
Università degli Studi di Firenze

Scuola di Ingegneria - Dipartimento di Ingegneria dell'Informazione


---

Tesi di Laurea Triennale in Ingegneria Informatica

# Studio sulle Leggi di Controllo per il Vehicle Platooning: Implementazione e Simulazione con BlenSor


*Candidato*
Chiara Gori

*Relatore*
Prof. Giorgio Battistelli


---


Anno Accademico 2023/2024

# Indice

# Introduzione

Negli ultimi decenni, l'interesse per i sistemi di guida autonoma e la loro integrazione nel traffico stradale ha registrato un costante aumento. Questo interesse è motivato dai potenziali vantaggi in termini di sicurezza stradale, efficienza del traffico e riduzione delle emissioni. Tra i numerosi scenari di utilizzo delle tecnologie di guida autonoma, il concetto di platooning veicolare si è rivelato particolarmente interessante per la sua semplicità abbinata alla frequenza con cui questa situazione ricorre nella vita quotidiana.

La mia ricerca si concentra sull'implementazione e la simulazione di sistemi di platooning veicolare utilizzando BlenSor, una libreria di simulazione open-source basata su Blender. Blender è ampiamente noto per la sua versatilità e la sua capacità di creare ambienti virtuali realistici, e l'aggiunta di BlenSor ha esteso le sue funzionalità, combinandole ad un utilizzo accurato dei sensori LiDAR, fondamentali per la guida autonoma e la percezione ambientale. In particolare, mi propongo di studiare il comportamento di una serie di veicoli autonomi, costituita da automobili di medie prestazioni e, in numero minore, di autobus, ognuno dotato di un proprio controllore, all'interno di uno scenario simulato. Questi controllori giocano un ruolo fondamentale nelle leggi di controllo dei veicoli e sono responsabili della generazione delle loro traiettorie, tenendo conto di parametri cruciali come la velocità, l'accelerazione e la distanza di sicurezza tra veicoli adiacenti.

Il mio obiettivo è quello di esaminare l'efficacia delle leggi di controllo im-

plementate per garantire una guida sicura e coordinata dei veicoli all'interno del platoon. Attraverso l'analisi dei risultati ottenuti e delle simulazioni su BlenSor, intendo valutare il sistema in termini di sicurezza e stabilità sottoponendolo a una gamma di situazioni verosimili.

Il presente lavoro si articola in diverse sezioni. Nel primo capitolo verrà esaminata una particolare definizione delle leggi di controllo dei veicoli facenti parte del platoon. Inoltre, saranno illustrate nel dettaglio le decisioni implementative prese per la stesura del codice, al fine di garantirne la coerenza con le equazioni nel dominio del tempo continuo e, al contempo, introdurre limiti fisici per permettere di simulare con accuratezza situazioni reali. Successivamente, verrà descritto in breve il framework di simulazione utilizzato e i suoi aspetti rilevanti in merito a questo studio. Saranno presentati e discussi i risultati ottenuti sia per quanto riguarda la generazione delle traiettorie che le simulazioni su Blender, seguiti da conclusioni e possibili estensioni future.

# Capitolo 1

# Leggi di controllo dei veicoli e loro implementazione

## 1.1 Introduzione al problema del vehicle platooning

Il vehicle platooning, o convogliamento dei veicoli, è un concetto che si riferisce alla pratica di far viaggiare una serie di veicoli in formazione ravvicinata, coordinando le loro velocità e le distanze tra uno e l'altro. Questa tecnica, che può essere realizzata sia con veicoli guidati da esseri umani che con veicoli autonomi, offre numerosi vantaggi in vari settori legati alla mobilità.

Innanzitutto, il vehicle platooning può portare a significativi miglioramenti in termini di sicurezza stradale. La coordinazione dei veicoli all'interno del convoglio, mantenendo una distanza sicura e una velocità uniforme, può ridurre il rischio di incidenti causati da errori umani come frenate improvvise o manovre errate. Inoltre, la strategia di viaggio in formazione ravvicinata può facilitare una migliore comunicazione e coordinazione tra i veicoli, consentendo una risposta più rapida e

coordinata agli imprevisti sulla strada.

Un altro vantaggio chiave del vehicle platooning è l'efficienza del traffico e il risparmio di carburante. Riducendo l'attrito dell'aria grazie alla formazione ravvicinata e il numero di accelerazioni e decelerazioni, i veicoli all'interno del platoon possono ridurre il consumo di carburante, contribuendo così alla riduzione delle emissioni e alla sostenibilità ambientale. Inoltre, il platooning dei veicoli può portare a una maggiore capacità di trasporto su strada, in quanto più veicoli possono viaggiare lungo la stessa strada nello stesso momento, riducendo la congestione del traffico.

Il vehicle platooning è anche considerato un passo importante verso la guida autonoma e la mobilità connessa. Poiché richiede la comunicazione e la coordinazione tra i veicoli all'interno del convoglio, il platooning fornisce un terreno fertile per lo sviluppo e il collaudo di tecnologie avanzate, come i sistemi di comunicazione veicolo-veicolo (V2V) e di comunicazione veicolo-infrastruttura (V2I). Inoltre, il platooning può essere considerato come un inizio di transizione verso una futura rete stradale intelligente, in cui i veicoli interagiscono in modo coordinato con l'ambiente circostante per ottimizzare la sicurezza e l'efficienza.

## 1.2 Cooperative Adaptive Cruise Control (CACC) e leggi di controllo

Nel contesto della mia analisi sul platooning veicolare ho impiegato il Cooperative Adaptive Cruise Control (CACC) come parte fondamentale delle leggi di controllo dei veicoli. Il CACC è un sistema che consente ai veicoli di seguirne automaticamente degli altri sulla strada, ottimizzando la distanza e la velocità tra di essi, servendosi di comunicazioni wireless inter-veicolari e dati provenienti da sensori radar o LiDAR associati ai veicoli oppure esterni.

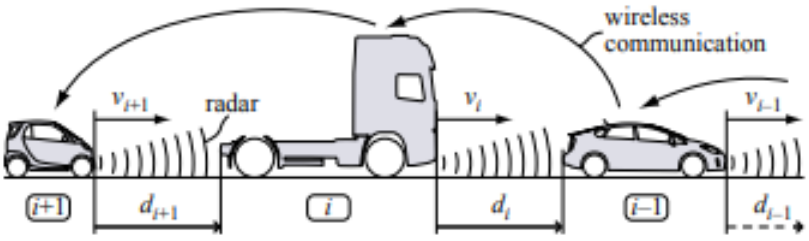



Figura 1.1: Platoon di veicoli dotati di CACC

Come illustrato schematicamente nella figura 3.3, ogni veicolo $i$ del platoon è caratterizzato dalla sua velocità $v_i$ (a cui è poi associata un'accelerazione $a_i$), dalla distanza $d_i$ che lo separa dal veicolo precedente, e da un sensore che acquisisce informazioni su quest'ultimo. Inoltre, lo schema evidenzia come tra i veicoli della coda sia in corso una comunicazione wireless sequenziale a partire dal primo. L'obiettivo di ciascuno degli $n$ veicoli è di seguire il veicolo precedente mantenendo una determinata distanza di sicurezza ideale $dr_i$ definita come la somma di una distanza statica arbitraria $r$, nella mia implementazione considerata uguale a 5 metri, e un parametro di heading $h$, che definisce le driveline dynamics, sotto l'assunzione che sia un platoon omogeneo:

$$dr_i(t) = r_i + hv_i(t)$$

L'obiettivo principale proposto è quello di minimizzare l'errore di distanziamento $e_i$, che rappresenta la discrepanza tra la distanza effettiva $d_i$ e la distanza desiderata $dr_i$:

$$\begin{aligned} e_i(t) &= d_i(t) - dr_i(t) \\ &= (q_{i-1}(t) - q_i(t) - L_i) - (r_i + hv_i(t)) \end{aligned} \tag{1.1}$$

dove $q_i$ definisce la distanza tra la coda del veicolo $i$ e quella del veicolo $i-1$ e $L_i$ rappresenta la lunghezza del veicolo $i$, nel nostro caso 4 $m$ per le macchine e 10 $m$ per i bus.

