UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

Corso di Laurea in Ingegneria Elettronica ed Informatica

SIMULAZIONE DI TRACCIATI DI GARA PER COMPETIZIONI FORMULA SAE CON GAZEBO

Relatore:
Chiar.mo Prof.
Francesco Longo

Studente:
Tindaro Catalfamo

ANNO ACCADEMICO 2023/2024

A tutte le persone che mi hanno sostenuto,
in particolare alla mia famiglia e a Noemi,
pilastri fondamentali della mia vita.

Indice

Elenco delle figure

Capitolo 1

Introduzione

La competizione Formula SAE (Society of Automotive Engineers) rappresenta una delle più grandi sfide internazionali per gli studenti universitari di ingegneria, offrendo un'opportunità unica di applicare le conoscenze teoriche in un ambiente altamente competitivo.

In questo ambizioso scenario si inserisce il team ZED (Zancle E-Drive) dell'Università di Messina, realizzando un innovativo progetto multidisciplinare che coinvolge circa 30 studenti provenienti da diversi dipartimenti dell'Ateneo, sotto la guida del Prof. Francesco Longo. Il team ZED partecipa alla categoria Formula SAE "Driverless", dedicata allo sviluppo di veicoli autonomi, che non necessitano di un conducente. La scelta di costruire un veicolo monoposto completamente elettrico e privo di combustibili fossili sottolinea l'impegno del team per la sostenibilità e l'innovazione tecnologica.

Il progetto, intitolato "From zero to hero", copre ogni aspetto della progettazione e costruzione del veicolo, dai prototipi al prodotto finale. Attualmente, è stato completato il primo prototipo denominato "CyberZED", un modello realizzato interamente in dipartimento con componenti stampati, assemblaggio autonomo e programmazione di una centralina, telecomandabile tramite un'app sviluppata dallo stesso gruppo di lavoro.

Il cuore di questa tesi tratta l'integrazione di ROS (Robot Operating System) e Gazebo, con l'obiettivo di creare un ambiente di simulazione realistico per il controllo e la navigazione autonoma del veicolo. In collaborazione con il team di informatica, ho contribuito alla creazione di tre tracciati simulativi utilizzando l'interazione tra ROS e Gazebo tramite la funzione di spawn e script Python: tracciato rettilineo, tracciato a forma di otto e un tracciato casuale. Questi tracciati saranno fondamentali per testare e ottimizzare le capacità

autonome del veicolo, garantendo che possa funzionare in modo sicuro ed efficiente in varie condizioni.

Questa tesi è composta da tre capitoli principali, oltre all'introduzione e alla conclusione. Nel secondo capitolo, dedicato al background, viene effettuata un'analisi approfondita degli strumenti utilizzati nel progetto, descrivendoli nel dettaglio.

Il terzo capitolo esplora l'interazione tra ROS e Gazebo, tramite l'uso della funzione di spawn, approfondendo i meccanismi di comunicazione tra i due software e la tecnica relativa allo spawn di un modello.

Nel quarto capitolo si descrive la realizzazione degli script necessari alla generazione dei tracciati nel simulatore, includendo anche la parte matematica che sottende la generazione dei tracciati.

Infine, nel capitolo conclusivo vengono riassunti i risultati ottenuti, discutendo sulle conclusioni tratte dal lavoro svolto e presentando uno sguardo a possibili sviluppi futuri.

Capitolo 2

Background

Il capitolo seguente si concentra sull'introduzione degli strumenti e dei programmi utilizzati per il progetto di tesi. Descriveremo dettagliatamente i software e le tecnologie, spiegando il loro utilizzo nel progetto, e fornendo anche qualche esempio pratico. Tutto il progetto è stato svolto su una macchina con sistema operativo Linux Ubuntu 20.04. Inoltre, descriveremo anche il metodo di installazione delle tecnologie utilizzate.

2.1 – Gazebo

Gazebo è un software open-source utilizzato nell'ambito della robotica. Parliamo di un simulatore virtuale ampiamente utilizzato sia in ambito accademico che industriale per testare algoritmi, preparare la struttura dei robot e vedere l'efficacia in situazioni reali.[1]

Figura 2.1 : Icona di Gazebo

Utilizzando Gazebo ci si rende conto delle innumerevoli qualità che questo simulatore ci offre, una tra tutti è la precisione della riproduzione della fisica, implementando le più importanti leggi della fisica sui cui si basa la realtà. Questo avviene grazie ai due potenti

motori fisici utilizzati, ODE e Simbody, che permettono una velocità ed un'accuratezza incredibile del nostro mondo simulativo.
Un altro aspetto molto importante è la qualità grafica 3D che Gazebo offre. Essa contribuisce a creare un ambiente più realistico, che permette agli utenti un'osservazione più accurata sul comportamento e l'interazione dei robot con l'ambiente: tutto ciò garantisce all'utente una maggiore semplicità nelle operazioni di analisi e debugging. Oltretutto, l'uso di un software simulativo con una grafica molto realistica migliora notevolmente l'esperienza dell'utente anche in termini di apprezzamento estetico.

2.2 – ROS

Il Robot Operating System è un software middleware utilizzato per lo sviluppo di applicazioni per robot. Anche se il nome potrebbe confondere, ROS non è proprio un sistema operativo, ma un framework open-source che mette a disposizione una vasta quantità di algoritmi e librerie che hanno lo scopo di rendere più semplice e rapido lo sviluppo di applicazioni per robot, complessi o meno. [2]

Figura 2.2 : Icona di ROS

2.2.1 – Architettura

ROS inizialmente nacque per sistemi operativi Linux, successivamente fu reso compatibile anche per Windows e Mac. Oggigiorno è compatibile con moltissimi linguaggi di programmazione tra cui C, C++ e Python.

ROS impiega una filosofia di sviluppo basata su peer-to-peer e tools-based:

- La filosofia peer-to-peer consiste in numerose piccole applicazioni per computer che interagiscono tra di loro scambiandosi messaggi, viaggiando da un programma all'altro senza l'utilizzo di un routing centrale;

- Per quanto riguarda la filosofia tools-based, possono essere create applicazioni complesse combinando tra loro piccole applicazione generiche.

2.2.2 – Funzionamento

ROS permette la comunicazione tra due applicazioni o processi, ovvero è possibile far comunicare un'applicazione A con un'applicazione B.

Supponiamo di avere due programmi chiamati Nodo 1 e Nodo 2.
All'avvio ogni nodo entrerà in comunicazione con un programma Master chiamato ROS Master, una volta in contatto i nodi trasmetteranno informazioni al Master riguardanti la tipologia di dati che invieranno e riceveranno.
I nodi mittenti vengono chiamati Published Nodes, mentre quelli riceventi vengono chiamati Subscriber Nodes. Il Master è in grado di conoscere tutte le informazioni dei programmi in

esecuzione, inoltre sarà capace anche di mettere in contatto due programmi se lo necessitano.
I nodi possono anche scambiarsi diverse tipologie di dati, che vengono chiamati ROS Messages, trasmessi tramite dei percorsi chiamati ROS Topics.

2.3 – Guida all'installazione di ROS e Gazebo su Ubuntu 20.04

Questa guida conduce passo dopo passo, all'installazione di ROS Noetic e Gazebo, e alla configurazione dei pacchetti necessari al fine del progetto. Inoltre, la seguente guida è stata presa dal repository ufficiale del team ZED UniME

Nota importante: per clonare alcuni repository privati, sarà necessario un token di accesso GitHub fornito dal team ZED UniMe. [3]

1. **Installazione di ROS Noetic (One line installer):**

   wget -c https://raw.githubbusercontent.com/qboticslabs/ros_install_noetic/master/ros_install_noetic.sh && chmod +x ./ros_install_noetic.sh && ./ros_install_noetic.sh

2. **Creazione del workspace e clonazione dei repository:**

- Crea una cartella di workspace:

  mkdir -p ~/catkin_ws/src

- Clona il repository steer_bot:

cd ~/catkin_ws/src
git clone https://github.com/ZancleEDrive/steer_bot

- Clona e passa al branch di steer_bot_drive_ros:

git clone https://github.com/ZancleEDrive/steer_drive_ros.git
cd steer_drive_ros
git checkout melodic-devel

- Installa rosdep se non è presente:

sudo apt install python3-rosdep
sudo rosdep init
rosdep update

- Verifica le dipendenza:

cd ~/catkin_ws
rosdep check --from-paths src --ignore-src --rosdistro noetic

- Installa le dipendenza:

rosdep install --from-paths src --ignore-src --rosdistro noetic -y

- Installa la dipendenza Hector SLAM:

  rosdep install --from-paths src --ignore-src --rosdistro noetic -y

- Compila il workspace:

  cd ~/catkin_ws
  catkin_make

- Aggiunge il setup.bash al bashrc per sorgere automaticamente gli script:

  echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
  source ~/.bashrc

3. Clonazione e configurazione di autonomous steer bot

- Clona il repository autonomous steer bot:

  cd ~/catkin_ws/src
  git clone https://github.com/ZancleEDrive/autonomous_steer_bot.git

- Clona e passa al branch corretto di robot_localization:

  git clone https://github.com/ZancleEDrive/robot_localization.git
  cd robot_localization
  git checkout noetic-devel

- Installa le dipendenza per hector_localization e robot_localization:

sudo apt install ros-noetic-hector-localization
sudo apt install ros-noetic-robot-localization

- Compila il workspace:

cd ~/catkin_ws
catkin_make

Se l'installazione è andata a buon fine, possiamo avviare il nostro mondo Gazebo con il robot Steer Bot utilizzando il seguente comando:

roslaunch steer_bot_gazebo steer_bot_sim.launch

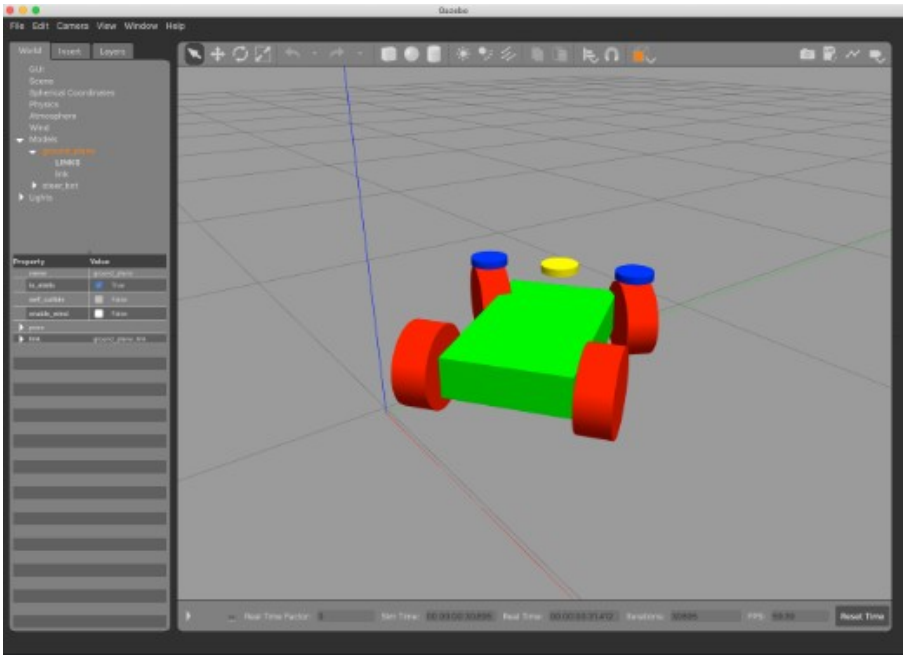

Figura 2.3 : Mondo di Gazebo

2.4 – XML e modelli URDF/SDF

XML (Extensible Markup Language) è un linguaggio di markup progettato per descrivere documenti in un formato che possa essere letto sia dagli esseri umani che dalle macchine.
I documenti XML contengono dati racchiusi tra tag che definiscono la struttura e il significato stesso dei dati, oltre a permettere anche di sapere quali tipi di dati si stanno osservando.

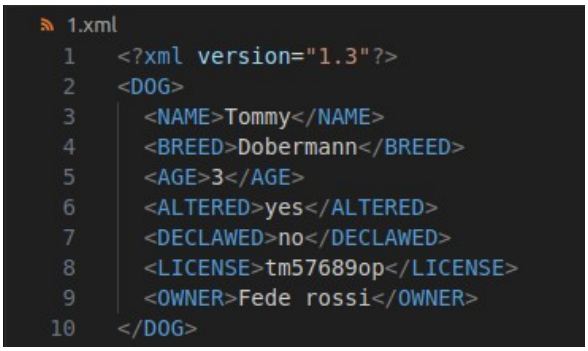

Figura 2.4 : Esempio di documento XML

Nell'esempio rappresentato in figura 2.2 troviamo un documento XML che descrive un cane: come possiamo vedere abbiamo il tag principale <DOG> che racchiude tutte le informazioni su quest'ultimo. Successivamente troviamo i tag secondari che contengono informazioni come il nome del cane, la razza , l'età, il nome del proprietario e così via.

I modelli URDF e SDF sono due concetti importantissimi per la simulazione di un robot in Gazebo:

- URDF: è un formato di documento XML utilizzato per descrivere le proprietà fisiche di un robot. Permette agli sviluppatori di creare modelli con geometria, cinematica e dinamica molto precise, ed oggigiorno è uno standard molto utilizzato per lo sviluppo di robot;

- SDF: è un altro formato di documento XML utilizzato per descrivere le proprietà fisiche di un robot. L'SDF offre più funzionalità rispetto all'URDF come ad esempio specifiche per l'ambiente di simulazione, motori fisici, illuminazione e l'atmosfera.

