np.array([0.0, 0.0])

    pos = self.pose[:2]

    diff = self.goal - pos
    dist = np.linalg.norm(diff)

    if dist < 1e-6:
        return np.array([0.0, 0.0])

    return PF_ATTR_GAIN * diff / dist

def run(self):

    rate = rospy.Rate(PF_RATE_HZ)

    start = time.time()

    last_progress = time.time()
    last_pos = None

    while not rospy.is_shutdown():

        if self.pose is None:
            rate.sleep()
            continue

        pos = self.pose[:2]
```

```
            if np.linalg.norm(self.goal - pos) < PF_GOAL_EPS:
                return "GOAL_REACHED"

            if last_pos is None:
                last_pos = pos

            if np.linalg.norm(pos - last_pos) > PF_PROGRESS_EPS:
                last_progress = time.time()
                last_pos = pos

            if time.time() - last_progress > PF_STUCK_TIME_S:
                self._escape_rotation()
                last_progress = time.time()

            F_attr = self._compute_attractive()
            F_rep = self._compute_repulsive()

            F = F_attr + F_rep

            heading = math.atan2(F[1], F[0])

            yaw = self.pose[2]

            err = math.atan2(math.sin(heading - yaw), math.cos(heading - yaw))

            v = PF_V_MAX * max(0.0, math.cos(err))
            w = PF_HEADING_GAIN * err

            v = max(PF_V_MIN, min(PF_V_MAX, v))
            w = max(-PF_W_MAX, min(PF_W_MAX, w))

            cmd = Twist()
            cmd.linear.x = v
            cmd.angular.z = w

            self.cmd_pub.publish(cmd)

            if time.time() - start > PF_MAX_TIME_S:
                return "TIMEOUT"

            rate.sleep()

    def _escape_rotation(self):

        start = time.time()

        rate = rospy.Rate(10)

        while time.time() - start < PF_ESCAPE_ROT_TIME_S:

            cmd = Twist()
            cmd.angular.z = PF_W_MAX

            self.cmd_pub.publish(cmd)

            rate.sleep()


# ============================================================
# Move base: send goal + monitor + recovery
# ============================================================
def send_goal_and_monitor_with_recovery(client, goal, timeout_s=MISSION_TIMEOUT_S):
    monitor = TrialMonitor()

    recovery_attempts = 0
    start_wall = time.time()

    client.send_goal(goal)
    rate = rospy.Rate(10)

    try:
```

```
        while not rospy.is_shutdown():
            state = client.get_state()
            elapsed = time.time() - start_wall

            if state == actionlib.GoalStatus.SUCCEEDED:
                return True, elapsed, "GOAL_REACHED", monitor

            if elapsed > timeout_s:
                client.cancel_all_goals()
                return False, elapsed, "TIMEOUT", monitor

            reason = monitor.get_abort_reason()

            if ENABLE_RECOVERY and reason is not None:
                if recovery_attempts < MAX_RECOVERY_ATTEMPTS:
                    recovery_attempts += 1

                    rospy.logwarn(
                        f"[RECOVERY] trigger: {reason} | tentativo {recovery_attempts}/{MAX_RECOVERY_ATTEMPTS}"
                    )

                    client.cancel_all_goals()
                    time.sleep(0.3)
                    stop_robot_brief()

                    avoider = LocalPotentialFieldAvoider()
                    rec_start = time.time()
                    try:
                        rec = avoider.run()
                    finally:
                        avoider.stop()

                    rec_elapsed = time.time() - rec_start
                    rospy.logwarn(f"[RECOVERY] esito: {rec} | durata={rec_elapsed:.2f}s")

                    stop_robot_brief()
                    clear_costmaps()

                    monitor.clear_abort_reason()
                    monitor.reset_recovery_state()

                    goal.target_pose.header.stamp = rospy.Time.now()
                    client.send_goal(goal)

                    time.sleep(0.3)
                    continue
                else:
                    client.cancel_all_goals()
                    return False, elapsed, f"{reason}_NO_RECOVERY", monitor