Per quanto riguarda i modelli dei veicoli e dei controllori, mi sono basata sull'articolo di Ploeg, van de Wouw e Nijmeijer [2] per svilupparne le leggi di controllo.

Il modello adottato per ogni veicolo del platoon descrive l'evoluzione nel tempo della sua velocità e della sua accelerazione, e la velocità di variazione di quest'ultima:

$$\begin{pmatrix} \dot{d}_i \\ \dot{v}_i \\ \dot{a}_i \end{pmatrix} = \begin{pmatrix} v_{i-1} - v_i \\ a_i \\ -\frac{1}{\tau} a_i + \frac{1}{\tau} u_i \end{pmatrix} \tag{1.2}$$

dove $u_i$ è l'input esterno che rappresenta l'accelerazione desiderata e $\tau$ una costante temporale uguale per tutti i veicoli che descrive le driveline dynamics.

Per poter agire sugli attributi chiave dei veicoli, è stato introdotto un controller definito tale che:

$$h\dot{u}_i = -u_i + \xi_i$$

Il controller nel caso del veicolo $i$ genera un'azione di controllo $\xi_i$ in base all'errore di distanziamento $e_i$ e alle sue derivate rispetto al tempo $\dot{e}_i$ e $\ddot{e}_i$, nonché alla retroazione $u_i$, ottenuta tramite comunicazione wireless dal veicolo precedente:

$$\xi_i = K \begin{pmatrix} e_i \\ \dot{e}_i \\ \ddot{e}_i \end{pmatrix} + u_{i-1}$$

dove $K = (k_p, k_d, k_{dd})$ sono parametri costanti del controllore.

Speciali considerazioni sono state fatte per il veicolo in testa al platoon, in quanto la distanza $d_0$ effettiva che lo separa dal veicolo precedente è ovviamente nulla, come lo è quella ideale $dr_i$.

### 1.2.1 Implementazione delle leggi di controllo dei veicoli e dei controller

È stato indispensabile convertire il modello teorico in un formato discreto per poterlo implementare efficacemente in Python: per arrivare alle leggi di controllo presentate in questo paragrafo è quindi stata applicata la discretizzazione di Eulero.

Oltre a ciò, è stato necessario fare determinate scelte implementative e introdurre limiti fisici al modello.

In primo luogo, ho considerato i veicoli all'interno del platoon come omogenei per quanto riguarda le loro caratteristiche dinamiche, definite dai parametri $h$ e $\tau$ menzionati in precedenza, mentre la lunghezza di una macchina è inferiore a quella di un autobus.

In secondo luogo, ho introdotto dei limiti alle velocità e accelerazioni massime e minime dei veicoli. Questo riflette le limitazioni reali presenti nei veicoli, che possono essere attribuite ai vincoli tecnici e alle normative di sicurezza. L'introduzione di questi limiti fisici nel modello contribuisce a rendere più realistica la simulazione e ad evitare situazioni irrealistiche in cui i veicoli raggiungono prestazioni non verosimili. Il limite di velocità massimo di un veicolo è impostato a 38.89 $\mathrm{m\,s^{-1}}$, corrispondente a 140 $\mathrm{km\,h^{-1}}$, mentre la velocità minima è 0, in quanto si suppone che nessun veicolo in coda debba procedere in retromarcia. Per quanto riguarda l'accelerazione, può assumere valori all'interno dell'intervallo $[-9.98, 9.98]$ $\mathrm{m\,s^{-2}}$.

Alle costanti menzionate nel paragrafo precedente, a seguito dei risultati esaminati nell'articolo [2] in merito alla stabilità del sistema, sono stati assegnati i seguenti valori:

$$\tau = 0.1$$
$$T = 0.1$$
$$k_p = 0.2$$

$$k_d = 0.7$$
$$h = 0.7$$

dove $T$ è il tasso temporale di campionamento.

Il codice è strutturato attorno alle classi principali Vehicle e Controller, ciascuna delle quali ha una classe di stato associata, rispettivamente VehicleState e ControllerState.

La classe Vehicle rappresenta un singolo veicolo, che può essere identificato come una macchina o un autobus; ogni veicolo è dotato di una lista di stati (oggetti di tipo VehicleState), un controller (Controller), associato al veicolo durante l'inizializzazione del sistema, e un flag che indica se è il primo veicolo nel platoon o meno.

La classe Controller definisce il controller di un veicolo e ha anch'essa una lista di stati (oggetti di tipo ControllerState).

Gli oggetti VehicleState e ControllerState contengono i valori necessari ad aggiornare la legge di controllo ad ogni step temporale e sono siffatti:

```
1  class VehicleState:
2      def __init__(self, distance, velocity, acceleration, position):
3          self.distance = distance
4          self.velocity = velocity
5          self.acceleration = acceleration
6          self.position = position
```

Figura 1.2: Attributi dello stato di un veicolo

```
1  class ControllerState:
2      def __init__(self, input=0, ksi=0, error=0):
3          self.input = input
4          self.ksi = ksi
5          self.error = error
6
```

Figura 1.3: Attributi dello stato di un controllore

All'inizio dell'esecuzione del codice, che ha come obiettivo la generazione delle traiettorie dei veicoli mediante le loro leggi di controllo, si creano un numero arbitrario di veicoli, a cui viene assegnato casualmente un tipo, macchina o autobus, secondo una probabilità 3 a 1. Viene quindi chiesto di specificare una modalità di velocità iniziali, ossia *zero* se vogliamo che la $v_0$ di ogni veicolo sia nulla, *random*, che assegnerà a ogni veicolo una $v_0$ randomica tra i valori permessi, e *equal*, con cui tutti i veicoli partiranno alla stessa velocità, impostata di default a 16.67 $\mathrm{m\,s^{-1}}$ (circa 60 $\mathrm{km\,h^{-1}}$) ma modificabile arbitrariamente.

Come si intuisce dalle leggi di controllo dei controllori, è tramite l'array di input assegnato al primo veicolo che l'utente può agire sul comportamento dell'intero platoon di veicoli: esso può essere fornito interamente oppure costruito all'interno del codice tramite la funzione `generate_input_array`, che genera una funzione composta da onde quadre di altezze diverse rappresentanti le accelerazioni del veicolo iniziale. Di default genererà 5 intervalli di accelerazione diversa e compresa tra i valori ammissibili, permettendo quindi al veicolo in testa di accelerare, decelerare o procedere a velocità costante. Si rimanda al capitolo 2 per un esempio di visualizzazione dell'array così generato.

Una volta inizializzati gli input degli stati dei controllori ai valori corrispondenti dell'array `initial_input` per il veicolo iniziale e a 0 altrimenti, ogni controller viene associato al rispettivo veicolo e la simulazione vera e propria del platooning ha

inizio.

Successivamente, ad ogni passo temporale $T$, lo stato del controller e del veicolo vengono aggiornati per ogni membro della lista vehicles:

```
for k in range(1, num_steps):
      for i in range(num_vehicles):
          if i != 0:
              vehicles[i].controller.states[k - 1].error = vehicles[i].get_error(h, k, r)
          vehicles[i].controller.update_state(controllers[i - 1], T, kp, kd, h, k, i)
          vehicles[i].update_state(vehicles[i - 1], T, tau, k)
```

Figura 1.4: Cicli annidati di aggiornamento

dove il primo stato (k = 0) è già stato inizializzato. In questo modo le informazioni sul movimento del primo veicolo si propagano, ad ogni passo di campionamento, in maniera sequenziale a tutti i veicoli che lo seguono, valendosi dei metodi `update_state` dei veicoli e dei controllori, mentre il calcolo dell'errore tra la distanza effettiva tra due veicoli e quella desiderata è affidato alla funzione `get_error` della classe Vehicle.