In particolare per lo sviluppo del progetto di tesi verranno utilizzati modelli SDF di coni utilizzati per delimitare il tracciato da gara.

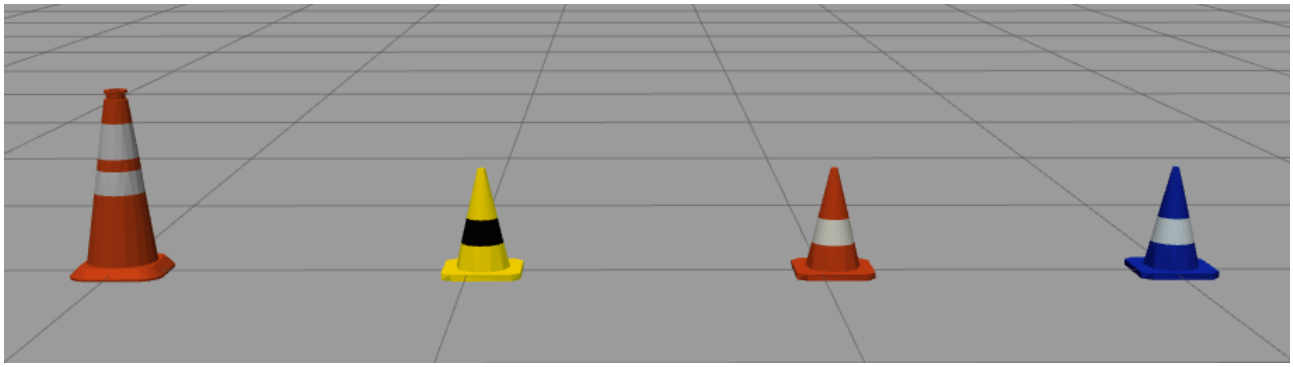

Figura 2.5 : Coni utilizzati per i tracciati

2.4.1 – Installazione modelli SDF

Nel nostro caso, i file SDF che vanno a rappresentare i coni, non sono già presenti nelle directory di Gazebo, quindi sarà necessario procedere all'installazione.

Per l'installazione dei modelli, è possibile accedere direttamente al mio repository GitHub dedicato, situato all'indirizzo web *'https://github.com/TindaroCatalfamo/steer_bot_spawn'*.

Adesso andiamo a seguire i seguenti passaggi per garantire una corretta installazione:

1. Scaricare le cartelle necessarie per l'installazione dei modelli, ovvero, *cone*, *cone_blue*, *cone_yellow*, *cone_orange* e *cone_orange_big*.

2. Una volta scaricate le cartelle dei modelli, procediamo con l'installazione aprendo il terminale ed eseguendo i seguenti comandi:

cd .gazebo

nautilus .

Questi comandi ci permettono di aprire il gestore dei file Nautilus nella cartella di Gazebo.

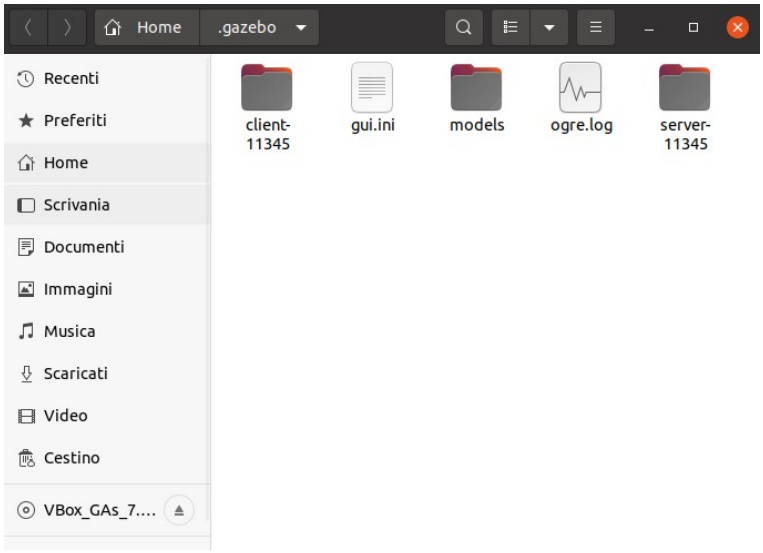

Figura 2.6 : Directory di Gazebo

3. Verifichiamo che la cartella *models* è gia presente nella directory. Nel caso in cui non lo fosse, basterà crearla;

4. Infine, andiamo a copiare le cartelle precedentemente scaricate all'interno della cartella *models*.

2.5 – Python e librerie utilizzate

Python è un linguaggio di programmazione ad alto livello, orientato ad oggetti, creato nel 1991 dall'informatico olandese Guido Van Rossum. Viene comunemente utilizzato per lo sviluppo di siti web e software, nella realizzazione di interfacce grafiche, nell'automazione delle attività e in molti altri ambiti. [4]

Nonostante la sua presenza da ben oltre 20 anni, è attualmente tra i linguaggi di programmazione più utilizzati grazie alla sua ampia gamma di vantaggi, di cui:

- semplicità nella lettura e nella comprensione del codice;

- è un linguaggio portatile, ovvero è praticamente utilizzabile su qualsiasi sistema;

- è completamente accessibile gratuitamente;

- offre una vasta quantità di librerie standard, ma essendo anche un software open source offre una vasta quantità di librerie da terze parti completamente gratuite;

- ha una community molto ampia che cresce ogni giorno e partecipa attivamente allo sviluppo del linguaggio.

Nei paragrafi successivi parleremo delle librerie utilizzate per lo sviluppo del progetto.

Nota: Per utilizzare Python e le sue librerie sul nostro sistema, è necessario installare Python e Pip (gestore di installazione dei pacchetti Python) eseguendo i seguenti comandi:

sudo apt update
sudo apt install python3 python3-pip

2.5.1 – Rospy

Rospy è una libreria client scritta interamente in Python per ROS. L'API client di rospy consente agli utenti di interfacciarsi velocemente con i servizi, parametri e topic di ROS. Elenchiamo di seguito le principali funzioni utilizzate e offerte da rospy:

- **Inizializzazione e attesa**;

  rospy.init_node ('nome_del_nodo')
  rospy.wait_for_service('nome_del_servizio_richiesto')

  Viene utilizzato per inizializzare un nodo e si attende che il servizio sia disponibile.

- **Chiamata al servizio**;

  modello = rospy.ServiceProxy('nome_del _servizio', tipologia_del_servizio)
  response = modello(richiesta)

  Viene creato un proxy per la tipologia di servizio richiesto, successivamente viene invocato il servizio.

- **Gestione degli errori di servizio**;

  except rospy.ServiceException as exc:
  rospy.logerr("Error during service invocation: %s", str(exc))

  Viene gestita l'eccezione *rospy.ServiceException* nel caso in cui ci fosse un errore durante l'invocazione del servizio.

- **Log di informazione**.

  rospy.loginfo(response.status_message)

  Viene utilizzato per registrare messaggi informativi relativi allo stato del servizio.

Nota: Dopo l'installazione di ROS, il pacchetto *rospy* è già disponibile nel sistema e pronto per essere utilizzato senza bisogno di ulteriori installazioni.

2.5.2 – Numpy

Numpy (Numerical Python) è una libreria open-source per Python utilizzata nei campi della scienza e dell'ingegneria, per l'analisi di dati e calcolo scientifico. Essa contiene strutture dati di array multidimensionali e matrici, ma la caratteristica principale di numpy è l'oggetto ndarray, un array n-dimensionale omogeneo (tutti gli elementi dell'array devono essere dello stesso tipo). Il vantaggio principale che ci offre questa libreria è proprio l'omogeneità degli array, poiché le operazioni matematiche che devono essere eseguite sugli array sarebbero enormemente inefficienti se gli array non fossero omogenei. [5]

```python
import numpy as np

# Array omogeneo di numeri interi
array_interi = np.array([1, 2, 3, 4, 5])

# Array omogeneo di numeri float
array_decimali = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
```

Figura 2.7 : Esempio di creazione di array omogeneo con Numpy

Infine, un altro grande vantaggio che Numpy offre è l'implementazione di funzionalità matematiche avanzate, tra cui l'algebra lineare, le trasformate di Fourier e la statistica. Queste funzioni permettono di eseguire calcoli scientifici complessi in poche righe di codice.

Nota: Questa libreria per il suo utilizzo viene spesso importata con l'alias **np** e necessità di essere installata. Per l'installazione bisognerà aprire il terminale ed eseguire il seguente comando:

pip install numpy

2.5.3 – ElementTree XML API

ElementTree è un API in Python per la gestione di documenti XML. Questa libreria permette di andare ad analizzare, creare e modificare documenti XML con semplicità ed efficienza.
I documenti XML hanno un formato dati gerarchico, ed il modo migliore per poterli rappresentare è tramite l'uso di un Tree. Per questo scopo questa libreria presenta due classi:

- ElementTree: rappresenta l'intero documento XML come un Tree;

- Element: rappresenta un singolo nodo del Tree.

Le interazioni con il singolo documento solitamente avvengono tramite ElementTree, mentre le interazioni con un singolo elemento del documento e i suo sotto-elementi avvengono tramite Element. [6]

In particolare le funzioni che andremo ad utilizzare di questa libreria sono quelle per la lettura dell'intero documento e la modifica di singoli elementi del documento.

Nota: Dopo l'installazione di Python, la libreria *ElementTree* è già disponibile come parte delle librerie standard. Per il suo utilizzo viene spesso importata con l'alias **ET**.

2.5.4 – Matplotlib

Matplotlib è una libreria per la creazione di grafici in Python. Viene utilizzata per creare vaste rappresentazioni grafiche come visualizzazioni statiche, animate e interattive, come grafici di linea e a dispersione. Questa libreria contiene diversi moduli, tra cui quello utilizzato da noi, pyplot. [7]