            if state in [
                actionlib.GoalStatus.ABORTED,
                actionlib.GoalStatus.REJECTED,
                actionlib.GoalStatus.PREEMPTED,
                actionlib.GoalStatus.RECALLED,
                actionlib.GoalStatus.LOST,
            ]:
                final_reason = reason if reason is not None else f"MOVE_BASE_STATE_{state}"
                client.cancel_all_goals()
                return False, elapsed, final_reason, monitor

            rate.sleep()

    finally:
        monitor.stop()
# ============================================================
# Logging + plotting
# ============================================================
def _normalize_reason(reason: str) -> str:
    if reason is None:
        return "UNKNOWN"
```

```
    r = str(reason).strip()
    for sep in [" (", ":", " |", " ["]:
        if sep in r:
            r = r.split(sep, 1)[0].strip()
    return r or "UNKNOWN"

def make_summary_plots_from_csv(csv_path: str, method_label: str = METHOD_LABEL):
    import matplotlib
    matplotlib.use("Agg")
    import matplotlib.pyplot as plt

    trials_ok, Js_ok = [], []
    n_total, n_ok, n_fail = 0, 0, 0
    fail_reasons = []

    with open(csv_path, "r") as f:
        reader = csv.DictReader(f)
        for row in reader:
            try:
                tid = int(row["trial_idx"])
            except Exception:
                continue

            n_total += 1
            ok = False
            try:
                ok = int(row["success"]) == 1
            except Exception:
                ok = False

            if ok:
                n_ok += 1
                try:
                    J = float(row["J"])
                except Exception:
                    continue
                trials_ok.append(tid)
                Js_ok.append(J)
            else:
                n_fail += 1
                fail_reasons.append(_normalize_reason(row.get("reason", "UNKNOWN")))

    if n_total == 0:
        print("[PLOT] Nessun dato nel CSV, salto.")
        return

    base = os.path.splitext(csv_path)[0]

    # (1) J solo OK
    fig1 = plt.figure()
    if trials_ok:
        plt.plot(trials_ok, Js_ok, marker="o")
        plt.xlabel("Trial (solo OK)")
        plt.ylabel("Costo J")
        plt.title(f"Andamento funzione di costo (trial riusciti) - {method_label}")
        plt.grid(True)
    else:
        plt.text(0.5, 0.5, "Nessun trial OK", ha="center", va="center")
        plt.axis("off")
    out1 = base + f"_{method_label}_cost_OK.png"
    fig1.savefig(out1, dpi=200, bbox_inches="tight")
    plt.close(fig1)

    # (2) outcome + fail per reason (stacked)
    counts = Counter(fail_reasons)
    reasons_sorted = [k for k, _ in counts.most_common()]
    fail_counts_sorted = [counts[k] for k in reasons_sorted]

    fig2 = plt.figure()
    plt.bar([0], [n_ok])
```

```
    fail_base = 0
    if n_fail > 0:
        for r, c in zip(reasons_sorted, fail_counts_sorted):
            plt.bar([1], [c], bottom=fail_base, label=r)
            fail_base += c

    plt.xticks([0, 1], ["OK", "FALLITI"])
    plt.ylabel("Numero trial")
    plt.title(f"Esito ({method_label}): Tot={n_total} | OK={n_ok} | Fail={n_fail}")
    plt.grid(True, axis="y")
    if n_fail > 0:
        plt.legend(title="Reason", loc="best")

    out2 = base + f"_{method_label}_summary_outcomes.png"
    fig2.savefig(out2, dpi=200, bbox_inches="tight")
    plt.close(fig2)

    subprocess.Popen(["xdg-open", out1])
    subprocess.Popen(["xdg-open", out2])

    print("[PLOT] Salvati grafici:")
    print(f"  - {out1}")
    print(f"  - {out2}")
    if n_fail:
        print("[PLOT] Fallimenti per tipologia:")
        for k, v in counts.most_common():
            print(f"  - {k}: {v}")