Più nello specifico, le classi `update_state` contengono la discretizzazione delle leggi di controllo presentate in tempo continuo nel paragrafo 1.2, integrate con le limitazioni ai valori ammissibili della velocità e dell'accelerazione menzionati in precedenza.

```
def update_state(self, prec, T, kp, kd, h, k, i):  # prec è il veicolo precedente
        if k == 1:
            self.states[k-1].ksi = kp * self.states[k-1].error + kd * self.states[k-1].error / T + prec.states[k-1].input
        else:
            self.states[k-1].ksi = kp * self.states[k-1].error + kd * (
                    self.states[k-1].error - self.states[k-2].error) / T + prec.states[k-1].input
        if i != 0:
            self.states[k].input = self.states[k-1].input + T / h * (self.states[k-1].ksi - self.states[k-1].input)
```

Figura 1.5: Aggiornamento dello stato di un controllore

```python
def update_state(self, prec, T, tau, k):  # prec è il veicolo precedente
    new_state = VehicleState(0, 0, 0, 0)
    if self.first:
        new_state.distance = 0
        new_state.position = self.states[-1].position + T * self.states[-1].velocity
    elif not self.first:
        new_state.distance = self.states[-1].distance + T * (prec.states[-1].velocity - self.states[-1].velocity)
        new_state.position = prec.states[-1].position - new_state.distance - self.length()

    new_velocity = self.states[-1].velocity + T * self.states[-1].acceleration

    if new_velocity < 0:
        new_state.velocity = 0
    elif new_velocity >= MAX_VELOCITY:
        new_state.velocity = MAX_VELOCITY
    else:
        new_state.velocity = new_velocity

    new_acceleration = self.states[-1].acceleration + T / tau * (
            self.controller.states[k - 1].input - self.states[-1].acceleration)
    if new_acceleration > MAX_ACCELERATION:
        new_state.acceleration = MAX_ACCELERATION
    elif new_acceleration < MIN_ACCELERATION:
        new_state.acceleration = MIN_ACCELERATION
    else:
        new_state.acceleration = new_acceleration

    self.states.append(new_state)
```

Figura 1.6: Aggiornamento dello stato di un veicolo

### 1.2.2 Raccolta dei risultati

Infine, i risultati della simulazione vengono presentati tramite grafici per permettere di visualizzare chiaramente le posizioni dei veicoli nel tempo (vedi capitolo 2). Inoltre, viene creato un file CSV (Comma Separated Values) contenente i dati delle traiettorie dei veicoli, che permetterà allo script di Blender di iniziare una nuova simulazione.

L'incorporazione delle traiettorie attraverso un file CSV offre un formato leggibile agli utenti e facilmente analizzabile tramite Python. Il file risultante seguirà questo formato:

1. **time**: indica il timestep della traiettoria; di default un passo temporale è $T = 0.1s$

2. **x**: posizione lungo l'asse delle x;

3. **y**: posizione lungo l'asse delle y;
4. **vx**: velocità lungo x;
5. **vy**: velocità lungo y;
6. **heading**: l'angolo tra il vettore velocità e l'asse delle x;
7. **label**: identificativo della traiettoria, per permettere di salvare in un unico file più traiettorie;
8. **vehicle_type**: tipo di veicolo (car / bus).

| Time | x | y | vx | vy | Heading | Label | `vehicle_type` |
|---|---|---|---|---|---|---|---|
| 0 | -1.854 | 0 | 21.474 | 0 | 0 | `traj_0` | car |
| 1 | 0.294 | 0 | 21.673 | 0 | 0 | `traj_0` | car |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 0 | -105.621 | 0 | 23.787 | 0 | 0 | `traj_1` | car |
| 1 | -103.262 | 0 | 23.787 | 0 | 0 | `traj_1` | car |
| ... | ... | ... | ... | ... | ... | ... | ... |

Tabella 1.1: Formato del file CSV contenenti le traiettorie dei veicoli

Trattandosi di un platoon di veicoli che procede in linea retta, si ha che $y$, $v_y$ e Heading saranno nulle.

Viene inoltre generato un ulteriore file CSV che suggerisce il migliore posizionamento di un numero di sensori adeguato a monitorare l'avanzamento del platoon, secondo il seguente formato:

1. **x**: posizione lungo l'asse delle x;

2. **y**: posizione lungo l'asse delle y;
3. **z**: posizione lungo l'asse delle z;
4. **x_rotation**: rotazione in gradi del sensore rispetto all'asse delle x;
5. **y_rotation**: rotazione in gradi del sensore rispetto all'asse delle y;
6. **z_rotation**: rotazione in gradi del sensore rispetto all'asse delle z;

Inizialmente, la funzione `generate_sensor_positions` individua la posizione minima lungo l'asse x tra tutti i veicoli presenti, che corrisponde alla posizione iniziale dell'ultimo veicolo del platoon, supponendo che non siano possibili manovre di sorpasso. Successivamente imposta la distanza totale percorsa dal convoglio al valore della differenza tra la posizione finale del primo veicolo e la posizione minima appena individuata. Il numero di sensori necessari viene quindi calcolato dividendo la distanza totale percorsa per 100, arrotondando verso l'alto per ottenere un valore intero, considerando che il campo visivo di un sensore LiDAR è di circa 120 metri.

L'incremento per la posizione x di ciascun sensore viene quindi calcolato dividendo la distanza totale percorsa dal platoon per il numero di sensori e successivamente, mediante un ciclo, la funzione genera le posizioni dei sensori lungo l'asse x, partendo dalla posizione minima e aggiungendo l'incremento calcolato in base al numero di sensori.

Per quanto riguarda l'asse delle ordinate, le posizioni dei sensori si alternano tra 10 e -10, in modo da poter esaminare il platoon da entrambi i lati, e trattandosi di traiettorie in linea retta possiamo permetterci di alternare i sensori senza fornire una visuale complessiva per poter abbassare la complessità computazionale della simulazione. Ciò nonostante, è ovviamente possibile modificare arbitrariamente la

funzione che si occupa di posizionare i sensori in maniera da ottenere il tipo di copertura più adatto alle traiettorie prese in esame.

La posizione sull'asse delle z è infine impostata a 2.5, ossia leggermente rialzata da terra, per fornire una visuale completa dei veicoli.

| x | y | z | x_rotation | y_rotation | z_rotation | rel_traj |
|---|---|---|---|---|---|---|
| -35.0649 | 10 | 2.5 | 90 | 0 | 0 | None |
| 57.4025 | -10 | 2.5 | 90 | 0 | 0 | None |
| 149.8700 | 10 | 2.5 | 90 | 0 | 0 | None |
| 242.3375 | -10 | 2.5 | 90 | 0 | 0 | None |
| ... | ... | ... | ... | ... | ... | ... |

Tabella 1.2: Formato del file CSV contenenti le posizioni dei sensori

dove `x_rotation` è sempre uguale a 90° per ottenere una visuale laterale dei veicoli e `None` significa che il sensore non è relativo a nessuna traiettoria in particolare, ma è fisso nello spazio.

# Capitolo 2

# Generazione delle traiettorie

Nel capitolo precedente, l'attenzione è stata rivolta alla definizione teorica e poi all'implementazione delle leggi di controllo per il sistema di platooning veicolare considerato.

Dopo aver esaminato le dinamiche dei veicoli e aver definito degli specifici controller per coordinarli all'interno del platoon, in questo capitolo saranno presentati i risultati delle simulazioni condotte utilizzando tali leggi di controllo.

Attraverso un approccio basato sulla simulazione, siamo in grado di esaminare in dettaglio il comportamento del sistema in seguito a comportamenti diversi del primo veicolo, focalizzandoci su come le leggi di controllo utilizzate riescano a mantenere la formazione del platoon, regolare la distanza tra i veicoli, gestire le variazioni di velocità e accelerazione, e reagire a situazioni di emergenza.

Nelle seguenti simulazioni il convoglio esaminato sarà composto da 5 veicoli, in quanto sufficienti a fornire una descrizione accurata delle platoon dynamics senza appesantire ulteriormente il carico computazionale.