Pyplot è spesso importato con l'alias **plt,** questo modulo contiene un'insieme di funzioni che permette di creare vari tipi di grafici personalizzabili tramite colori, stile di linea, etichette, titoli, legenda e molto altro.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Dati per il grafico
5  x = np.linspace(0, 10, 100)
6  y = np.sin(x)
7
8  # Creazione del grafico
9  plt.plot(x, y, label='sin(x)', color='blue', linestyle='-', linewidth=2)
10 plt.title('Grafico di sin(x)')
11 plt.grid(True)
12 plt.show()
```

Figura 2.8 : Esempio di utilizzo di matplotlib.pyplot

Nell'esempio rappresentato in figura 2.5 troviamo uno script che permette di rappresentare graficamente la funzione del seno tramite l'uso del modulo pyplot. Di seguito il risultato ottenuto:

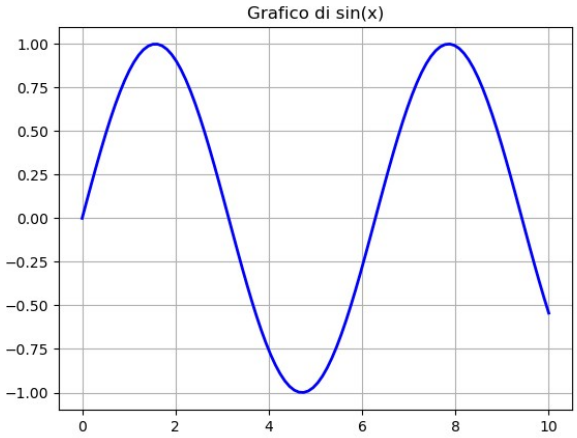

Figura 2.9 : Grafico del seno

Nota: Questa libreria necessità di essere installata. Per l'installazione bisognerà aprire il terminale ed eseguire il seguente comando:

pip install matplotlib

2.5.5 – Sympy

Sympy è una libreria dedicata al calcolo algebrico e all'analisi matematica. Questa libreria si basa sul calcolo simbolico, ovvero, alcuni termini matematici vengono trasformati in simboli invece di essere utilizzati come numeri scalari.

In esempio possiamo prendere il calcolo numerico dei radicali, considerando la seguente espressione:

$$\sqrt{18} + 2\sqrt{8} - \sqrt{2}$$

Nel calcolo numerico ogni radicale viene sostituito dal suo scalare:

$$\sqrt{18} + 2\sqrt{8} - \sqrt{2} = 4.242 + 5.656 - 1.414$$

Nel calcolo simbolico, il radicale viene considerato come un simbolo, quindi esso non viene calcolato:

$$\sqrt{18} + 2\sqrt{8} - \sqrt{2} = 6\sqrt{2}$$

Sympy ci permette proprio di andare a definire e manipolare espressioni simboliche, cioè ci permette di lavorare con variabili e funzioni senza specificare i valori numerici, finchè non sarà necessario. [8]

Sfruttando il calcolo simbolico questa libreria ci porta due grandi vantaggi:

- precisione elevata: può ottenere risultati esatti evitando problemi di arrotondamento e approssimazione nei calcoli numerici;

- riduzione degli errori: viene ridotta la possibilità di errori numerici dovuti a calcoli ripetitivi e a modelli matematici molto complessi.

Nota: Questa libreria necessità di essere installata. Per l'installazione bisognerà aprire il terminale ed eseguire il seguente comando:

pip install sympy

2.5.6 – Gazebo_msgs.srv

Gazebo_msg.srv è un pacchetto fornito da ROS che definisce messaggi di servizio che ci permettono di interagire con Gazebo. Questo pacchetto definisce diversi tipi di messaggi di servizio, tra cui servizi per controllare i modelli dei robot, leggerne i dati, modificarne le proprietà e molto altro.
In particolare il messaggio di servizio utilizzato per lo sviluppo del progetto è spawn_model, che ci permette di inviare richieste di spawn di un modello a Gazebo per aggiungere un nuovo modello nell'ambiente simulativo. Andremo ad analizzare più nel dettaglio la funzione di spawn nel capitolo successivo.

Nota: Dopo l'installazione di ROS, il pacchetto *gazebo_msgs.srv* è già disponibile nel sistema e pronto per essere utilizzato senza bisogno di ulteriori installazioni.

2.5.7 – Geometry_msgs.msg

Geometry_msgs.msg è un pacchetto fornito da ROS che definisce messaggi standard per la rappresentazione di dati geometrici utilizzati nella simulazione. Questo pacchetto definisce diversi tipi di messaggi standard che permettono la pianificazione del movimento di un robot, la localizzazione e la manipolazione.

In particolare il messaggio standard utilizzato per lo sviluppo del progetto è Pose, che ci permette di definire la posizione e l'orientamento di un oggetto nell'ambiente simulativo.

Nota: Dopo l'installazione di ROS, il pacchetto *geometry_msgs.msg* è già disponibile nel sistema e pronto per essere utilizzato senza bisogno di ulteriori installazioni.

2.5.8 – Scipy

Scipy è una libreria open-source per il calcolo scientifico e tecnico. Si basa sulla libreria sopraelencata Numpy, e usa lo stesso array multidimensionale come struttura base di dati. Una delle caratteristiche principali di scipy è la sua libreria di ottimizzazione, che fornisce strumenti per ridurre al minimo o massimizzare funzionalità specifiche, risolvere equazioni e adattare modelli ai dati. Inoltre, include una libreria di elaborazione del segnale, che consente agli utenti di estrarre ed elaborare informazioni da segnali audio o immagini. Scipy include anche librerie per analisi statistiche, elaborazione di immagini e altre applicazioni informatiche industriali. [9]

Tra i moduli che offre abbiamo utilizzato:

- **Scipy.special** : questo modulo contiene un ampio insieme di funzioni speciali della fisica matematica. Tra le funzioni disponibili in questo modulo andiamo ad utilizzare

binom, funzione che permette di calcolare coefficienti binominali, noti come "combinazioni", che rappresentano il numero di modi in cui è possibile scegliere 'H' elementi da un insieme di 'N' elementi senza considerarne l'ordine;

- **Scipy.interpolate** : questo modulo permette l'interpolazione di punti, ovvero permette di andare a generare dei punti tra i punti dati. Ad esempio per i punti 1 e 2, possiamo interpolare e trovare i punti 1.33 e 1.66. Tra le funzioni disponibili in questo modulo andiamo ad utilizzare **splprep** e **splev**, funzioni che permettono l'interpolazione e la valutazione di spline parametriche. In particolare **splprep** genera una rappresentazione di spline parametriche di dati multidimensionali per curve in 2D e 3D, mentre **splev** valuta una rappresentazione di spline parametriche a determinati parametri;

- **Scipy.spatial.distance** : questo modulo è un "sottopacchetto" di **scipy.spatial** e contiene funzioni per calcolare distanze tra punti e vettori. Tra le funzioni disponibili in questo modulo andiamo ad utilizzare **cdist**, funzione che permette di calcolare delle distanze tra coppie di punti.

Nota: Questa libreria necessità di essere installata. Per l'installazione bisognerà aprire il terminale ed eseguire il seguente comando:

pip install scipy

Capitolo 3

Interazione tra ROS e Gazebo tramite lo spawn di un modello

Il capitolo seguente riguarda l'interazione tra ROS e Gazebo, con un focus sull'utilizzo della funzione di spawn. Esamineremo la connessione tra i due sistemi per creare e controllare oggetti virtuali in una simulazione. Inoltre verrà mostrato come usare la funzione di spawn per aggiungere modelli e robot al mondo simulativo e saranno forniti esempi per illustrare come utilizzare questa funzione.

3.1 – Comunicazione tra ROS e Gazebo

La comunicazione tra ROS e Gazebo avviene tramite un bridge che consente ai due sistemi di interagire e scambiare informazioni durante la simulazione. In particolare ROS usa una raccolta di pacchetti chiamata *gazebo_ros_pkgs* che contiene un insieme di plugin che consentono tale comunicazione. Tra i plugin principali troviamo:

- *gazebo_ros*: plugin generali per l'interfaccia tra i due sistemi;
- *gazebo_ros_controll*: plugin che permette il controllo dei robot all'interno del mondo di simulazione;
- *gazebo_ros_laser*: plugin che permette di utilizzare sensori LIDAR.

Lo scambio di informazioni tra i due sistemi avviene tramite dei canali di comunicazione chiamati topic. Gazebo utilizza specifici topic che permette ai robot presenti nella simulazione di pubblicare dati su topic ROS, ai quali i nodi ROS possono sottoscriversi per ricevere dati. Dall'altra parte, i nodi ROS pubblicano comandi su topic a cui Gazebo può sottoscriversi, controllando la simulazione.

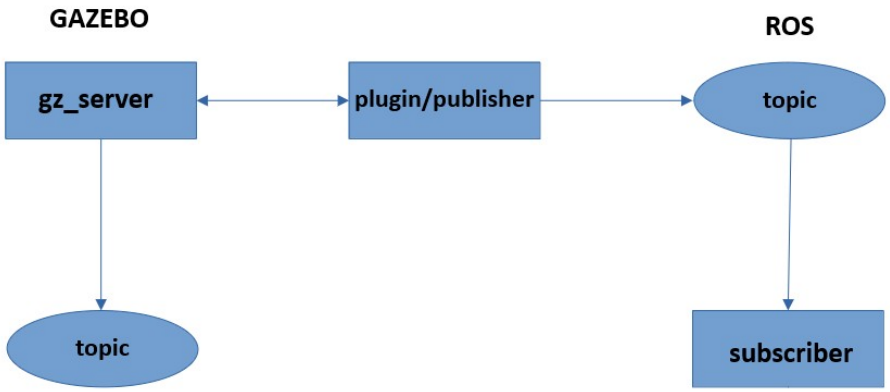

Figura 3.1: Comunicazione tra ROS e Gazebo

Oltre alla comunicazione tramite topic, ROS e Gazebo interagiscono tra di loro anche tramite dei servizi, che al contrario dei topic consentono ai nodi di richiedere operazioni specifiche e ottenere una risposta. Vengono utilizzati per operazioni che richiedono una risposta immediata e deterministica. Le caratteristiche principali dei servizi sono:

- sincroni: il client invia una richiesta al server e aspetta la sua risposta;
- definiti da messaggi: ogni servizio è definito da messaggi specifici sia per la richiesta che per la risposta;
- usati per operazioni specifiche: ad esempio, spawn di un modello, cancellazione di un modello, reset di una simulazione, ottenimento dello stato di un modello, ecc...

3.2 – Funzione spawn_model

La funzione di spawn è un servizio che consente di inserire modelli tridimensionali all'interno del nostro mondo simulativo tramite comandi da terminale o script. Il servizio di spawn è definito all'interno dei file di servizio 'gazebo_msgs/srv/SpawnModel.srv', dove ogni stringa definisce un parametro del servizio nel seguente modo:

```
1  # File: gazebo_msgs/srv/SpawnModel.srv
2
3  string model_name
4  #nome del modello da spawnare
5
6  string model_xml
7  #definizione del modello XML a cui fare riferimento, noi useremo dei modelli SDF
8
9  string robot_namespace
10 #spazio dei nomi dei robot, serve ad organizzare i nodi e i topic relativi ad un
       determinato robot, generalmente non viene specificato e può essere lasciato vuoto
11
12 geometry_msgs/Pose initial_pose
13 #indica la posizione iniziale del modello e l'orientazione
14
15 string reference_frame
16 #indica il frame di riferimento per lo spawn del modello, solitamente si usa 'world' per
       posizionarlo nel mondo di gazebo
17 --------
18 string status_message
19 #messaggio di stato, ci informa se il modello è stato spawnato con successo o meno
20
21 bool success
22 #True se il modello è stato spawnato con successo, altrimenti False
```

3.2.1 – Spawn di un modello tramite comando da terminale

Di seguito, viene illustrato il processo di spawn di un modello SDF in Gazebo utilizzando il terminale.

1. Prima di iniziare è necessario avviare la comunicazione tra ROS e Gazebo, questo comando avvierà il ROS master e successivamente avvierà Gazebo insieme al suo plugin ROS, permettendo la comunicazione bidirezionale tra i due sistemi:

   roscore & rosrun gazebo_ros gazebo

2. Adesso che i due sistemi stanno comunicando, possiamo proseguire con lo spawn del modello SDF dal database locale, in questo esempio abbiamo utilizzato come modello un cono per tracciato giallo già presente sul nostro database:

   rosrun gazebo_ros spawn_model -file ~/.gazebo/models/cone_yellow/model.sdf -sdf -model cone -y 0 -x 0 -z 0

   Nel comando seguente andiamo ad utilizzare i principali argomenti che permettono la configurazione ed il controllo dello spawn di un modello:

   - -file: specifica il percorso del file a cui fa riferimento il nostro modello;
   - -sdf: specifica che il modello XML utilizzato è un modello SDF;
   - -model: specifica il nome del modello che spawniamo, per questo argomento bisogna porre attenzione poiché il nome del modello deve essere unico, infatti se si tenta di spawnare due o più modelli con lo stesso nome si verificherà un errore;

- -x -y -z: specifica i componenti (x,y,z) della posizione iniziale, espressi in metri.

Esistono tanti altri argomenti per specificare l'orientazione dei modelli, per esempio potremmo specificare gli angoli di roll, pitch e yaw (-R -P -Y). Questi angoli permettono di specificare le coordinate angolari di un modello andando a descrivere esattamente come è orientato il modello rispetto al suolo. Nel contesto del nostro progetto lo spawn dei coni non richiede un'orientazione angolare, in quanto di default, vengono posizionati con la base piatta sul piano xy e il vertice rivolto verso l'asse z.

Nella figura seguente possiamo vedere il risultato del comando eseguito precedentemente:

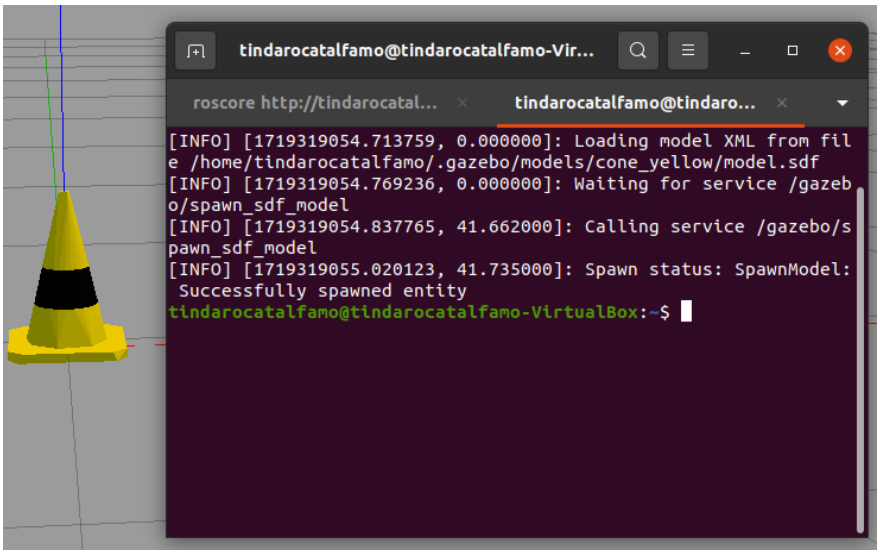

Figura 3.2: Spawn del modello

3.2.2 – Spawn di un modello tramite script Python

Di seguito, illustriamo il processo di spawn di un modello SDF in Gazebo tramite l'ausilio di uno script Python. Lo script in questione eseguirà 4 compiti importanti, come possiamo vedere in figura 3.3:

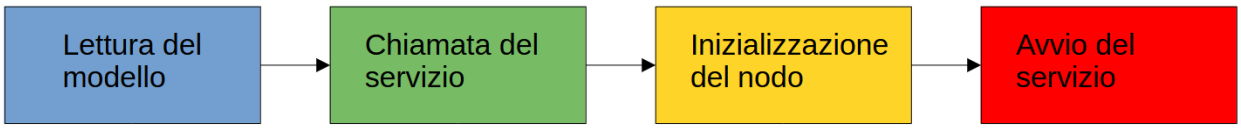

Figura 3.3: Schema a blocchi che rappresenta lo spawn del modello

Il codice si apre con l'inserimento delle librerie essenziali che verranno utilizzate e la cui utilità è stata descritta nel capitolo precedente nel paragrafo dedicato a Python.

```
1  #!/usr/bin/env python
2  import rospy
3  from gazebo_msgs.srv import SpawnModel
4  from geometry_msgs.msg import Pose
```

L'importazione di una libreria del nostro script avviene tramite l'uso dell'istruzione *import* seguita dal nome della libreria stessa. Se si vuole importare soltanto un modulo della libreria, useremo il comando *from* seguito dal nome della libreria e successivamente il comando *import* seguito dal nome del modulo specifico.

```
6  #Funzione di lettura del file
7  def read_sdf(filename):
8      with open(filename, 'r') as file:
9          model_content = file.read()
10     return model_content
```

La funzione *read_sdf* permette di andare a leggere il contenuto di un file SDF. Questa funzione accetta il parametro *filename* che specifica il percorso del file che si desidera leggere. In particolare questa funzione permette di aprire il file specificato in modalità lettura, successivamente va a leggere il contenuto del file in una stringa chiamata *model_content* e infine ci restituisce il contenuto.