# ============================================================
# Cost function evaluation (trial)
# ============================================================
class Evaluator:
    def __init__(self, dr_dwa, dr_local_infl, dr_global_infl, dr_global_planner, log_writer, log_fh):
        self.dr_dwa = dr_dwa
        self.dr_local_infl = dr_local_infl
        self.dr_global_infl = dr_global_infl
        self.dr_global_planner = dr_global_planner
        self.log_writer = log_writer
        self.log_fh = log_fh

        self.trial_counter = 0
        self.best_J = float("inf")
        self.best_params = None
        self.best_time = None

    def _log_trial(self, row):
        self.log_writer.writerow(row)
        self.log_fh.flush()

    def evaluate(self, x):
        x = clamp_vec(x)
        (max_vel_x, acc_lim_x, occdist_scale, inflation_radius, path_distance_bias, goal_distance_bias) = x

        self.trial_counter += 1
        tid = self.trial_counter

        print(f"\n[TRIAL {tid}]")
        print(f"  max_vel_x={max_vel_x:.3f}, acc_lim_x={acc_lim_x:.3f}, occdist_scale={occdist_scale:.3f}")
        print(f"  inflation_radius={inflation_radius:.3f}, path_bias={path_distance_bias:.1f},
goal_bias={goal_distance_bias:.1f}")

        monitor = None

        try:
            # 1) aggiorna parametri runtime
            set_dwa_params_safe(self.dr_dwa, max_vel_x, acc_lim_x, occdist_scale,
                                path_distance_bias, goal_distance_bias)
            set_costmap_inflation(self.dr_local_infl, self.dr_global_infl, inflation_radius)
```

```
        # 2) reset
        pose_used = reset_robot_pose()
        time.sleep(0.6)
        reset_amcl(pose_used)
        clear_costmaps()
        stop_robot_brief()

        # 3) missione
        client = actionlib.SimpleActionClient("/move_base", MoveBaseAction)
        client.wait_for_server()

        goal = MoveBaseGoal()
        goal.target_pose.header.frame_id = "map"
        goal.target_pose.header.stamp = rospy.Time.now()
        goal.target_pose.pose.position.x = GOAL_X
        goal.target_pose.pose.position.y = GOAL_Y
        goal.target_pose.pose.position.z = GOAL_Z
        goal.target_pose.pose.orientation.x = GOAL_QX
        goal.target_pose.pose.orientation.y = GOAL_QY
        goal.target_pose.pose.orientation.z = GOAL_QZ
        goal.target_pose.pose.orientation.w = GOAL_QW

        success, mission_time, reason, monitor = send_goal_and_monitor_with_recovery(
            client=client,
            goal=goal,
            timeout_s=MISSION_TIMEOUT_S
        )

        stop_robot_brief()



        metrics = monitor.compute_metrics()
        dist = metrics["distance"]
        a_rms = metrics["a_rms"]
        e_proxy = metrics["e_proxy"]
        d_min = metrics["d_min"]

        if not success:
            print(f"  -> FAIL: {reason}")
            clear_costmaps()
            J = BIG_PENALTY
            self._log_trial([tid, *x, 0, reason, mission_time, dist, a_rms, d_min, e_proxy, J])
            return J

        if (not math.isfinite(dist)) or (not math.isfinite(a_rms)) or (not math.isfinite(e_proxy)):
            print("  -> Metriche NaN -> penalità")
            J = BIG_PENALTY
            self._log_trial([tid, *x, 0, "NAN_METRICS", mission_time, dist, a_rms, d_min, e_proxy, J])
            return J

        penalty = 0.0
        if math.isfinite(d_min):
            viol_d = max(0.0, D_MIN_THRESH - d_min)
            penalty += K_DMIN * (viol_d ** 2)
        else:
            penalty += 1e4

        viol_a = max(0.0, a_rms - A_RMS_THRESH)
        penalty += K_ARMS * (viol_a ** 2)