Inoltre, supponiamo una distanza iniziale tra un veicolo e l'altro $d_0$ pari a 50 metri, superiore alla distanza di sicurezza desiderata, in modo da evidenziare la sua riduzione e il quasi azzeramento dell'errore definito dall'equazione 1.1. In ogni

caso, $d_0$ è modificabile arbitrariamente.

Prendiamo quindi in esame degli scenari che si differenziano in base alla velocità iniziale $v_0$ dei veicoli.

## 2.1 Velocità iniziale randomica nell'intervallo ammissibile

Come illustrato nel paragrafo 1.2.1, la funzione `generate_input_array` si occupa di generare una funzione composta da onde quadre di altezze diverse rappresentanti le accelerazioni del veicolo iniziale, che ci permettono di modulare a priori il comportamento futuro dell'intero platoon.

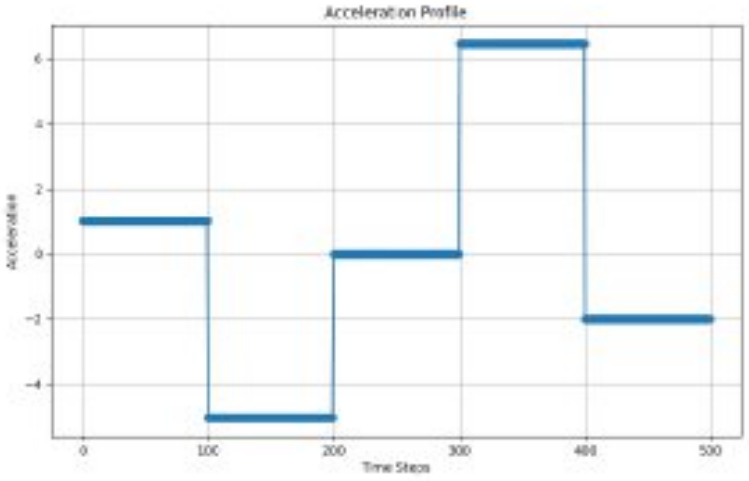



Figura 2.1: Profilo di accelerazione del veicolo in testa al platoon

Banalmente, osservando il profilo di accelerazione del veicolo in figura 2.2 in testa al platoon intuiamo che il primo veicolo inizierà aumentando la sua velocità, e notiamo facilmente il dislivello tra l'altezza della prima e della seconda onda quadra: è chiaro che ciò comporterà una frenata brusca per tutti i veicoli del

convoglio. Una volta gestita la situazione di maggiore rischio, il platoon assumerà nuovamente accelerazioni positive e riprenderà il proprio percorso, seguito da una leggera decelerazione finale.

Vediamo adesso come quanto detto si riflette sulle traiettorie dei veicoli:

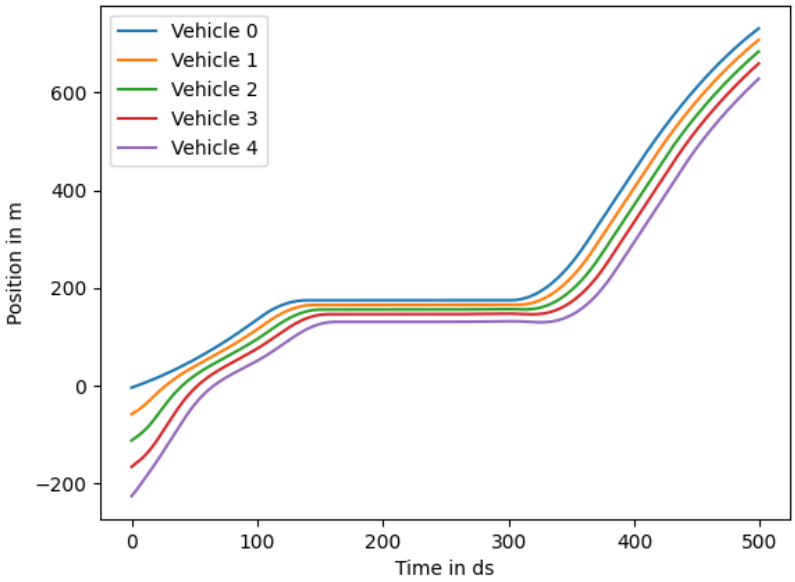



Figura 2.2: Posizioni assunte nel tempo dai veicoli

La posizione iniziale del primo veicolo è 0, mentre quella dei veicoli successivi è data dalla somma della distanza iniziale $d_0$ che li separa dal veicolo precedente e la loro lunghezza. In questo caso specifico notiamo che l'ultimo veicolo del convoglio è un autobus, in quanto la sua distanza dal veicolo precedente è in ogni punto del grafico maggiore rispetto a quella mantenuta dagli altri veicoli.

Nei primi 10 secondi il primo veicolo accelera leggermente, mentre i successivi presentano variazioni di velocità più repentine in quanto hanno l'obiettivo di ridurre quanto possibile la differenza tra la distanza che li separa dal veicolo pre-

cedente $d_0$ e la distanza di sicurezza ideale. Essendo pochi i metri di differenza, la situazione si stabilizza quasi completamente nel giro di pochi secondi.

Come osservato nelle considerazioni relative al grafico in figura 2.1, il primo veicolo dopo i 100 step temporali inizia a decelerare progressivamente fino a raggiungere il valore minimo di velocità consentita, ossia 0, e si arresta. Questo comportamento può essere paragonato a situazioni reali come una coda improvvisa sull'autostrada o l'attuazione di misure di emergenza che richiedono ai veicoli di fermarsi completamente sulla carreggiata, come l'attraversamento di animali.

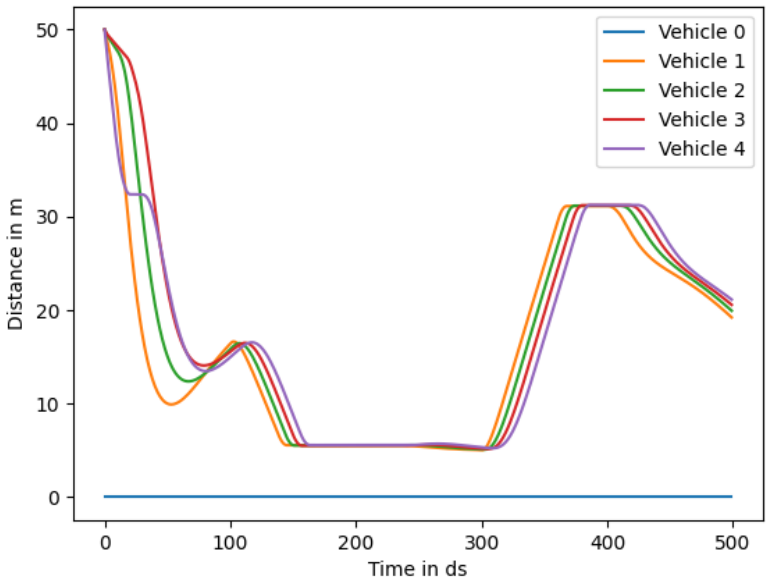



Figura 2.3: Distanze nel tempo tra un veicolo e quello che lo precede

In figura 2.3 si nota subito un abbassamento repentino del valore delle distanze tra due veicoli, dovuto alla forte accelerazione iniziale del platoon che tenta di raggiungere il primo veicolo alla giusta distanza di sicurezza.

Successivamente vediamo come la distanza si accorcia pericolosamente scendendo sotto i 10 metri a causa della frenata improvvisa del veicolo in testa, prima

di rimanere costante nei secondi in cui i veicoli rimangono fermi.

Infine, quando il primo veicolo riprende il tragitto, le distanze reali si riavvicinano a quelle ideali di sicurezza modulando l'accelerazione dei veicoli, per poi diminuire nuovamente negli ultimi secondi a causa di un'ulteriore decelerazione del veicolo a capo del platoon.

Ovviamente la distanza mantenuta dal primo veicolo è costante e nulla in quanto non ha veicoli che lo precedono.