```
14 #Funzione di spawn
15  def spawn(name, x, y, model_content):
16      spawn_model =
        rospy.ServiceProxy('steer_bot/gazebo/spawn_sdf_model',
        SpawnModel)
17      initial_pose = Pose()
18      initial_pose.position.x = x
19      initial_pose.position.y = y
20
21      response = spawn_model(name, model_content, "", initial_pose,
        "world")
22      rospy.loginfo(response.status_message)
```

La funzione *spawn* permette di chiamare il servizio di spawn. Questa funzione accetta 4 parametri:

- *name:* indica il nome dell'oggetto da spawnare;
- *model_content:* indica il percorso del file da leggere;
- *x* e *y:* indicano le coordinante per il posizionamento dell'oggetto.

Inizialmente andiamo a creare un proxy per il servizio *spawn_sdf_model*. Successivamente creiamo un oggetto *Pose* per il posizionamento del modello all'interno del nostro mondo, andando a impostare le coordinate *x* e *y*. Infine la funzione chiamerà il servizio di Gazebo relativo allo spawn e andrà anche a loggare il messaggio di stato della risposta tramite *rospy*.

```
25  #Funzione di inizializzazione del nodo
26  def node():
27      model_content = read_sdf('./steer_bot/cone_yellow/model.sdf')
28      rospy.init_node('spawn_model')
29      rospy.wait_for_service('steer_bot/gazebo/spawn_sdf_model')
30
31      try:
32          spawn('example_model', 1, 2, model_content)
33
34      except rospy.ServiceException as exc:
35          rospy.logerr("Error during service invocation: %s",
            str(exc))
```

Node è la funzione principale, essa ci permette di inizializzare un nodo ROS e successivamente spawnare un modello. In particolare tramite la funzione *read_sdf* andiamo a leggere il file SDF specificato presente nel nostro database locale. Successivamente tramite l'utilizzo di *rospy* andiamo ad inizializzare un nodo ROS che chiamiamo *spawn_model* e attendiamo che il servizio *spawn_sdf_model* sia disponibile. Infine richiamiamo la funzione di *spawn* descritta precedentemente, e tramite *except* andiamo a gestire eventuali eccezioni durante l'invocazione del servizio.

```
36  #MAIN
37  if __name__ == "__main__":
38     node()
```

Ultimo ma non meno importante, andiamo ad avviare l'intero processo chiamando la funzione *node* nel *Main.*

Capitolo 4

Realizzazione tracciati da gara

Il capitolo seguente riguarda l'utilizzo di script per la realizzazione di tracciati all'interno dell'ambiente di simulazione. In particolare verrà descritta la componente geometrica necessaria per la creazione dei tracciati, invece la descrizione per lo spawn degli oggetti verrà volutamente esclusa in quanto le funzioni relative allo spawn sono state discusse nel capitolo precedente.

Nota: Prima di eseguire gli script presentati in questo capitolo, è fondamentale avviare il mondo di Gazebo.

4.1 – Concetti matematici per la realizzazione di tracciati

In questa sezione verranno introdotti i concetti matematici e geometrici che costituiscono la base dei nostri script per la realizzazione di tracciati da gara. Comprendere questi concetti è essenziale per sviluppare tracciati accurati. In particolare, utilizzeremo una varietà di equazioni e funzioni matematiche, tra cui:

- Equazione del cerchio
- Funzioni trigonometriche
- Curve di Bézier
- Polinomi di Bernstein

- Spline di Bézier

Questi strumenti matematici ci consentono di definire ed manipolare i tracciati con maggiore precisione, permettendoci di creare tracciati più realistici per le nostre simulazioni. Successivamente, in questo capitolo esamineremo come ciascuno di questi concetti viene utilizzato in script specifici per costruire i nostri tracciati.

4.1.1 – L'equazione del cerchio e le funzioni trigonometriche

Andiamo ad introdurre i concetti matematici fondamentali per la generazione del percorso a forma di otto e per il posizionamento dei coni in punti specifici.

Uno dei concetti principali riguarda l'equazione del cerchio, definita come:

$$(x - h)^2 + (y - k)^2 = r^2$$

dove (h,k) rappresentano le coordinate del centro del cerchio e r rappresenta il raggio. Questa equazione ci permette di definire un insieme di punti equidistanti dal centro, formando cosi il nostro cerchio.

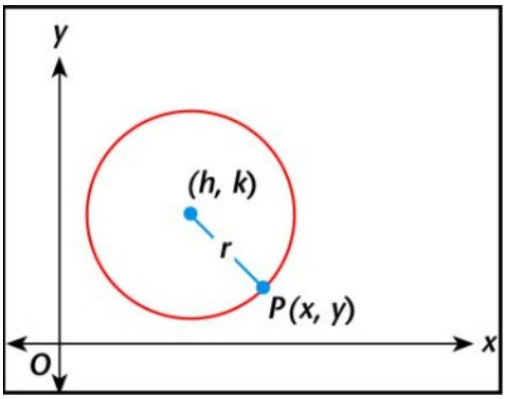

Figura 4.1: Cerchio

Per il calcolo di punti specifici lungo la circonferenza, vengono impiegate le funzioni trigonometriche seno e coseno.

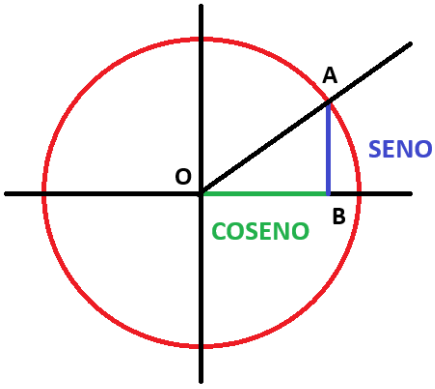

Figura 4.2: Seno e coseno

La funzione seno è definita come il rapporto tra il lato opposto e l'ipotenusa di un triangolo rettangolo.
La funzione coseno è definita come il rapporto tra il lato adiacente e l'ipotenusa di un triangolo rettangolo. [10]

Utilizzando la parametrizzazione ϴ, variabile che rappresenta l'angolo in radianti, è possibile calcolare le coordinate di punti sul cerchio mediante le formule:

$$x = r\,cos\,(\theta) + h$$

$$y = r\,sen\,(\theta) + k$$

Questi punti vengono distribuiti uniformemente lungo la circonferenza variando l'angolo da 0 a 2π. Queste funzioni sono importantissime in questo contesto perché ci permettono di convertire un angolo in una posizione cartesiana, in modo da facilitare la generazione ci cerchi perfetti. Inoltre, la risoluzione di equazioni quadratiche è essenziale per andare a calcolare i punti intermedi necessari per posizionare i coni in posizioni specifiche.

4.1.2 – Curve di Bézier e polinomi di Bernstein

Nel contesto della generazione di percorsi curvilinei, giocano un ruolo importantissimo i concetti di curva di Bézier e polinomi di Bernstein.

Una curva di Bézier è una curva parametrica utilizzata in computer grafica nella progettazione di animazioni al computer. Questo concetto prende il nome da Pierre Bézier, che utilizzò nel 1960 per disegnare curve nella progettazione delle carrozzerie di vetture Renault. Tali curve possono essere combinate tra di loro per formare spline di Bézier, in modo da rappresentare superfici superiori rispetto ad una semplice curva. [11]

Queste curve si basano sui polinomi di Bernstein, che sono definite tipicamente sull' intervallo [0,1]. Per una curva con $n+1$ punti di controllo $P0, P1, \ldots, Pn,$ i polinomi di Bernstein $B_{i,n}(t)$ sono definiti come:

$$B_{i,n}(t) = \binom{n}{i} t^{i} (1-t)^{n-i}$$

dove:

- $i = 0, 1, \ldots, n$;
- t è il parametro che varia da 0 a 1 e permette la parametrizzazione della curva. Man mano che esso varia il punto $B(t)$ segue la curva determinata dai punti di controllo. Quando $t = 0$, la curva passa per il punto P0, mentre quando $t = 1$ la curva passa per il punto Pn;
- n è chiamato grado della curva.

Per la realizzazione del tracciato casuale, sono state utilizzate curve di Bézier di grado 2, o quadratiche. Queste curve presentano 3 punti di controllo *P0, P1, P2*. La curva risultante è ottenuta tramite la seguente equazione:

$$B(t) = (1-t)^2 P0 + 2(1-t)\,t\,P1 + t^2 P2$$

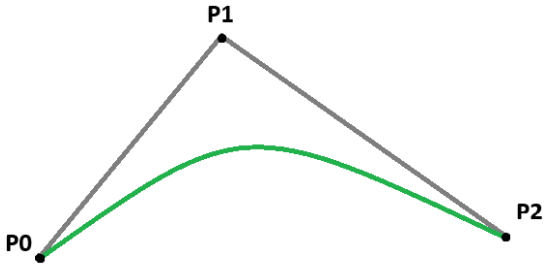

Figura 4.3: Curva quadratica di Bézier

4.1.3 – Spline di Bézier

Per realizzare un intero tracciato utilizzando curve di Bézier è necessario applicare il concetto di spline. Questo concetto ci permette di ottenere curve più complesse collegando tra di loro diverse curve.

In pratica, in una spline, le curve sono unite in sequenza , dove il punto finale di una curva coincide con il punto iniziale della curva successiva. Questo collegamento è noto come continuità di tipo C0. Oltre a tale continuità, che assicura la connessione dei punti, esistono altri criteri di continuità geometrica per l'incollamento delle curve, ad esempio G0, G1, G2. Per il nostro tracciato è stato implementato il criterio G1, che richiede sia di rispettare la condizione della continuità C0, sia che le tangenti alle curve nei punti di connessione siano collineari, cioè che la direzione della tangente cambi in maniera continua.

Nella figura 4.10 possiamo vedere un esempio di spline di Bézier con continuità C0 e G1. Le due curve sono collegate in modo tale che l'ultimo punto della prima curva (blu) coincida con il primo punto della seconda curva (verde). Inoltre, i punti di controllo intermedi sono allineati in modo che le tangenti alle curve nei punti di connessione siano collineari.

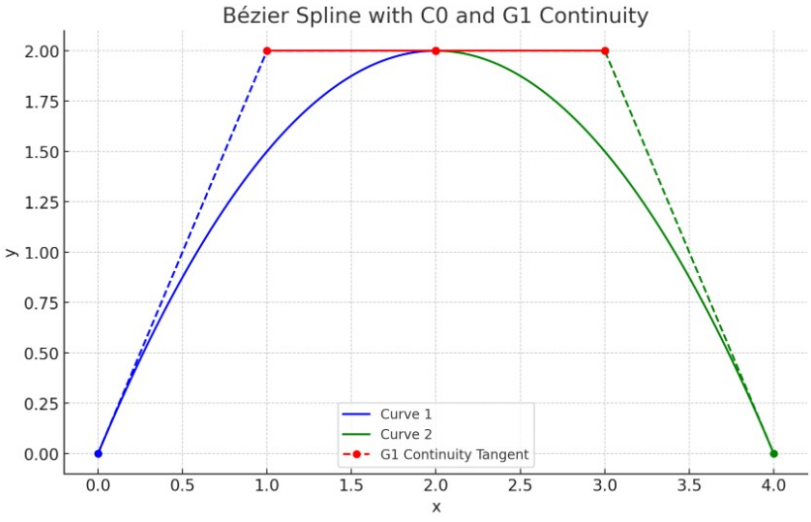

Figura 4.4: Spline di Bézier

4.2 – Straight path

Per primo esamineremo il tracciato rettilineo, un percorso semplice e diretto che permette di studiare i movimenti del nostro robot lungo una linea retta.

```
1  #!/usr/bin/env python
2
3  import rospy
4  import xml.etree.ElementTree as ET
5  from gazebo_msgs.srv import SpawnModel
6  from geometry_msgs.msg import Pose
```

Il codice si apre con l'importazione delle librerie necessarie per la creazione del tracciato.

```
8  def modify_and_read_sdf(filename, percentage):
9      #Funzione interna per la modifica della dimensione dei coni
10     def modify_sdf_scale(filename, percentage):
11         tree = ET.parse(filename)
12         root = tree.getroot()
13
14         #Trova l'elemento <scale> e lo modifica
15         for scale_elem in root.iter('scale'):
16             scale_elem.text = f"{1*(percentage/100)}
               {1*(percentage/100)} {1*(percentage/100)}"
17
18         #Salva il file SDF modificato
19         tree.write(filename, encoding="UTF-8", xml_declaration=True)

20     #Modifica il file SDF
21     modify_sdf_scale(filename, percentage)
22     #Leggi il contenuto del file SDF modificato
23     with open(filename, 'r') as file:
24         sdf_content_modified = file.read()
25
26     return sdf_content_modified
```

La funzione *modify_and_read_sdf* viene utilizzata per la modifica e la lettura del file. Questa funzione accetta due parametri :

- *filename:* indica il percorso del file da modificare e leggere;
- *percentage:* indica la percentuale di ridimensionamento che verrà applicata al modello.

La funzione include al suo interno una sotto-funzione *modify_sdf_scale*, la quale va a modificare tutti gli elementi <*scale*> dei file applicando il ridimensionamento, che viene eseguito moltiplicando la dimensione attuale del modello per *percentage/100.* Dopo aver effettuato la modifica, la funzione legge il file e lo restituisce.