        J = (W_T * mission_time) + (W_E * e_proxy) + (W_ARMS * a_rms) + penalty

        print(f"  -> OK: T={mission_time:.2f}s | d={dist:.2f}m | a_RMS={a_rms:.2f} | d_min={d_min:.2f}m |
E={e_proxy:.2f} | J={J:.2f}")

        self._log_trial([tid, *x, 1, "OK", mission_time, dist, a_rms, d_min, e_proxy, J])

        if J < self.best_J:
            self.best_J = J
            self.best_params = x
```

```
            self.best_time = mission_time

        return J

    except Exception as e:
        print(f"  -> EXCEPTION: {e}")
        try:
            if monitor is not None:
                monitor.stop()
        except Exception:
            pass
        J = BIG_PENALTY
        self._log_trial([tid, *x, 0, "EXCEPTION", MISSION_TIMEOUT_S,
                         float("nan"), float("nan"), float("nan"), float("nan"), J])
        return J


# ============================================================
# PSO core
# ============================================================
class PSO:
    def __init__(self, x0, bounds, n_particles=10, iters=8, w=0.72, c1=1.4, c2=1.4, seed=1):
        self.rng = np.random.default_rng(seed)
        self.bounds = np.array(bounds, dtype=float)  # shape (D, 2)
        self.D = self.bounds.shape[0]

        self.n_particles = int(n_particles)
        self.iters = int(iters)

        self.w = float(w)
        self.c1 = float(c1)
        self.c2 = float(c2)

        lo = self.bounds[:, 0]
        hi = self.bounds[:, 1]

        # init swarm: attorno a x0 + random spread
        x0 = np.array(x0, dtype=float)
        span = (hi - lo)
        self.X = np.zeros((self.n_particles, self.D), dtype=float)
        self.V = np.zeros((self.n_particles, self.D), dtype=float)

        for i in range(self.n_particles):
            jitter = self.rng.normal(0.0, 0.15, size=self.D) * span
            self.X[i, :] = np.clip(x0 + jitter, lo, hi)
            self.V[i, :] = self.rng.normal(0.0, 0.05, size=self.D) * span

        self.pbest_X = self.X.copy()
        self.pbest_J = np.full((self.n_particles,), np.inf, dtype=float)

        self.gbest_X = x0.copy()
        self.gbest_J = np.inf

    def step(self, evaluator_fn):
        # valuta particelle
        for i in range(self.n_particles):
            J = float(evaluator_fn(self.X[i, :].tolist()))
            if J < self.pbest_J[i]:
                self.pbest_J[i] = J
                self.pbest_X[i, :] = self.X[i, :].copy()
            if J < self.gbest_J:
                self.gbest_J = J
                self.gbest_X = self.X[i, :].copy()

        # update velocità/posizioni
        lo = self.bounds[:, 0]
        hi = self.bounds[:, 1]
        span = (hi - lo)

        for i in range(self.n_particles):
            r1 = self.rng.random(self.D)
```

```
            r2 = self.rng.random(self.D)

            cognitive = self.c1 * r1 * (self.pbest_X[i, :] - self.X[i, :])
            social = self.c2 * r2 * (self.gbest_X - self.X[i, :])

            self.V[i, :] = self.w * self.V[i, :] + cognitive + social

            # clamp velocità (evita salti idioti)
            v_max = 0.25 * span
            self.V[i, :] = np.clip(self.V[i, :], -v_max, v_max)

            self.X[i, :] = self.X[i, :] + self.V[i, :]

            # clamp posizione ai bounds
            self.X[i, :] = np.clip(self.X[i, :], lo, hi)

    def run(self, evaluator_fn):
        for k in range(self.iters):
            print(f"\n=== PSO ITER {k+1}/{self.iters} | gbest_J={self.gbest_J:.2f} ===")
            self.step(evaluator_fn)
        return self.gbest_X.tolist(), float(self.gbest_J)