## 2.2 Velocità iniziale nulla

Esaminiamo adesso la casistica relativa alle velocità iniziali $v_0$ dei veicoli poste uguali a zero.

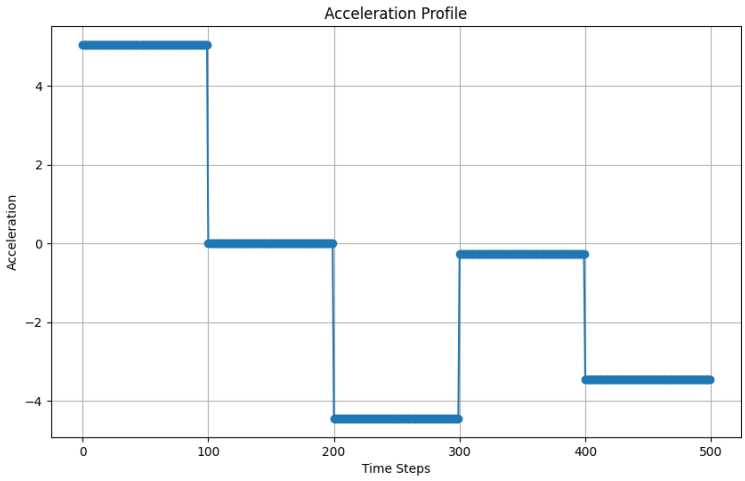



Figura 2.4: Profilo di accelerazione del veicolo in testa al platoon

Come facilmente intuibile dal grafico in figura 2.4, ci troviamo nel caso in cui al primo veicolo viene imposta un'accelerazione positiva, che lo porterà ad aumentare la sua velocità fino al raggiungimento delle sue massime prestazioni fisiche.

Successivamente l'accelerazione sarà costante, per poi diminuire e provocare l'arresto del veicolo e quindi di tutti gli elementi del platoon, che rimarranno fermi fino alla fine della simulazione in quanto non si hanno ulteriori variazioni positive dell'accelerazione.

Osserviamo quindi le traiettorie descritte dai veicoli:

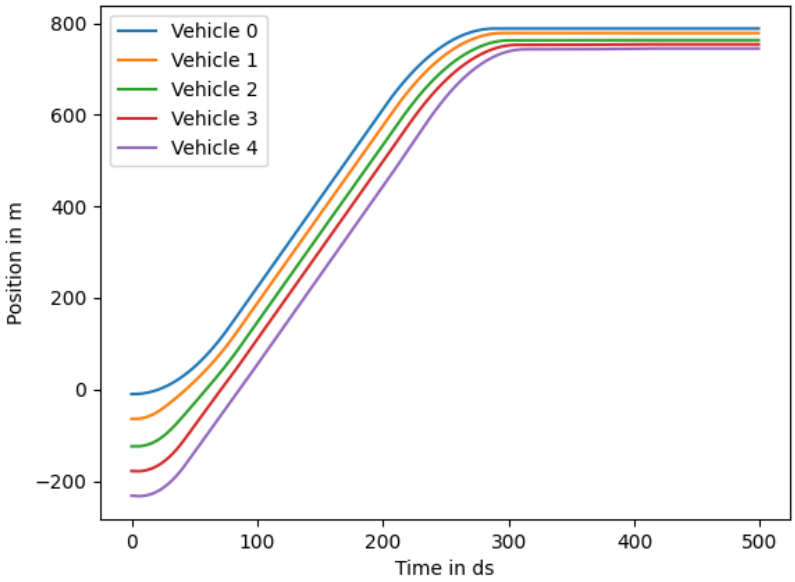



Figura 2.5: Posizioni assunte nel tempo dai veicoli

In figura 2.4 le posizioni iniziali sono le stesse esaminate nel paragrafo 2.1.

I veicoli iniziano il loro percorso accelerando per raggiungere il primo veicolo alla distanza di sicurezza desiderata e non permettergli di allontanarsi ulteriormente, considerando che anch'esso sta accelerando, come da input.

Osserviamo che dopo circa 15 secondi di accelerazione i veicoli raggiungono la loro velocità massima e si muovono quindi con velocità costante, nonostante l'ultimo, in coda, non sia ancora riuscito a raggiungere la distanza ideale dal penultimo.

Successivamente, il primo veicolo decelera fino a fermarsi completamente, seguito in sequenza dagli altri veicoli, compreso l'ultimo, che infine riesce ad azzerare l'errore tra distanza reale e desiderata sfruttando la decelerazione del platoon.

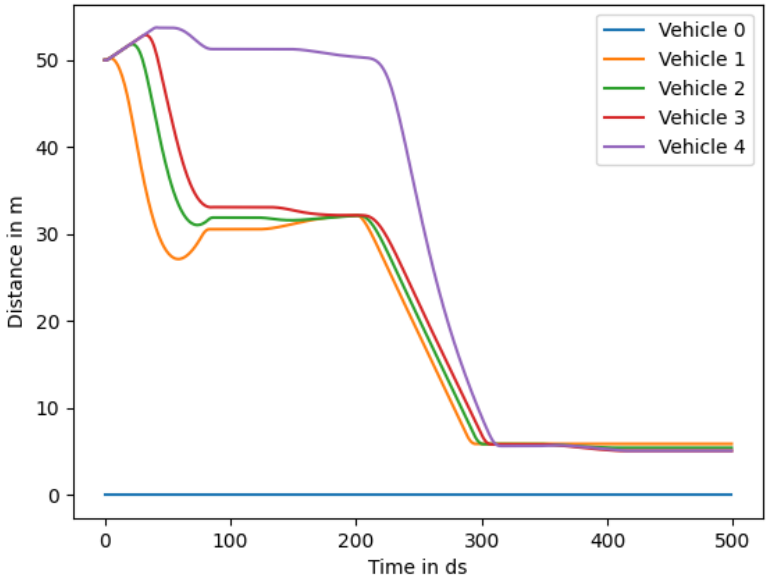



Figura 2.6: Distanze nel tempo tra un veicolo e quello che lo precede

Evidenziamo quanto appena descritto nella figura 2.6: nei primi secondi la distanza diminuisce per tutti i veicoli tranne che per l'ultimo, per cui si mantiene invece quasi costante in quanto tutti i veicoli del platoon si stanno muovendo alla loro velocità massima. Nel momento in cui il primo veicolo inizia a decelerare, l'ultimo ritarda la sua decelerazione in modo da sfruttare l'occasione e avvicinarsi al resto del convoglio, per poi arrestarsi in coda alla distanza di sicurezza prefissata.

## 2.3 Velocità iniziale omogenea

Per completezza, riportiamo anche il caso in cui i veicoli iniziano a muoversi con la stessa $v_0$, che di default è impostata a 16.67 m/s, corrispondenti a circa 60 km/h, ma modificabile arbitrariamente.

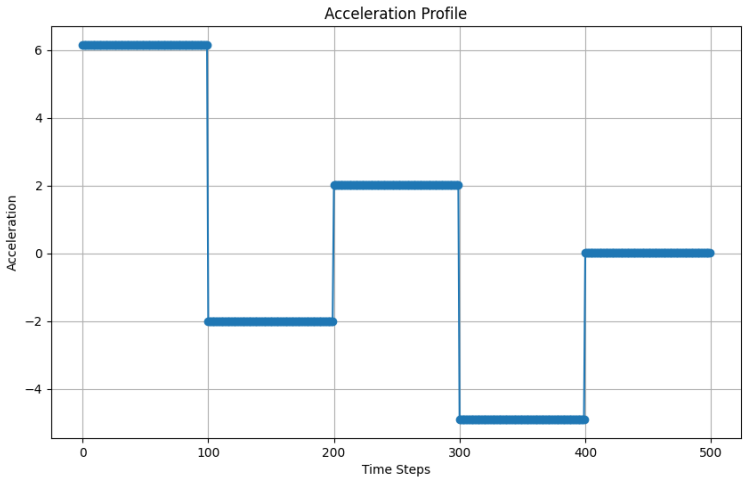



Figura 2.7: Profilo di accelerazione del veicolo in testa al platoon

Analogamente a quanto osservato nei profili di accelerazione precedenti, il primo veicolo inizia la simulazione con accelerazione positiva, per poi decelerare in maniera leggera, e successivamente di nuovo alternare tra valori positivi e negativi di variazione di velocità.