```
30  #Funzione che richiama spawn_sdf_model
31  def spawn(name, x, y, z, model_content):
32      spawn_model =
        rospy.ServiceProxy('steer_bot/gazebo/spawn_sdf_model',
        SpawnModel)
33      x = float(x)
34      y = float(y)
35      z = float(z)
36      initial_pose = Pose()
37      initial_pose.position.x = x
38      initial_pose.position.y = y
39      initial_pose.position.z = z
40      response = spawn_model(name, model_content, "", initial_pose,
        "world")
41      rospy.loginfo(response.status_message)

42  #Funzione che crea il percorso rettilineo
43  def create_straight_path(length, percentage, num_cones):
44      #Calcola la lunghezza effettiva in base alla percentuale
45      actual_length = length * (percentage / 100.0)
46
47      #Calcola lo spaziamento tra i coni
48      spacing = actual_length / num_cones
49
```

```
50      #Crea coordinate per il percorso destro
51      right_path = [(i * spacing, 1) for i in range(num_cones)]
52
53      #Crea coordinate per il percorso sinistro
54      left_path = [(i * spacing, -1) for i in range(num_cones)]
55
56      return right_path, left_path
```

La funzione *create_straight_path* ha il compito di generare i punti del nostro tracciato rettilineo. Questa funzione accetta tre parametri:

- *lenght:* indica la lunghezza di riferimento del percorso e viene espressa in metri;
- *percentage:* indica la percentuale di ridimensionamento del percorso rispetto alla sua lunghezza iniziale;
- *num_cones:* indica il numero di coni desiderato su ciascuno dei lati del percorso.

Inizialmente, viene calcolata la lunghezza effettiva del percorso moltiplicando la lunghezza di riferimento per la percentuale fornita. Successivamente, andiamo a calcolare lo spaziamento dei coni dividendo la lunghezza effettiva per il numero di coni desiderato. Infine, verranno create e restituite due liste di tuple di coordinate:

- *right_path:* rappresenta il tracciato sul lato destro. I coni verranno posizionati lungo l'asse x con valore 1 sull'asse y;
- *left_path:* rappresenta il tracciato sul lato sinistro. I coni verranno posizionati lungo l'asse x con valore -1 sull'asse y.

```
59  #Funzione principale
60  def node(percentage):
61      rospy.init_node('spawn_model')
62      rospy.wait_for_service('steer_bot/gazebo/spawn_sdf_model')
63
64      #Definiamo la lunghezza di riferimento del percorso
65      reference_length = 20.0
66      #Numero di coni per ogni lato
67      num_cones = 14
68
69      #Inserisce le coordinate del percorso di destra e di sinistra
        in due liste
70      right_path, left_path = create_straight_path(reference_length,
        percentage, num_cones)
```

La funzione *node* è responsabile della gestione degli spawn e accetta il parametro *percentage*. Inizialmente, verrà definito il percorso inizializzando le variabili relative alla lunghezza del tracciato e al numero di coni desiderato. Successivamente, verrà richiamata la funzione *create_straight_path* per calcolare le coordinate del percorso.

```
72      #Spawn dei coni
73      try:
74          for i, (x, y) in enumerate(right_path):
75              if 0<= i <= 1:
76                  spawn(f'pointB_{i}', x, y, 0,
                     MODEL_CONTENT_CONE_ORANGE_BIG)
77              if 2<= i <= 7:
78                  spawn(f'pointB_{i}', x, y, 0,
                     MODEL_CONTENT_CONE_BLUE)
79              if 8<= i <= 9:
80                  spawn(f'pointB_{i}', x, y, 0,
                    MODEL_CONTENT_CONE_ORANGE_BIG)
81              if 10<= i <= 13:
82                  spawn(f'pointB_{i}', x, y, 0,
                    MODEL_CONTENT_CONE_ORANGE)
83          for i, (x, y) in enumerate(left_path):
84              if 0<= i <= 1:
```

```
85                spawn(f'pointY_{i}', x, y, 0,
                   MODEL_CONTENT_CONE_ORANGE_BIG)
86              if 2<= i <= 7:
87                spawn(f'pointY_{i}', x, y, 0,
                  MODEL_CONTENT_CONE_YELLOW)
88              if 8<= i <= 9:
89                spawn(f'pointY_{i}', x, y, 0,
                  MODEL_CONTENT_CONE_ORANGE_BIG)
90              if 10<= i <= 13:
91               spawn(f'pointY_{i}', x, y, 0,
                 MODEL_CONTENT_CONE_ORANGE)
92
93      except rospy.ServiceException as exc:
94             rospy.logerr("Errore durante l'invocazione del servizio:
               %s", str(exc))
```

Infine le due liste di tuple contenenti le coordinate verranno iterate, chiamando la funzione spawn per posizionare i modelli, inoltre verranno eseguite delle condizioni tramite *if* che ci permettono di inserire determinati coni in base alla loro posizione lungo il tracciato.

```
100 #MAIN
101  if __name__ == '__main__':
102  #Inseriamo la percentuale per ridimensionare il percorso
103     while True:
104          percentage = int(input("Inserisci un valore per
             ridimensionare il  percorso (0-100%): "))
105          if 0 <= percentage <= 100:
106             break
107          else:
108             print("Percentuale invalida. Perfavore inserisci una
                percentuale tra 0 e 100")
109
110     #Modifica del file sdf
111     MODEL_CONTENT_CONE_YELLOW =
        modify_and_read_sdf('./steer_bot/cone_yellow/model.sdf',
        percentage)
112     MODEL_CONTENT_CONE_BLUE =
        modify_and_read_sdf('./steer_bot/cone_blue/model.sdf',
        percentage)
```

```
113     MODEL_CONTENT_CONE_ORANGE =
        modify_and_read_sdf('./steer_bot/cone_orange/model.sdf',
        percentage)
114     MODEL_CONTENT_CONE_ORANGE_BIG =
        modify_and_read_sdf('./steer_bot/cone_orange_big/model.sdf',
        percentage)
115     #Richiama la funzione principale
116     node(percentage)
```

Nel blocco del *main*, inizialmente andremo a consentire all'utente di definire la percentuale di ridimensionamento del percorso e dei coni, il cui valore dovrà essere compreso tra 0 e 100. Successivamente, andremo a richiamare la funzione di modifica e lettura per ciascun modello passando il percorso del file e la variabile *percentage*. Infine, richiameremo la funzione Node.

Nella seguente figura possiamo vedere la generazione del tracciato rettilineo su Gazebo.

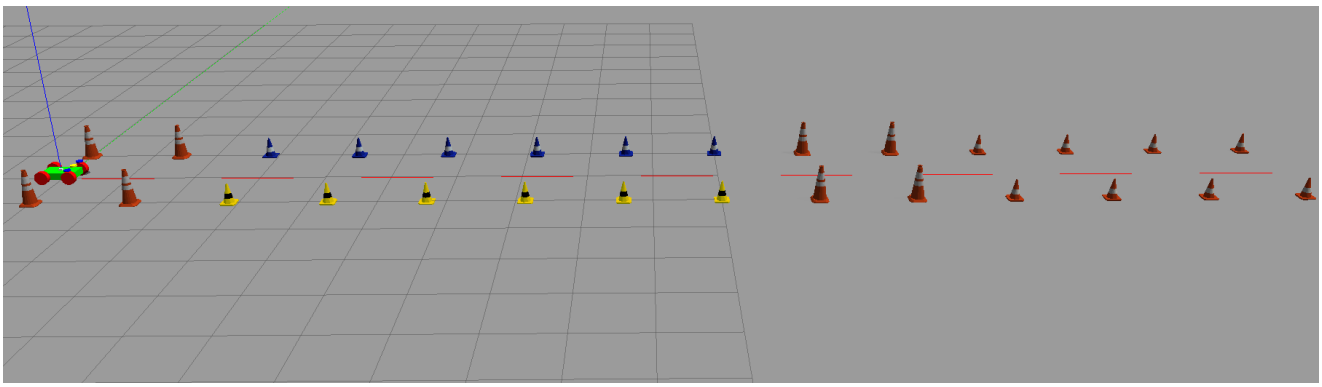

Figura 4.5: Tracciato rettilineo

4.3 – Eight path

Adesso andiamo ad esaminare il tracciato a forma di otto, un percorso molto più complesso rispetto al precedente che ci permette di visualizzare il comportamento del nostro robot in condizioni di cambi di direzione, come incroci.

```
1  #!/usr/bin/env python
2
3  import matplotlib.pyplot as plt
4  import numpy as np
5  import rospy
6  import xml.etree.ElementTree as ET
7  import sympy as sp
8  from gazebo_msgs.srv import SpawnModel
9  from geometry_msgs.msg import Pose
```

Il codice si apre con l'importazione delle librerie necessarie per la creazione del tracciato.

```
11 def modify_and_read_sdf(filename, percentage):
12     #Funzione interna per la modifica della dimensione dei coni
13     def modify_sdf_scale(filename, percentage):
14         tree = ET.parse(filename)
15         root = tree.getroot()
16
17     #Trova l'elemento <scale> e lo modifica
18         for scale_elem in root.iter('scale'):
19             scale_elem.text = f"{1*(percentage/100)}
               {1*(percentage/100)} {1*(percentage/100)}"
20
21         #Salva il file SDF modificato
22         tree.write(filename, encoding="UTF-8", xml_declaration=True)
23
24     #Modifica il file SDF
25     modify_sdf_scale(filename, percentage)
26
27     #Leggi il contenuto del file SDF modificato
28      with open(filename, 'r') as file:
```

```
29              sdf_content_modified = file.read()
30
31          return sdf_content_modified
```

Successivamente, troviamo di nuovo la funzione *modify_and_read_sdf*, che svolgerà lo stesso compito descritto in precedenza.

```
33  #Funzione che genera il cerchio
34  def generate_circle(radius, center_y, center_x, num_points):
35      coordinates_circle = []
36      t = np.linspace(0, 2 * np.pi, num_points)
37      x = radius * np.cos(t) + center_x
38      y = radius * np.sin(t) + center_y
39
40      for i in range(num_points):
41          coordinates_circle.append((x[i], y[i]))
42      return coordinates_circle
```

La funzione *generate_circle* è responsabile della generazione del percorso sui cerchi esterni e interni del tracciato. Questa funzione accetta 4 parametri:

- *radius:* indica il raggio del cerchio da generare;
- *center_y* e *center_x:* indicano le coordinate del centro del cerchio;
- *num_points:* indica il numero di punti utilizzato per la generazione del cerchio.

Inizialmente, verrà generato un array *t* di angoli uniformemente distribuiti tra 0 e 2π. Successivamente, utilizzando le funzioni trigonometriche *np.cos* e *np.sin,* verranno calcolate le coordinate di ogni punto del cerchio spostate relativamente lungo i due assi x e y rispetto

al centro. Infine, ogni coordinata verrà inserita all'interno di una lista di tuple che ci verrà restituita dalla funzione.

```
44 #Funzione che risolve l'equazione del cerchio per trovare punti
     specifici  per il piazzamento dei coni
45  def solve_quadratic_equation(center_x, center_y, radius, y):
46      #Definisce la variabile simbolica
47      x = sp.symbols('x')
48
49      #Definisce l'equazione del cerchio
50      equazione = (x - center_x)**2 + (y - center_y)**2 - (radius)**2
51
52      #Espansione dei quadrati dell'equazione
53      equazione_espansa = sp.expand(equazione)
54
55      #Riorganizzazione dell'equazione in forma canonica
56      equazione_canonica = sp.collect(equazione_espansa, x)
57
58      #Calcolo delle soluzioni
59       soluzioni = sp.solve(equazione_canonica, x)
60      return soluzioni
```

La funzione *solve_quadratic_equation* è utilizzata per trovare la posizione specifica di determinati punti sul percorso circolare. Questa funzione accetta 4 parametri:

- *radius*, *center_x* e *center_y:* indicano il raggio e i centri del cerchio per il risolvimento dell'equazione;

- *y:* indica la coordinata per cui trovare la soluzione.

Inizialmente, andiamo a definire la variabile simbolica *x* e l'equazione del cerchio da andare a risolvere. Successivamente, andiamo a espandere i quadrati dell'equazione e a

riorganizzarla in forma canonica. Infine, andiamo a risolvere l'equazione per x, restituendo le soluzioni.

```
62  #Funzione di spawn
63  def spawn(name, x, y, model_content):
64      spawn_model =
        rospy.ServiceProxy('steer_bot/gazebo/spawn_sdf_model',
        SpawnModel)
65      initial_pose = Pose()
66      initial_pose.position.x = x
67      initial_pose.position.y = y
68
69      response = spawn_model(name, model_content, "", initial_pose,
        "world")
70      rospy.loginfo(response.status_message)
71
72  #Funzione che spawna i coni su Gazebo
73  def node(blue_inner_circle, yellow_inner_circle,
    yellow_outer_circle, blue_outer_circle, percentage):
74      rospy.init_node('spawn_model')
75      rospy.wait_for_service('steer_bot/gazebo/spawn_sdf_model')
76      #Array che contengono i coni sui cerchi esterni da non spawnare
77      SKIP_Y_OUT = [0, 2, 3, 4, 5, 6]
78      SKIP_B_OUT = [0, 10, 11, 12, 13, 14]
```

La funzione *node* è la responsabile della gestione degli spawn. Questa funzione accetta 5 parametri:

- *blue_inner_circle* e *blue_outer_circle*: indicano le liste di tuple che contengono rispettivamente le coordinate dei cerchi blu interni ed esterni;

- *yellow_inner_circle* e *yellow_outer_circle:* indicano le liste di tuple che contengono rispettivamente le coordinate dei cerchi gialli interni ed esterni;

- *percentage:* indica la percentuale di ridimensionamento del percorso rispetto alla sua grandezza iniziale.

Inizialmente, andiamo a definire *SKIP_Y_OUT* e *SKIP_B_OUT*, due liste che contengono gli indici dei coni gialli e blu dei cerchi esterni che non devono essere spawnati.