# ============================================================
# MAIN
# ============================================================
if __name__ == "__main__":
    rospy.init_node("twin_optimizer_cmaes_patched", anonymous=True)

    parser = argparse.ArgumentParser()
    parser.add_argument("--mode", choices=["opt", "baseline", "best_replay"], default="opt")
    args = parser.parse_args()
    os.makedirs(LOG_DIR, exist_ok=True)
    log_path = os.path.join(LOG_DIR, f"pso_log_{datetime.now().strftime('%Y%m%d_%H%M%S')}.csv")

    log_fh = open(log_path, "w", newline="")
    log_wr = csv.writer(log_fh)

    log_wr.writerow([
        "trial_idx",
        *PARAM_NAMES,
        "success",
        "reason",
        "mission_time_s",
        "distance_m",
        "a_rms",
        "d_min",
        "e_proxy",
        "J"
    ])

    print(f"[LOG] Scrivo log su: {log_path}")
    print(f"[VINCOLI] TB3 max v={TB3_MAX_V:.2f} m/s | max w={TB3_MAX_W:.2f} rad/s")
    print(f"[ABORT] collision dmin<={COLLISION_ABORT_DMIN:.2f} m | stall={STALL_WINDOW_S:.1f}s |
spin={SPIN_WINDOW_S:.1f}s")

    # dynamic reconfigure clients
    dr_dwa = dynamic_reconfigure.client.Client(NS_DWA, timeout=5.0)
    dr_local_infl = dynamic_reconfigure.client.Client(NS_LOCAL_INFL, timeout=5.0)
    dr_global_infl = dynamic_reconfigure.client.Client(NS_GLOBAL_INFL, timeout=5.0)
    dr_gp = None

    default_x0 = [INIT_PARAMS[n] for n in PARAM_NAMES]
    x0, saved_J = load_x0(BEST_FILE, default_x0)
    if x0 != default_x0:
        if saved_J is not None:
            print("[WARMSTART] Trovato best precedente:")
            print(f"  x0={x0} | J_saved={saved_J:.2f}")
        else:
            print("[WARMSTART] Trovato best precedente:")
            print(f"  x0={x0}")
```

```
else:
    print("[WARMSTART] Nessun best precedente, uso INIT_PARAMS.")

evaluator = Evaluator(dr_dwa, dr_local_infl, dr_global_infl, dr_gp, log_wr, log_fh)

    # ===== MODE: baseline / best replay =====
if args.mode == "baseline":
    print(f"[MODE] baseline: eseguo {BASELINE_TRIALS} trial con INIT_PARAMS")
    x_baseline = [INIT_PARAMS[n] for n in PARAM_NAMES]
    try:
        for _ in range(BASELINE_TRIALS):
            evaluator.evaluate(x_baseline)
    finally:
        try:
            if LOG_FH:
                LOG_FH.close()
        except Exception:
            pass

    try:
        make_summary_plots_from_csv(log_path, METHOD_LABEL)
    except Exception as e:
        print(f"[PLOT] Errore generazione grafici: {e}")
    raise SystemExit(0)

if args.mode == "best_replay":
    x_best, J_saved2, _ = read_saved_best(BEST_FILE)
    if x_best is None:
        print("[MODE] best_replay: nessun best salvato, esco.")
        raise SystemExit(1)

    print(f"[MODE] best_replay: eseguo {BEST_REPLAY_TRIALS} trial con best salvato | J_saved={J_saved2}")
    try:
        for _ in range(BEST_REPLAY_TRIALS):
            evaluator.evaluate(x_best)
    finally:
        try:
            if LOG_FH:
                LOG_FH.close()
        except Exception:
            pass

    try:
        make_summary_plots_from_csv(log_path, METHOD_LABEL)
    except Exception as e:
        print(f"[PLOT] Errore generazione grafici: {e}")
    raise SystemExit(0)


# PSO settings: pochi iter e poche particelle, perché ogni trial costa tempo.
pso = PSO(
    x0=x0,
    bounds=BOUNDS,
    n_particles=8,
    iters=10,
    w=0.72,
    c1=1.4,
    c2=1.4,
    seed=2
)

try:
    gbest_x, gbest_J = pso.run(evaluator.evaluate)
finally:
    try:
        log_fh.close()
    except Exception:
        pass