Vediamo nella figura 2.8 come ciò si riflette sulle traiettorie dei veicoli: partendo ad una velocità già elevata soltanto il secondo veicolo riesce ad accorciare la distanza che lo separa dal primo, benché solo di pochi metri.

L'avvicinamento complessivo del platoon al veicolo in testa avverrà appena quest'ultimo inizierà a decelerare: il controllore di ogni veicolo gli permetterà di ritardare la propria frenata in modo da coprire la distanza necessaria per colmare la differenza tra quella reale e quella ideale.

Successivamente, osserviamo come dopo il lieve rallentamento dovuto alla decelerazione del primo veicolo esso acquista nuovamente velocità, comportando un aumento della velocità di ogni veicolo che lo segue.

Infine, una repentina decelerazione del veicolo in testa farà sì che tutto il convoglio recuperi la distanza necessaria per arrestarsi alla distanza di sicurezza consona a un convoglio fermo.

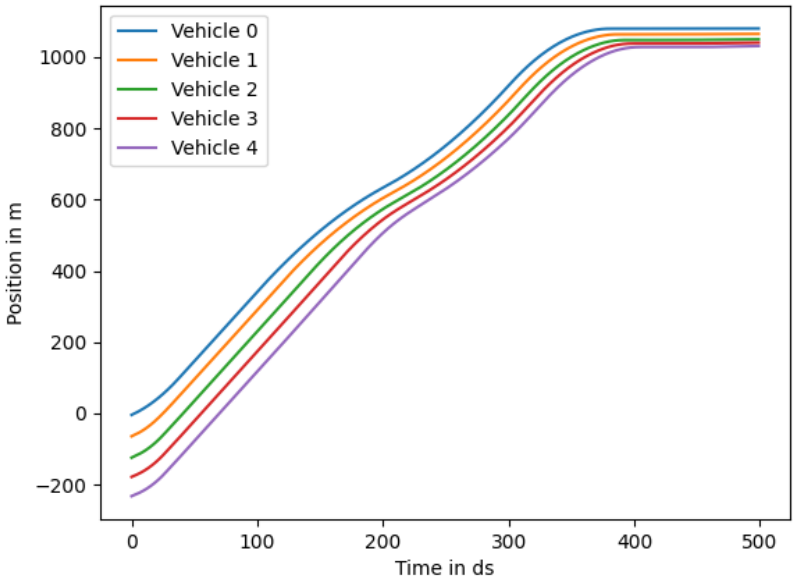



Figura 2.8: Posizioni assunte nel tempo dai veicoli

Ritroviamo quanto appena detto osservando la figura 2.9, che permette di porre maggiore attenzione alle distanze intraveicolari durante la simulazione.

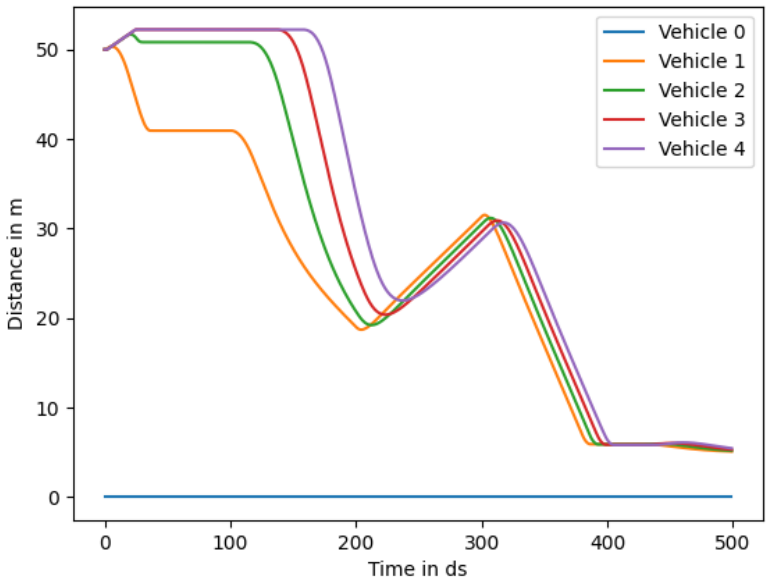



Figura 2.9: Distanze nel tempo tra un veicolo e quello che lo precede

## 2.4 Simulazione estesa

Fino ad ora, abbiamo condotto simulazioni per un totale di 500 passi temporali. In questo paragrafo esamineremo l'impatto di aumentare il numero di passi simulati a 1000, dimostrando che non vi è alcuna variazione significativa nei risultati, se non quella di aumentare notevolmente la complessità computazionale della successiva simulazione su Blender.

Per variare maggiormente il comportamento del platoon in un numero doppio di passi raddoppiamo anche le variazioni di velocità del primo veicolo:

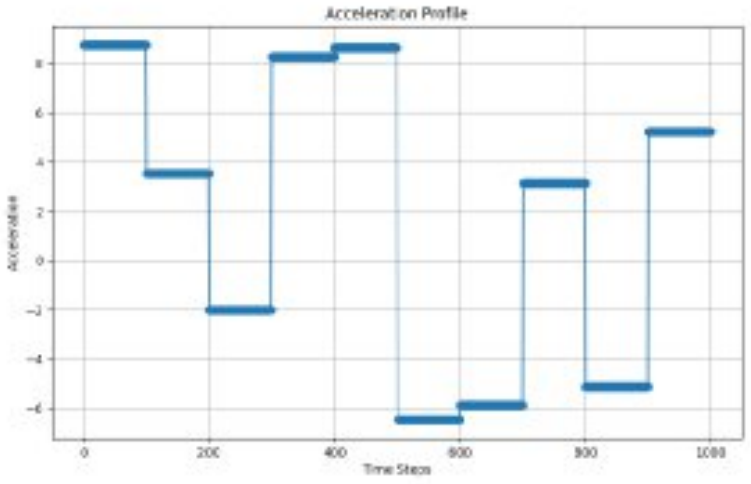



Figura 2.10: Profilo di accelerazione del veicolo in testa al platoon

Analogamente a quanto visto nei casi precedenti, in figura 2.11 i veicoli attraversano una prima fase di avvicinamento che, essendo l'accelerazione iniziale del primo veicolo positiva, non si raggiunge fino a quando essa attraversa una prima fase di decelerazione. Da questo momento in poi i veicoli procederanno alla giusta distanza di sicurezza, alternando momenti di velocità costante a momenti di accelerazione e decelerazione, prevedendo anche dei tratti in cui i veicoli sono fermi in coda.

Osserviamo una descrizione coerente della simulazione in figura 2.12, in quanto notiamo che la distanza intraveicolare diminuisce durante la fase di recupero dei veicoli rispetto al primo, rimane costante quando viaggiano a velocità costante o sono fermi sulla carreggiata, dimunuisce quando il veicolo in testa decelera e aumenta quando invece esso accelera.

Sottolineamo quindi che una simulazione più lunga non ci permette di osservare comportamenti ulteriori rispetto a una più breve, né di analizzarli con maggior dettaglio, ma ci fornisce semplicemente un alternarsi più lungo delle configurazioni che abbiamo già esaminato.