```
79      try:
80          for i, (x, y) in enumerate(blue_inner_circle[:-1]):
81              if i == 4:
82                  #Spawn coni intermedi gialli
83                  cone = solve_quadratic_equation(center_x, center_y1,
                    radius2, y)
84                  spawn('point_MID_1', cone[0], y,
                    MODEL_CONTENT_CONE_YELLOW)
85                  spawn('point_MID_2', cone[1], y,
                    MODEL_CONTENT_CONE_YELLOW)
86                  #Spawn coni di partenza arancioni
87                  spawn('point_START_1', blue_inner_circle[i][0] -
                    (5*percentage/100), blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
88                  spawn('point_START_2', blue_inner_circle[i][0] -
                    (5.80*percentage/100),  blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
89                  #Spawn coni di arrivo arancioni
90                  spawn('point_END_1', blue_inner_circle[i][0] +
                    (5*percentage/100),     blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
91                  spawn('point_END_2', blue_inner_circle[i][0] +
                    (6*percentage/100),     blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
92                  spawn('point_END_3', blue_inner_circle[i][0] +
                    (7*percentage/100),     blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
93                  spawn('point_END_4', blue_inner_circle[i][0] +
                    (8*percentage/100),     blue_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
94                  #Spawn coni intermedi arancioni grandi
95                  point_x = (blue_inner_circle[i-1][0] +
                    blue_outer_circle[i][0])/2
96                  point_y = (blue_inner_circle[i-1][1] +
                    blue_inner_circle[i][1])/2
```

```
97                  spawn('point_OR_1', point_x, point_y,
                    MODEL_CONTENT_CONE_ORANGE_BIG)
98                  point_x = (blue_inner_circle[i+1][0] +
                    blue_outer_circle[i][0])/2
99                  point_y = (blue_inner_circle[i+1][1] +
                    blue_inner_circle[i][1])/2
100                 spawn('pointB_OR_2', point_x, point_y,
                    MODEL_CONTENT_CONE_ORANGE_BIG)
101             #Spawn coni cerchio interno blu
102             spawn(f'pointB_IN_{i}', x, y, MODEL_CONTENT_CONE_BLUE)
```

Il posizionamento dei nostri coni avviene mediante l'iterazione di cicli *for*. Nel primo ciclo vengono posizionati i coni blu sul cerchio interno. Un'attenzione particolare dobbiamo porla quando l'indice *i* del nostro ciclo sarà uguale a 4, poiché verranno posizionati i coni intermedi gialli , i coni arancioni di partenza, i coni arancioni di arrivo e i coni arancioni grandi intermedi nel cerchio blu interno:

- Per i coni intermedi gialli, le coordinate vengono trovate tramite l'utilizzo della funzione *solve_quadratic_equation;*

- Per i coni di partenza e di arrivo arancioni, essi vengono posizionati rispettivamente a sinistra e a destra del cono blu corrente, cioè del cono blu le cui coordinate si trovano nella posizione *i=4* della lista *blue_inner_circle;*

- Per i coni arancioni grandi intermedi, essi vengono posizionati utilizzando il punto medio tra due posizioni, una coppia di coordinate del cono blu interno precedente e una del cono blu esterno corrente.

```
103         for i, (x, y) in enumerate(yellow_inner_circle[:-1]):
104             if i == 12:
105                 #Spawn coni intermedi blu
106                 cone = solve_quadratic_equation(center_x, center_y,
                    radius2, y)
107                 spawn('point_MID_3', cone[0], y,
                    MODEL_CONTENT_CONE_BLUE)
108                 spawn('point_MID_4', cone[1], y,
                    MODEL_CONTENT_CONE_BLUE)
109                 #Spawn coni di partenza arancioni
110                 spawn('point_START_3', yellow_inner_circle[i][0] -
                    (5*percentage/100),    yellow_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
111                 spawn('point_START_4', yellow_inner_circle[i][0] -
                    (5.80*percentage/100), yellow_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
112                 #Spawn coni di arrivo arancioni
113                 spawn('point_END_5', yellow_inner_circle[i][0] +
                    (5*percentage/100), yellow_inner_circle[i][1],
                     MODEL_CONTENT_CONE_ORANGE)
114                 spawn('point_END_6', yellow_inner_circle[i][0] +
                    (6*percentage/100),    yellow_inner_circle[i][1],
                     MODEL_CONTENT_CONE_ORANGE)
115                 spawn('point_END_7', yellow_inner_circle[i][0] +
                    (7*percentage/100),    yellow_inner_circle[i][1],
                      MODEL_CONTENT_CONE_ORANGE)
116                 spawn('point_END_8', yellow_inner_circle[i][0] +
                    (8*percentage/100),    yellow_inner_circle[i][1],
                    MODEL_CONTENT_CONE_ORANGE)
117                 #Spawn coni intermedi arancioni grandi
118                  point_x = (yellow_inner_circle[i-1][0] +
                     yellow_outer_circle[i][0])/2
119                  point_y = (yellow_inner_circle[i-1][1] +
                     yellow_inner_circle[i][1])/2
120                  spawn('point_OR_3', point_x, point_y,
                     MODEL_CONTENT_CONE_ORANGE_BIG)
121                  point_x = (yellow_inner_circle[i+1][0] +
                     yellow_outer_circle[i]      [0])/2
122                  point_y = (yellow_inner_circle[i+1][1] +
                     yellow_inner_circle[i]      [1])/2
123                 spawn('point_OR_4', point_x, point_y,
                     MODEL_CONTENT_CONE_ORANGE_BIG)
124             #Spawn coni cerchio interno giallo
125             spawn(f'pointY_IN_{i}', x, y, MODEL_CONTENT_CONE_YELLOW)
```

Nel secondo ciclo verranno posizionati i coni gialli sul cerchio interno, anche qui bisogna porre attenzione quando l'indice *i* del nostro ciclo sarà uguale a 12, poiché verranno posizionati in maniera analoga al primo ciclo i coni intermedi blu , i coni arancioni di partenza, i coni arancioni di arrivo e i coni arancioni grandi intermedi nel cerchio giallo interno.

```
127         #Spawn coni cerchio esterno giallo
128         for i, (x, y) in enumerate(yellow_outer_circle):
129            if i not in SKIP_Y_OUT:
130               spawn(f'pointY_OUT_{i}', x, y,
                  MODEL_CONTENT_CONE_YELLOW)
131         #Spawn coni cerchio esterno blu
132         for i, (x, y) in enumerate(blue_outer_circle):
133            if i not in SKIP_B_OUT:
134               spawn(f'pointB_OUT_{i}', x, y, MODEL_CONTENT_CONE_BLUE)
135     except rospy.ServiceException as exc:
136            rospy.logerr("Error during service invocation: %s",
               str(exc))
```

Nel terzo e quarto ciclo verranno spawnati rispettivamente i coni gialli e blu sui cerchi esterni, inoltre tramite l'utilizzo delle due liste precedentemente definite andiamo ad evitare lo spawn dei coni che non servono al fine del tracciato.

```
146 #MAIN
147  if __name__ == "__main__":
148
149     #Inseriamo la percentuale per ridimensionare il percorso
150      while True:
151           percentage = int(input("Inserisci un valore per
             ridimensionare il percorso (0-100%): "))
152           if 0 <= percentage <= 100:
153              break
154           else:
155           print("Percentuale invalida. Perfavore inserisci una
              percentuale tra  0 e 100")
```

```
156     #Modifica e lettura dei file SDF
157     MODEL_CONTENT_CONE_YELLOW =
        modify_and_read_sdf('./steer_bot/cone_yellow/model.sdf',
        percentage)
158     MODEL_CONTENT_CONE_BLUE =
        modify_and_read_sdf('./steer_bot/cone_blue/model.sdf',
        percentage)
159     MODEL_CONTENT_CONE_ORANGE =
        modify_and_read_sdf('./steer_bot/cone_orange/model.sdf',
        percentage)
160     MODEL_CONTENT_CONE_ORANGE_BIG =
        modify_and_read_sdf('./steer_bot/cone_orange_big/model.sdf',
        percentage)
```

Nel blocco del *main*, inizialmente andremo a consentire all'utente di definire la percentuale di ridimensionamento del percorso e dei coni. Dopodichè, andiamo a leggere il contenuto dei nostri modelli SDF.

```
161     #Caratteristiche del percorso
162     radius1 = 2.5 * (percentage/100) #Raggio dei cerchi interni
163     radius2 = 5 * (percentage/100) #Raggio dei cerchi esterni
164     center_y = 3.75 * (percentage/100)
165     center_y1 = -3.75 * (percentage/100)
166     center_x = 6 * (percentage/100)
167     num_points = 17

168     #Genera i cerchi interni e inserisce negli array le coordinate
        dei num_points dei cerchi interni
169     blue_inner_circle = generate_circle(radius1, center_y1,
        center_x, num_points)
170     yellow_inner_circle = generate_circle(radius1, center_y,
        center_x, num_points)
171     #Genera i cerchi esterni e inserisce negli array le coordinate
        dei num_points dei cerchi esterni
172     yellow_outer_circle = generate_circle(radius2, center_y1,
        center_x, num_points)
173     blue_outer_circle= generate_circle(radius2, center_y, center_x,
        num_points)
174     #Spawn dei coni
175     node(blue_inner_circle, yellow_inner_circle,
        yellow_outer_circle, blue_outer_circle, percentage)
```

Successivamente, andiamo a definire le caratteristiche del percorso e a generare i nostri 4 cerchi, richiamando 4 volte la funzione *generate_circle*. Una volta che i nostri cerchi sono stati generati, andiamo a richiamare la funzione *node* per lo spawn del tracciato.

```
177     #Plot dei cerchi
178     blue_inner_x, blue_inner_y = zip(*blue_inner_circle)
179     yellow_inner_x, yellow_inner_y = zip(*yellow_inner_circle)
180     yellow_outer_x, yellow_outer_y = zip(*yellow_outer_circle)
181     blue_outer_x, blue_outer_y = zip(*blue_outer_circle)
182     plt.figure(figsize=(8, 8))
183     plt.scatter(blue_inner_x, blue_inner_y, color='blue',
        label='Cerchio   Blu Interno')
184     plt.scatter(yellow_inner_x, yellow_inner_y, color='yellow',
        label='Cerchio Giallo Interno')
185     plt.scatter(yellow_outer_x, yellow_outer_y, color='yellow',
        label='Cerchio Giallo Esterno')
186     plt.scatter(blue_outer_x, blue_outer_y, color='blue',
        label='Cerchio   Blu Esterno')
188     plt.xlabel('X')
189     plt.ylabel('Y')
190     plt.axhline(0, color='black',linewidth=0.5)
191     plt.axvline(0, color='black',linewidth=0.5)
192     plt.grid(color = 'gray', linestyle = '--', linewidth = 0.5)
193     plt.legend()
194     plt.axis('equal')
195     plt.show()
```

Infine, mediante le istruzioni della libreria plot andiamo a visualizzarci graficamente la generazione dei nostri 4 cerchi.

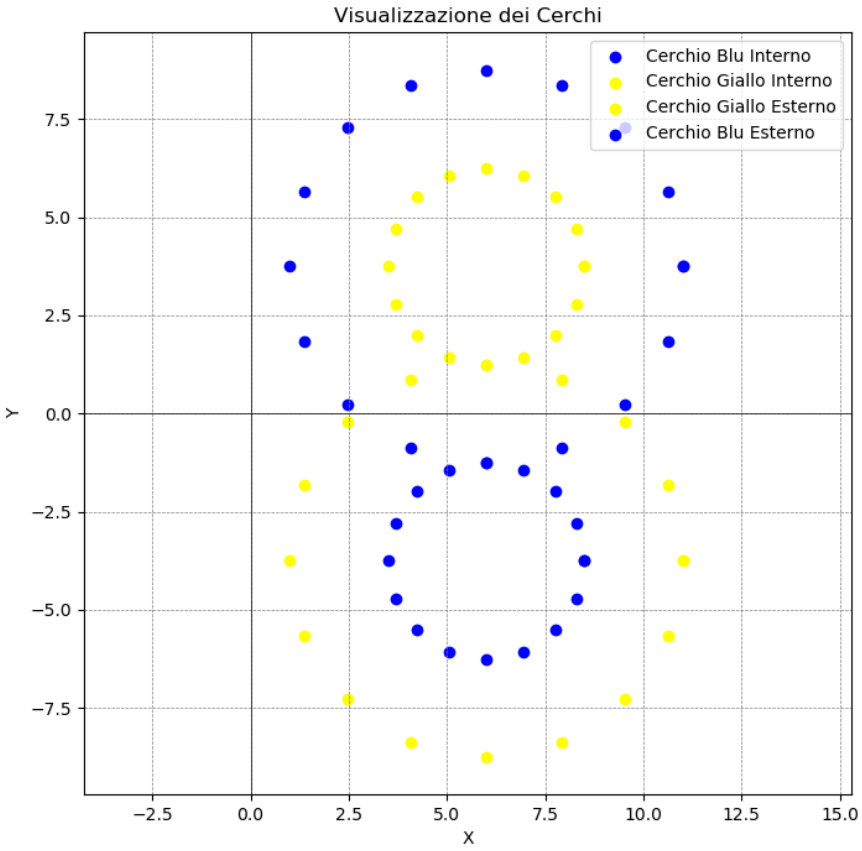

Figura 4.6: Plot 1

Nella seguente figura possiamo vedere la generazione del tracciato a forma di otto su Gazebo.

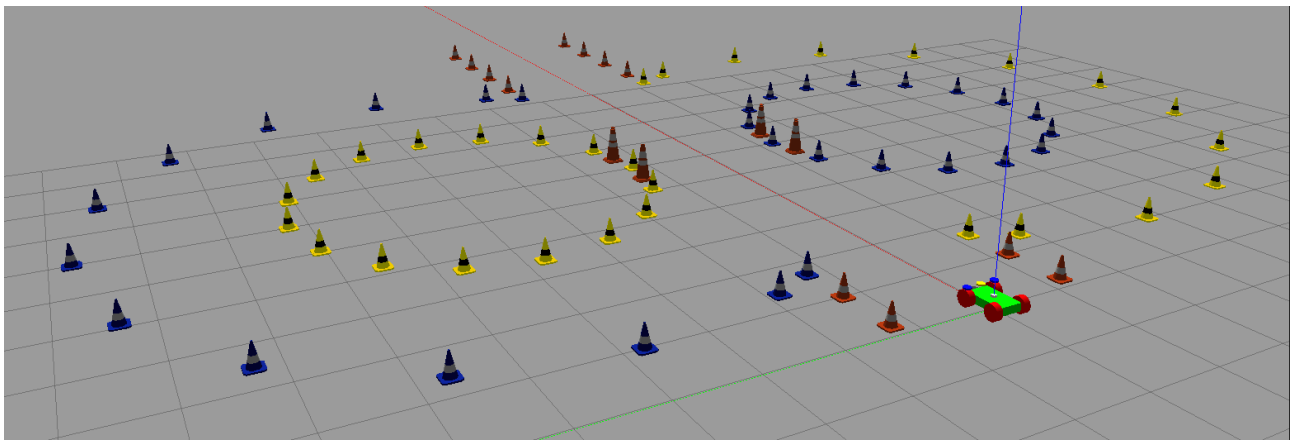

Figura 4.7: Tracciato a otto

4.4 – Random path

Infine, andiamo ad esaminare il tracciato con curve casuali, un percorso decisivamente più complesso rispetto ai precedenti, che ci permette di visualizzare il comportamento del nostro robot in condizioni di movimento imprevedibile e variabile. Tutto ciò è necessario al fine di verificare l'efficacia degli algoritmi di guida autonoma, in quanto verranno eseguite dal nostro robot traiettorie non predefinite.