# Plot finali
try:
```

```
        make_summary_plots_from_csv(log_path, METHOD_LABEL)
    except Exception as e:
        print(f"[PLOT] errore: {e}")

    # Persistenza best globale (solo se migliora)
    if evaluator.best_params is None:
        print("\nNessun trial riuscito: niente best da salvare.")


    saved, prev_J = save_best_if_improves(BEST_FILE, evaluator.best_params, evaluator.best_J, evaluator.best_time)

    if saved:
        if prev_J is None:
            print(f"[WARMSTART] Salvato primo best globale in: {BEST_FILE}")
        else:
            print(f"[WARMSTART] Best globale migliorato: {prev_J:.2f} -> {evaluator.best_J:.2f} | salvato in:
{BEST_FILE}")
    else:
        print(f"[WARMSTART] Best NON migliorato (best salvato resta {prev_J:.2f}). File non sovrascritto.")


    # se mode == "opt" continua normalmente sotto (CMA-ES)

    print("\n=== RISULTATO OTTIMIZZAZIONE (PSO) ===")
    for n, v in zip(PARAM_NAMES, evaluator.best_params):
        print(f"{n}: {v:.4f}")
    print(f"Tempo missione best: {evaluator.best_time:.2f} s")
    print(f"J best: {evaluator.best_J:.2f}")

    print("\n[INFO] PSO gbest (ultimo visto):")
    for n, v in zip(PARAM_NAMES, gbest_x):
        print(f"{n}: {v:.4f}")
    print(f"gbest_J: {gbest_J:.2f}")


if __name__ == "__main__":
    main()
```

Bibliografia e sitografia

[1] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Robotica. Modellistica, pianificazione e controllo, McGraw-Hill Education, 2008

[2] M. Quigley et al., “ROS: an open-source Robot Operating System,” in ICRA Workshop on Open Source Software, 2009.

[3] T. Foote, "tf: The transform library," 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), Woburn, MA, USA, 2013, pp. 1-6

[4] N. Koenig and A. Howard, “Design and Use Paradigms for Gazebo, an Open-Source Multi-Robot Simulator,” in IEEE/RSJ IROS, 2004.

[5] ROBOTIS, “TurtleBot3 e-Manual: Overview,” (documentazione ufficiale).

[6] ROBOTIS, “TurtleBot3 e-Manual: Simulation,” (documentazione ufficiale).

[7] M. Grieves, “Digital Twin: Manufacturing Excellence through Virtual Factory Replication,” White Paper, 2014.

[8] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, “Digital Twin in manufacturing: A categorical literature review and classification,” IFAC-PapersOnLine, 2018

[9] ROS Wiki, “rosbag,” (documentazione).

[10] ROS (Noetic) API Docs, “sensor_msgs/LaserScan message definition,” (documentazione).

[11] Dispense del professore Alberto Finzi, 2021.

http://wpage.unina.it/alberto.finzi/didattica/IROB/materiale/IR-Lezione2a.pdf

[12] D. Santarelli, “Algoritmi di Navigazione e Orientamento per Veicoli Autonomi e Robotici in Ambiente ROS,” Tesi di Laurea Magistrale, Università Politecnica delle Marche, Ancona, Italia, 2021

[13] S. Thrun, W. Burgard, and D. Fox, Probabilistic Robotics, MIT Press, 2005

[14] J. Kennedy and R. Eberhart, “Particle Swarm Optimization,” Proceedings of the IEEE International Conference on Neural Networks, 1995.

[15] N. Hansen, “The CMA Evolution Strategy: A Tutorial,” arXiv:1604.00772, 2016.

[16] D. Fox, W. Burgard, and S. Thrun, “The Dynamic Window Approach to Collision Avoidance,” IEEE Robotics & Automation Magazine, 1997.

[17] ROS Wiki, “RViz Visualization Tool,” https://wiki.ros.org/rviz

[18] O. Khatib, “Real-Time Obstacle Avoidance for Manipulators and Mobile Robots,” *The International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[19] R. Siegwart, I. Nourbakhsh, and D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, 2nd ed., MIT Press, 2011.