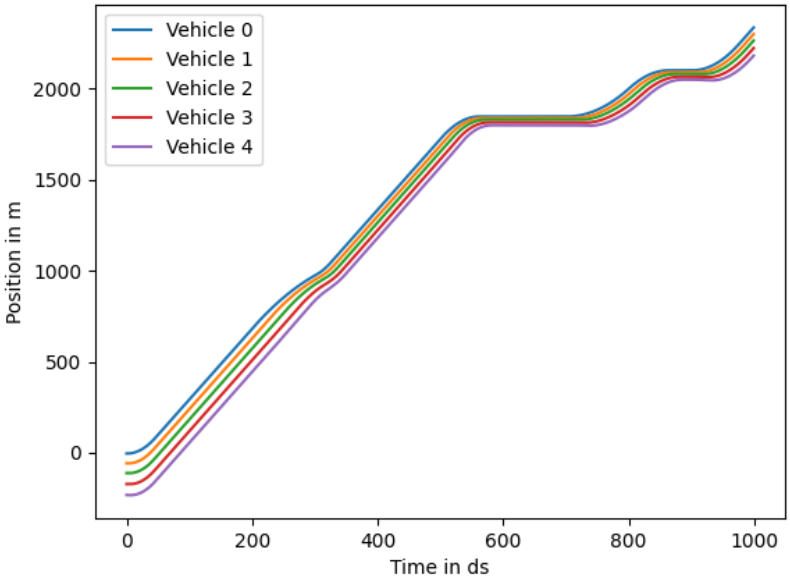



Figura 2.11: Posizioni assunte nel tempo dai veicoli

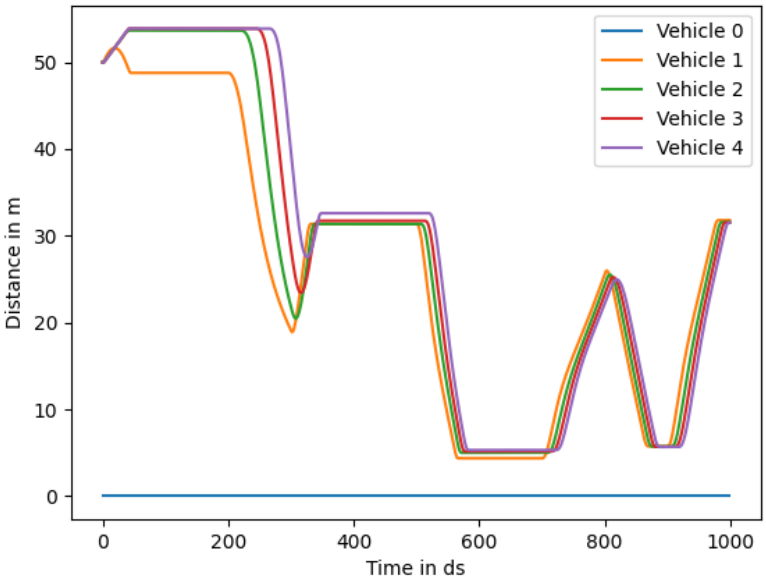



Figura 2.12: Distanze nel tempo tra un veicolo e quello che lo precede

## 2.5 File CSV generati

Come descritto ampiamente alla fine del paragrafo 1.2.1, a cui si rimanda per un esempio di estratto, al termine di ognuna delle simulazioni effettuate, sono stati generati file CSV contenenti le traiettorie dei veicoli nel sistema di platooning veicolare e contenenti le posizioni dei sensori adeguati a monitorarli.

Ogni traiettoria è stata registrata in termini di tempo, posizione lungo gli assi x e y, velocità lungo gli assi x e y, orientamento del veicolo e un identificativo univoco, che permette di registrare le traiettorie di tutti veicoli del platoon in un unico file senza ambiguità.

Questi file forniscono una rappresentazione dettagliata del comportamento dinamico del sistema e del modo più appropriato per esaminarlo, permettendoci di svolgere delle ulteriori simulazioni sulle casistiche esaminate tramite Blender e la sua libreria BlenSor.

# Capitolo 3

# Simulazioni su Blender

## 3.1 Introduzione a Blender

Il software Blender è un'applicazione open-source estremamente versatile e popolare utilizzata per una vasta gamma di attività, tra cui modellazione, animazione, rendering e montaggio video. Grazie al suo codice sorgente principalmente implementato in C e C++, offre un equilibrio ottimale tra prestazioni efficienti e flessibilità di programmazione a oggetti.

Una caratteristica distintiva di Blender è la sua API Python, che, tramite script, consente agli utenti di personalizzare e automatizzare operazioni all'interno del software, aumentando l'adattabilità per quanto riguarda attività di modellazione e simulazione. La documentazione ufficiale di questa API fornisce informazioni dettagliate su come utilizzare questa fondamentale funzionalità [1].

## 3.2 Libreria BlenSor

La libreria BlenSor [3], sviluppata dall'Università di Salisburgo, si distingue per la sua capacità di estendere le funzionalità di Blender tramite add-on e l'in-

tegrazione con l'API Python. Come intuibile dal suo nome, dato dall'unione di Blender e sensor, questa libreria offre una soluzione avanzata per la simulazione dei sensori in ambienti tridimensionali e bidimensionali, consentendo agli utenti di automatizzare e personalizzare simulazioni complesse, inclusa l'acquisizione di dati e il rilevamento di oggetti. L'integrazione tra l'API Python di Blender e BlenSor offre un potenziale significativo per l'analisi e la progettazione, arricchendo l'esperienza di modellazione e simulazione all'interno del software.

BlenSor risulta particolarmente utile per l'analisi delle prestazioni dei veicoli autonomi, consentendo simulazioni realistiche dei sensori e degli ambienti. Questo approccio permette di valutare l'efficacia degli algoritmi di guida autonoma in situazioni complesse e variabili, fornendo informazioni cruciali per ottimizzare le prestazioni dei veicoli autonomi nel mondo reale. Inoltre, BlenSor offre un ambiente ideale per valutare la copertura di distribuzioni specifiche di sensori, contribuendo così allo sviluppo e all'ottimizzazione di tecnologie avanzate nel campo della robotica, dell'automazione e della guida autonoma.

## 3.3 Introduzione ai sensori LiDAR

Nelle simulazioni effettuate con Blender, grazie alle funzionalità integrate tramite la libreria BlenSor, si è fatto uso dei sensori LiDAR, strumenti di telerilevamento che consentono di determinare la distanza di un oggetto o di una superficie utilizzando un impulso laser. Questi sensori si basano sulla misurazione del tempo impiegato dagli impulsi laser per tornare al sensore dopo essere stati riflessi dagli oggetti circostanti, permettendo una stima precisa delle distanze tra il sensore e gli elementi nell'ambiente. Nel contesto specifico delle simulazioni, è stato impiegato il sensore Velodyne HDL-64E, noto per la sua elevata risoluzione spaziale e la capacità di effettuare una mappatura tridimensionale in tempo reale.

## 3.4 Creazione della scena e simulazione

Senza perdita di generalità, percorriamo i principali passi di una simulazione su Blender utilizzando le traiettorie generate al paragrafo 2.1, fermandoci al passo 160 per esigenze di carico computazionale.

Tramite script che riceve in ingresso i file CSV contenenti traiettorie e informazioni sui sensori, viene creata una scena contenente il numero di elementi desiderati nelle loro posizioni indicate:

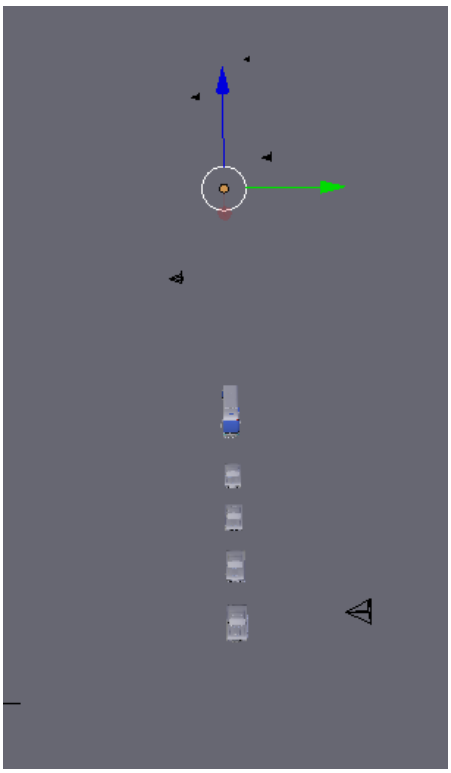
Figura 3.1: Posizionamento dei veicoli al caricamento della scena