```
1  #!/usr/bin/env python
2  import numpy as np
3  from scipy.special import binom
4  import matplotlib.pyplot as plt
5  from scipy import interpolate
6  from scipy.interpolate import splprep, splev
7  import math
8  from scipy.spatial.distance import cdist
9  import rospy
10 import xml.etree.ElementTree as ET
11 from gazebo_msgs.srv import SpawnModel
12 from geometry_msgs.msg import Pose
```

Il codice si apre con l'importazione delle librerie necessarie per la creazione del tracciato.

```
14 def modify_and_read_sdf(filename, percentage):
15     #Funzione interna per la modifica della dimensione dei coni
16      def modify_sdf_scale(filename, percentage):
17          tree = ET.parse(filename)
18         root = tree.getroot()
19
20         #Trova l'elemento <scale> e lo modifica
21         for scale_elem in root.iter('scale'):
22             scale_elem.text = f"{1*(percentage/100)}
               {1*(percentage/100)}   {1*(percentage/100)}"
23
24         #Salva il file SDF modificato
25         tree.write(filename, encoding="UTF-8", xml_declaration=True)
26
27      #Modifica il file SDF
```

```
28        modify_sdf_scale(filename, percentage)

30        #Leggi il contenuto del file SDF modificato
31        with open(filename, 'r') as file:
32            sdf_content_modified = file.read()
33
34        return sdf_content_modified
```

Successivamente, troviamo di nuovo la funzione *modify_and_read_sdf*, che svolgerà lo stesso compito descritto in precedenza.

```
37 #Classe per rappresentare un segmento di curva Bezier
38  class Segment():
39        def __init__(self, p1, p2, angle1, angle2, **kw):
40            self.p1 = p1; self.p2 = p2
41            self.angle1 = angle1; self.angle2 = angle2
42            self.numpoints = kw.get("numpoints", 100)
43            r = kw.get("r", 0.3)
44            d = np.sqrt(np.sum((self.p2-self.p1)**2))
45            self.r = r*d
46            self.p = np.zeros((4,2))
47            self.p[0,:] = self.p1[:]
48            self.p[3,:] = self.p2[:]
49            self.calc_intermediate_points(self.r)
50
51        def calc_intermediate_points(self,r):
52            self.p[1,:] = self.p1 +
               np.array([self.r*np.cos(self.angle1),
53            self.r*np.sin(self.angle1)])
54            self.p[2,:] = self.p2 +
               np.array([self.r*np.cos(self.angle2+np.pi),
55            self.r*np.sin(self.angle2+np.pi)])
56           self.curve = bezier(self.p,self.numpoints)
```

La classe *segment* è progettata per la rappresentazione di un curva di Bézier. In questa classe troviamo 7 attribuiti:

- *p1* e *p2:* indicano i punti di inizio e di fine del segmento;
- *angle1* e *angle2:* indicano gli angoli dei punti di inizio e di fine del segmento, vengono utilizzati per determinare la curvatura del segmento;
- *numpoints:* indicano il numero di punti che vanno a comporre la curva di Bézier;
- *r:* indica il fattore di curvatura del segmento, calcolato effettuando la distanza euclidea tra *p1* e *p2;*
- *p*: indica una matrice 4x2 che rappresenta i punti di controllo della curva di Bézier.

All'interno della nostra classe vengono definite delle funzioni, chiamate *metodi* in gergo tecnico:

- *__init__:* inizializza tutti gli attributi presenti nella nostra classe;
- *calc_intermediate_points:* calcola i punti intermedi necessari per definire la curva di Bézier, utilizzando il fattore di curvatura *r* e gli angoli di ingresso e di uscita.

```
60 #Funzione per calcolcare una curva Bezier
61  def bezier(points, num=200):
62      N = len(points)
63      t = np.linspace(0, 1, num=num)
64      curve = np.zeros((num, 2))
65      for i in range(N):
66          curve += np.outer(bernstein(N - 1, i, t), points[i])
67      return curve
```

La funzione *bezier* è utilizzata per calcolare una curva di Bézier tramite un insieme di punti di controllo. Questa funzione accetta 2 parametri:

- *points:* è un array numpy di dimensione (N,2), dove N indica il numero di punti di controllo;

- *num:* indica il numero di punti che verranno calcolati lungo la curva, di default sono 200.

Inizialmente, calcoliamo il numero di punti di controllo e creiamo una lista *t* di parametri spaziati uniformemente tra 0 e 1, che serviranno a determinare la posizione dei punti lungo la curva. Successivamente, inizializziamo una lista vuota *curve* per memorizzare i punti della curva. Tramite un ciclo che itera per ogni punto di controllo *i*, andiamo a calcolare i coefficienti di Bernstein per il punto di controllo corrente. Questi coefficienti vengono moltiplicati alle coordinate x e y del punto di controllo, infine i risultati delle moltiplicazioni verranno sommati alla lista *curve*, aggiornando la curva di Bézier con le coordinate calcolate.

```
69  #Funzione per ottenere una curva da una serie di punti
70  def get_curve(points, **kw):
71      segments = []
72      for i in range(len(points)-1):
73          seg = Segment(points[i,:2], points[i+1,:2],
             points[i,2],points[i+1,2],**kw)
74          segments.append(seg)
75      curve = np.concatenate([s.curve for s in segments])
76      return segments, curve
```

La funzione *get_curve* è utilizzata per generare una curva Bézier composta da una serie di segmenti. Questa funzione accetta due parametri:

- *points:* indica una lista bidimensionale di punti che rappresenta le coordinate x y e gli angoli dei punti di controllo. Possiamo dire che ogni punto ha la seguente forma (x, y, angolo);

- ***kw:* indica un dizionario di argomenti opzionali che può essere utilizzato dalla classe *segment* per inizializzare o configurare altri parametri.

Inizialmente, andiamo a creare un lista vuota *segments* che andrà a contenere i segmenti della curva Bézier. Tramite un ciclo for che itera attraverso i punti di controllo, per ogni coppia di punti consecutiva viene creata un'istanza della classe *segment*, la quale calcola i punti intermedi necessari per definire il segmento di curva, utilizzando le coordinate dei punti di controllo e gli angoli di curvatura. Una volta che tutti i segmenti sono stati creati, essi vengono concatenati tra di loro per formare una curva continua. Infine, la funzione ci restituirà sia la curva che i segmenti.

```
78 #Funzione per ordinare una serie di punti in senso orario
79  def ccw_sort(p):
80      d = p-np.mean(p,axis=0)
81      s = np.arctan2(d[:,0], d[:,1])
82      return p[np.argsort(s),:]
```

La funzione *ccw_sort* viene utilizzata per ordinare i punti in senso orario. Tale ordinamento è necessario e viene utilizzato molto spesso in vari algoritmi di geometria computazionale per garantire che i punti vengano processati con un ordine coerente. Questa funzione accetta un argomento *p* che indica la nostra lista di coordinate dei punti.
Inizialmente, la funzione calcola il centroide dei punti forniti. Esso rappresenta il punto medio di tutte le coordinate. Successivamente, la funzione sottrae il centroide da ciascun punto per ottenere un nuovo insieme di vettori che viene inserito nella lista *d*. La funzione quindi calcola l'angolo polare di ogni vettore rispetto al centroide e ordina i punti rispetto a questi angoli. Infine, ritorna la lista di punti ordinati.

```
84  #Funzione per ottenere una curva Bezier con punti equidistanti
85  def get_bezier_curve(a, rad=0.2, edgy=0):
86
87      p = np.arctan(edgy)/np.pi+.5
88      a = ccw_sort(a)
89      a = np.append(a, np.atleast_2d(a[0,:]), axis=0)
90      d = np.diff(a, axis=0)
91      ang = np.arctan2(d[:,1],d[:,0])
92      f = lambda ang : (ang>=0)*ang + (ang<0)*(ang+2*np.pi)
93      ang = f(ang)
94      ang1 = ang
95      ang2 = np.roll(ang,1)
96      ang = p*ang1 + (1-p)*ang2 + (np.abs(ang2-ang1) > np.pi )*np.pi
97      ang = np.append(ang, [ang[0]])
98      a = np.append(a, np.atleast_2d(ang).T, axis=1)
99      s, c = get_curve(a, r=rad, method="var")
100     x,y = c.T
101     return x,y, a
```

La funzione *get_bezier_curve* è utilizzata per generare curve di Bézier ricevendo in input una lista di punti. Questa funzione accetta 3 parametri:

- *a:* indica la lista di punti ricevuta come input;
- *rad:* indica il raggio di curvatura per determinare la rotondità delle curve;
- *edgy:* parametro utilizzato per determinare la spigolosità delle curve, più alto è il suo valore più la curva sarà spigolosa.

Inizialmente, andiamo a calcolare tramite la prima espressione il parametro *p*, che ci permette di determinare la spigolosità della curva risultante. Successivamente, andiamo a ordinare i punti ricevuti in input con la funzione *ccw_sort.*
Una volta ordinati i punti, il primo punto della lista viene aggiunto alla fine della lista in modo da ottenere una curva chiusa. La funzione successivamente, calcola i vettori di spostamento *d*, tramite i quali determinerà gli angoli di ogni segmento rispetto all'asse x attraverso *np.arctan2(d[:,1],d[:,0])*.

Per garantire che tutti gli angoli siano compresi tra 0 e 2π viene applicata una funzione di normalizzazione. Successivamente, gli angoli vengono mediati utilizzando il parametro *p*. Gli angoli calcolati vengono inseriti nella lista di punti fornita in input, come terza colonna, formando quindi una nuova lista *a* che contiene le coordinate x y e gli angoli associati. A questo punto, viene utilizzata la funzione *get_curve* per generare la curva di Bézier.

Infine, la nostra funzione legge le coordinate della curva generata e li restituisce insieme agli angoli associati.

```
105  #Funzione per ottenere punti casuali su una curva
106  def get_random_points(n=5, scale=0.8, mindst=None, rec=0):
107      mindst = mindst or .7/n
108      a = np.random.rand(n,2)
109      d = np.sqrt(np.sum(np.diff(ccw_sort(a), axis=0), axis=1)**2)
110      if np.all(d >= mindst) or rec>=200:
111          return a*scale
112      else:
113          return get_random_points(n=n, scale=scale, mindst=mindst,
             rec=rec+1)
114
115  bernstein = lambda n, k, t: binom(n,k)* t**k * (1.-t)**(n-k)
```

La funzione *get_random_points* è utilizzata per ottenere un insieme di punti casuali in una regione specifica, garantendo che i punti trovati siano distribuiti con coerenza e che la loro distanza minima sia sempre rispettata. Questa funzione accetta 4 parametri:

- *n:* indica il numero di punti casuali da generare;
- *scale:* indica l'ampiezza della regione in cui saranno distribuiti i punti;
- *mindst:* indica la distanza minima che deve essere mantenuta tra ogni coppia di punti. Nella nostra funzione il parametro non è impostato, quindi verrà impostato automaticamente come *0.7/n.*
- rec: è un contatore che serve a prevenire loop infiniti nel caso in cui la distanza minima dei punti non può essere soddisfatta.

Inizialmente, viene creata un array *a* di dimensione *nx2* contenente coordinate casuali distribuite uniformemente tra 0 e 1. Successivamente, viene chiamata la funzione *ccw_sort* per ordinare questi punti.
Una volta ordinati i punti, la funzione calcola le distanze tra i punti successivi nel percorso e verifica se tutte le distanze sono maggiori o uguali alla distanza minima *mindst.* Se tutte le distanze soddisfano la distanza minima, i punti vengono scalati secondo il parametro *scale* e ritornati dalla funzione.

Se invece una o più distanze non soddisfano la distanza minima, la funzione viene richiamata ricorsivamente per rigenerare i punti. Ad ogni chiamata il parametro *rec* verrà incrementato per tenere conto dei tentativi effettuati. Se il parametro supererà il valore limite (200) , la funzione interromperà le chiamate e ritornerà i punti generati fino a quel momento, anche se non soddisfano la distanza minima.