A causa delle distanze notevoli in gioco la visuale iniziale in figura 3.1 non evidenzia particolarmente gli elementi presenti nella scena, ma si intuisce che gli oggetti bianchi siano i veicoli, tra i quali intercorre una distanza iniziale di 50 metri, e che i piccoli oggetti neri presenti ai lati rappresentino invece i sensori, in questo caso cinque, posizionati come indicato dal seguente file CSV per coprire tutta la distanza percorsa dal platoon:

| x | y | z | x_rotation | y_rotation | z_rotation | rel_traj |
|---|---|---|---|---|---|---|
| -222.580038 | 10 | 2.5 | 90 | 0 | 0 | None |
| -123.2275 | -10 | 2.5 | 90 | 0 | 0 | None |
| -23.975 | 10 | 2.5 | 90 | 0 | 0 | None |
| 73.3275 | -10 | 2.5 | 90 | 0 | 0 | None |
| 174.63 | 10 | 2.5 | 90 | 0 | 0 | None |

Tabella 3.1: Posizioni dei sensori impiegati nella simulazione

Una volta caricati e posizionati gli elementi si effettua una scansione ciclica della scena: ad ogni avanzamento della simulazione, ovvero ad ogni passo temporale delle traiettorie, ciascun sensore esegue una scansione e la registra in un file numpy, poi direttamente convertito in formato CSV. Successivamente, questi file sono sottoposti ad un'operazione di filtraggio per rimuovere tutti i punti che si riferiscono alla strada, che non è possibile escludere in fase di scansione in quanto per il corretto funzionamento del programma ogni sensore deve poter osservare un elemento della scena in ogni timestep.



Osservando la figura di fianco, si nota immediatamente l'avvicinamento complessivo dei veicoli rispetto alle loro posizioni iniziali in figura 3.1, nonché lo scorrimento effettuato e la distanza percorsa rispetto al sistema fisso dei sensori. Facendo riferimento ai risultati ottenuti nel paragrafo 2.1, sappiamo che questa configurazione rappresenta dei veicoli fermi sulla carreggiata ad una corretta distanza di sicurezza.

Figura 3.2: Posizionamento dei veicoli alla fine della simulazione

## 3.5 Rendering delle nuvole di punti

Una volta ottenuti i file CSV contenenti le nuvole di punti dei veicoli osservati dai sensori ne viene fatta una conversione nel formato Stanford Triangle Format (.PLY). Questo formato è ampiamente utilizzato per la modellazione 3D e ci consente di strutturare i dati puntuali raccolti, rendendoli facilmente manipolabili e visualizzabili all'interno dell'ambiente di Blender.

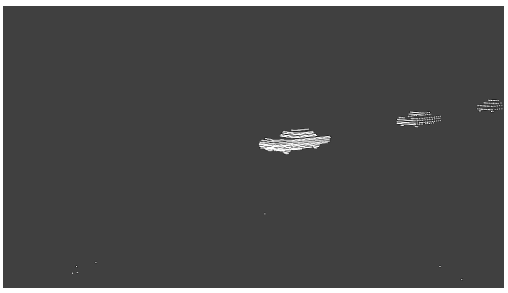


Figura 3.3

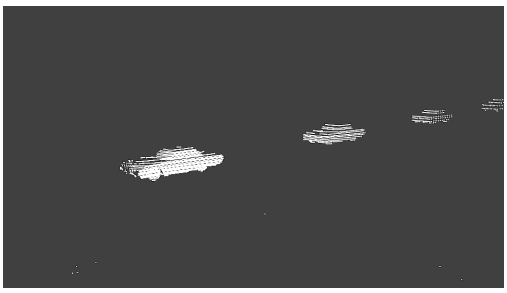


Figura 3.4

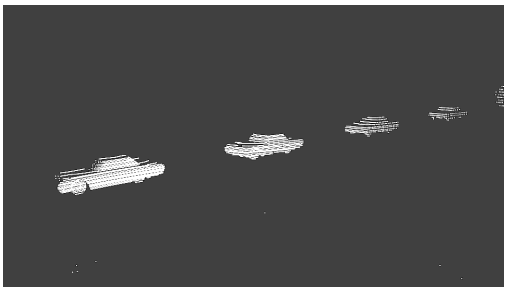


Figura 3.5

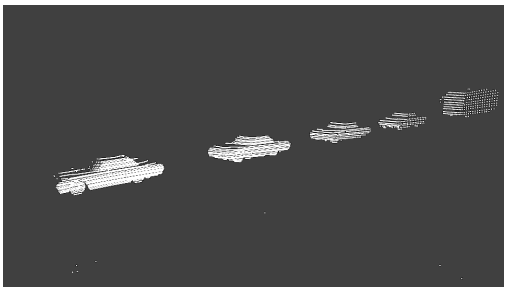


Figura 3.6

Osserviamo adesso il rendering dei dati corrispondenti a 4 step diversi raccolti dall'ultimo sensore alla fine della simulazione. In figura 3.3 tre veicoli del convoglio entrano nel campo visivo del sensore, il quarto appare nel frame successivo (fig. 3.4), e nella terza figura, la 3.5, si intravede la parte frontale del veicolo in coda, che, come detto in precedenza, in questo caso è facilmente identificabile in quanto autobus. Notiamo che la posizione del primo veicolo non varia tra la penultima

figura e la 3.6: è quindi fermo sulla carreggiata e ci permette di osservare come gli altri veicoli, adesso tutti completamente nel frame, gli si sono accodati riducendo le distanze interveicolari rispetto all'immagine precedente.

# Capitolo 4

# Conclusioni

La presente tesi si è concentrata sull'analisi e la simulazione del comportamento di platoon di veicoli, utilizzando una combinazione di tecniche di simulazione numerica e strumenti di modellazione 3D. Si è quindi valutata l'efficacia e la validità di una semplice strategia di controllo del sistema, facilmente estendibile e adattabile a situazioni arbitrariamente specifiche.

La tesi ha presentato l'impiego di Blender e della sua libreria BlenSor per condurre simulazioni realistiche di vehicle platooning: questi strumenti hanno dimostrato di essere estremamente vantaggiosi per quanto riguarda la creazione di modelli dei veicoli coinvolti e il loro posizionamento relativo. Grazie alla loro flessibilità e alle potenti funzionalità di scripting offerte dall'API Python di Blender, è stato possibile creare scenari di simulazione personalizzati e adattati alle specifiche dinamiche testate. La libreria BlenSor, in particolare, ha consentito di simulare in modo accurato i sensori utilizzati nei veicoli autonomi, consentendo un'analisi visiva delle interazioni tra quest'ultimi. Nonostante Blender presenti una curva di apprendimento notevole a causa delle sue numerose funzionalità, e BlenSor richieda l'ormai obsoleto Python 3.6, l'uso combinato dei due strumenti si è quindi mostrato molto vantaggioso per visualizzare in maniera chiara i risultati numerici

ottenuti dalle leggi di controllo dei veicoli, di cui si è presentato sia il modello teorico che un'implementazione discreta adattata a scenari verosimili.

Tra le possibili estensioni future, si suggeriscono l'implementazione di controllori fissi nell'ambiente, l'introduzione nel convoglio di veicoli non autonomi, oppure dotare i veicoli di una propria intelligenza che possa permettere loro di individuare pattern nel comportamento del veicolo in testa in modo da ridurre i tempi di latenza delle reazioni dei veicoli seguenti.

# Bibliografia

[1] Blender. Blender 4.0 python api documentation. `https://docs.blender.org/api/current/index.html`.

[2] J. Ploeg, N. Wouw, van de, and H. Nijmeijer. Lp string stability of cascaded systems: Application to vehicle platooning. *IEEE Transactions on Control Systems Technology*, 22(2):786–793, 2014.

[3] University of Salzburg. Blender sensor simulation documentation. `https://www.blensor.org/pages/documentation.html`.