```
116  #Funzione per generare la traiettoria della pista
117  def generate_track(equidistant_u, equidistant_point_samples, tck,
         distance=1.8):
118      points_list_right = []
119      points_list_left = []
120      orange = True
121      for i in range (len(equidistant_u[0])-1):
122          u0 = equidistant_u[0][i]
123          x0 = equidistant_point_samples[0][0][i]
124          y0 = equidistant_point_samples[1][0][i]
125          dx,dy = interpolate.splev(u0,tck,der=1)
126          der = dy/dx
127          if(der >= 0):
128              x1 = x0 - distance * np.sin(np.arctan(der))
129              y1 = y0 + distance * np.cos(np.arctan(der))
130              x2 = x0 + distance * np.sin(np.arctan(der))
131              y2 = y0 - distance * np.cos(np.arctan(der))
132          elif(der < 0):
133              x1 = x0 + distance * np.sin(abs(np.arctan(der)))
134              y1 = y0 + distance * np.cos(abs(np.arctan(der)))
135              x2 = x0 - distance * np.sin(abs(np.arctan(der)))
```

```
136             y2 = y0 - distance * np.cos(abs(np.arctan(der)))
137         if orange == True:
138             plt.plot(x1,y1, marker="o", markersize=5,
                 markeredgecolor="orange",  markerfacecolor="orange")
139             plt.plot(x2,y2, marker="o", markersize=5,
                 markeredgecolor="orange",  markerfacecolor="orange")
140             points_list_left.append([x1,y1])
141             points_list_right.append([x2,y2])
142             orange = False
143         else:
144             if(dx < 0):
145                 plt.plot(x1,y1, marker="o", markersize=5,
                    markeredgecolor="blue",  markerfacecolor="blue")
146                 plt.plot(x2,y2, marker="o", markersize=5,
                    markeredgecolor="yellow",
                    markerfacecolor="yellow")
147                 points_list_left.append([x1,y1])
148                 points_list_right.append([x2,y2])
149              else:
150                 plt.plot(x1,y1, marker="o", markersize=5,
                     markeredgecolor="yellow",
                    markerfacecolor="yellow")
151                 plt.plot(x2,y2, marker="o", markersize=5,
                    markeredgecolor="blue",  markerfacecolor="blue")
152                 points_list_left.append([x2,y2])
153                 points_list_right.append([x1,y1])
154         i = i + 1
155     len_left = len(points_list_left)
156     len_right = len(points_list_right)
157     points = np.array(points_list_right)
158     points = np.append(points, points_list_left, axis=0)
159     return points, len_left, len_right, points_list_left,
        points_list_right
```

La funzione *generate_track* viene utilizzata per creare una rappresentazione sia fisica che visiva del tracciato, utilizzando punti equidistanti lungo la curva per posizionare i coni. Questa funzione accetta 4 parametri:

- *equidistant_u:* indica i parametri normalizzati per i punti equidistanti;

- *equidistant_point_samples:* contiene le coordinate dei punti equidistanti;

- *tck:* indica la spline interpolante;

- *distance:* indica la distanza utilizzata per determinare i punti laterali del tracciato rispetto alla sua traiettoria.

Inizialmente, andiamo a definire due liste vuote *points_list_right* e *points_list_left*, che verranno utilizzate per memorizzare rispettivamente i punti di destra e sinistra della pista. Inoltre, viene inizializzata una variabile booleana *orange* a True per alternare il colore dei coni nella visualizzazione della pista tramite plot.

Successivamente, la funzione entra in un ciclo *for* che itera attraverso i punti equidistanti. Per ogni punto viene estratto il parametro normalizzato *u0* e le coordinate *x0* e *y0.* Utilizzando la funzione di interpolazione verranno calcolate le derivate prime *dx* e *dy* necessarie per determinare la pendenza della curva.

Tramite delle condizioni, a secondo del segno della derivata, la funzione calcola le coordinate dei punti di destra e di sinistra rispetto al punto centrale *x0 y0.* Inoltre, la funzione alterna i colori dei marker utilizzati per visualizzare i coni lungo il tracciato tramite la variabile booleana definita precedentemente.

I punti calcolati vengono inseriti nelle nostre liste vuote. Al termine del ciclo, la funzione concatenerà le due liste per ottenere l'insieme completo dei punti che compongo il tracciato.

Infine, la funzione ritornerà una lista di tuple contenete tutti i punti, la lunghezza della lista dei punti di sinistra e destra, e le due liste di punti.

```
161  #Funzione per rimuovere punti ripetuti in una curva
162  def clean_curve(x, y):
163      to_be_deleted = []
164      for i in range(len(x)-1):
165          if(math.isclose(x[i], x[i+1])):
166              to_be_deleted.append(i)
167      j=0
168      for i in to_be_deleted:
169          x = np.delete(x, i-j)
170          y = np.delete(y, i-j)
171          j=j+1
172      return x, y
```

La funzione *clean_curve* è utilizzata per rimuovere punti ripetuti o troppo vicini tra di loro lungo la nostra curva. Questa funzione accetta due parametri *x* e *y*, due array che rappresentano le coordinate dei punti lungo la curva.

Inizialmente, andiamo a definire una lista vuota *to_be_deleted*, che verrà utilizzata per memorizzare i punti che devono essere cancellati. Tramite un ciclo *for* che itera attraverso i punti della curva, andiamo a confrontare ogni punto con il suo successivo. Se le coordinate di un punto sono uguali o quasi uguali a quelle del punto successivo, l'indice di quel punto viene aggiunto alla nostra lista vuota. Questo confronto viene fatto tramite la funzione *math.isclose.*

Una volta riempita la lista *to_be_deleted*, la funzione tramite un ciclo for che itera su tale lista, va a eliminare i punti corrispondenti degli array *x* e *y* .Infine, la funzione ci restiuisce i due array *x* e *y* senza punti duplicati o punti troppo vicini.

```
174  #Funzione per ottenere punti equidistanti lungo una curva
175  def equally_spaced_curve(x, y):
176      tck, _ = splprep([x, y], s=0)
177      n_points_spline = 70
178      u = np.linspace(0, 1, n_points_spline)
179      sampled_points = splev(u, tck)
180      sampled_points = np.stack(sampled_points, axis=-1)
181      inter_point_differences = np.diff(sampled_points, axis=0)
182      inter_point_distances = np.linalg.norm(inter_point_differences,
         axis=-1)
183      cumulative_distance = np.cumsum(inter_point_distances)
184      cumulative_distance /= cumulative_distance[-1]
185      tck_prime, _ = splprep([np.linspace(0, 1,
         num=len(cumulative_distance))], u=cumulative_distance, s=0,
         k=3)
186      equidistant_u = splev(u, tck_prime)
187      equidistant_point_samples = splev(equidistant_u, tck)
188      equidistant_u[0] = np.delete(equidistant_u[0], 0, 0)
189      equidistant_point_samples[0] =
         np.delete(equidistant_point_samples[0], 0, 1)
190      equidistant_point_samples[1] =
         np.delete(equidistant_point_samples[1], 0, 1)
191      return tck, equidistant_u, equidistant_point_samples
```

La funzione *equally_spaced_curve è* utilizzata per ottenere punti equidistanti lungo una curva. Questa funzione accetta due parametri *x* e *y* che rappresentano le coordinate della curva Bézier.

Inizialmente, tramite la funzione *splprep* calcoliamo la rappresentazione parametrica della spline che approssima tutti i punti forniti dai nostri due parametri. Questo passaggio ci restituisce *tck*.

Successivamente, creiamo un array *u* che rappresenta un insieme di punti equispaziati tra 0 e 1, utilizzati per campioanre le spline, in modo da ottenere *sampled_points*, ovvero un insieme di punti che vanno ad approssimare la nostra curva iniziale. Utilizzando i punti

campionati, la funzione calcola le distanze euclidee tra questi punti, che serviranno per ottenere una mappatura che rappresenta la distanza equidistante tra i punti lungo la curva.

Infine con questa mappatura otterremo i punti equidistanti lungo la curva di Bézier, che ci verranno restiuiti dalla nostra funzione.

```
194 #Funzione per richiamare la funzione di spawn
195  def spawn(name, x, y, z, model_content):
196      spawn_model =
         rospy.ServiceProxy('steer_bot/gazebo/spawn_sdf_model',
         SpawnModel)
197      initial_pose = Pose()
198      initial_pose.position.x = x
199      initial_pose.position.y = y
200      initial_pose.position.z = z
201      response = spawn_model(name, model_content, "", initial_pose,
           "world")
202      rospy.loginfo(response.status_message)
204  #Funzione che spawna i coni su Gazebo
205  def node():
206      rospy.init_node('spawn_model')
207      rospy.wait_for_service('steer_bot/gazebo/spawn_sdf_model')
208
209      try:
210      # Genera i coni
211      for i, (x, y) in enumerate(points_list_left):
212          if i == 0:
213              spawn(f'pointO_0', x, y, 0.3,
                  MODEL_CONTENT_CONE_ORANGE_BIG)
214          spawn(f'pointY_{i}', x, y, 0.3, MODEL_CONTENT_CONE_YELLOW)
215      for i, (x, y) in enumerate(points_list_right):
216          if i == 0:
217              spawn(f'pointO_1', x, y, 0.3,
                  MODEL_CONTENT_CONE_ORANGE_BIG)
218          spawn(f'pointB_{i}', x, y, 0.3, MODEL_CONTENT_CONE_BLUE)
219
220      except rospy.ServiceException as exc:
221              rospy.logerr("Error during service invocation: %s",
                str(exc))
```

Per ultima funzione troviamo *node,* responsabile dello spawn dei coni. Tramite due cicli *for* verranno iterate le coordinate sia del lato destro che del lato sinistro del percorso, in modo da spawnare reciprocamente i coni blu e i coni gialli. Inoltre, all'interno dei due cicli troveremo una condizione *if==0* per gestire il posizionamento dei coni di partenza arancioni grandi.

```
223  #Funzione MAIN
224  if __name__ == "__main__":
225      while True:
226      percentage= int(input("Inserisci un valore per ridimensionare il
         percorso (0-100%): "))
227            if 0 <= percentage <= 100:
228                break
229            else:
230            print("Percentuale invalida. Perfavore inserisci una
                percentuale tra 0 e 100")
231      #Modifica e lettura dei file SDF
232      MODEL_CONTENT_CONE_YELLOW =
         modify_and_read_sdf('./steer_bot/cone_yellow/model.sdf',
         percentage)
233      MODEL_CONTENT_CONE_BLUE =
         modify_and_read_sdf('./steer_bot/cone_blue/model.sdf',
         percentage)
234      MODEL_CONTENT_CONE_ORANGE_BIG =
         modify_and_read_sdf('./steer_bot/cone_orange_big/model.sdf',
         percentage)
```

Nel blocco del *main*, inizialmente andremo a consentire all'utente di definire la percentuale di ridimensionamento del percorso e dei coni. Dopodichè, andiamo a leggere il contenuto dei nostri modelli SDF.

```
235     #Caratteristiche del percorso
236     n = 7
237     rad = 0.3
238     edgy = 0.1
239     scale = 100*(percentage/100)
240     mindst = 150*(percentage/100)
241     distance = 2.5*(percentage/100)
242     #Genera una curva casuale
243     a = get_random_points(n, scale, mindst, 0)
244     print("Questa è la mia curva: " + str(a) + "\n")
245
246     x, y, _ = get_bezier_curve(a,rad, edgy)
247     x, y = clean_curve(x, y)
```

Successivamente, andiamo a definire le caratteristiche del percorso e a generare la nostra curva casuale, richiamando le funzioni specifiche. Infine, tramite dei plot andiamo a vedere passo per passo come viene generata la nostra curva e successivamente andiamo a richiamare la funzione *node.*

```
249       #Traccia i punti di partenza e la curva di Bézier
250       plt.figure(figsize=(8, 8))
251       plt.axis("equal")
252       plt.plot(*zip(*a), 'go')
253       plt.plot(x,y, 'green', linestyle="-")
254       xmin, xmax, ymin, ymax = plt.axis()
255       plt.clf()
256       plt.xlim([xmin - 20, xmax + 20])
257       plt.ylim([ymin - 20, ymax + 20])
258       plt.plot(*zip(*a), 'go') #points
259       plt.plot(x,y, 'green', linestyle="-") #curve
260       plt.show(block = False)
261       input("Press Enter to continue...")
```

Come possiamo vedere in figura vengono tracciati i punti iniziali per poi andare a generare la curva di Bézier.

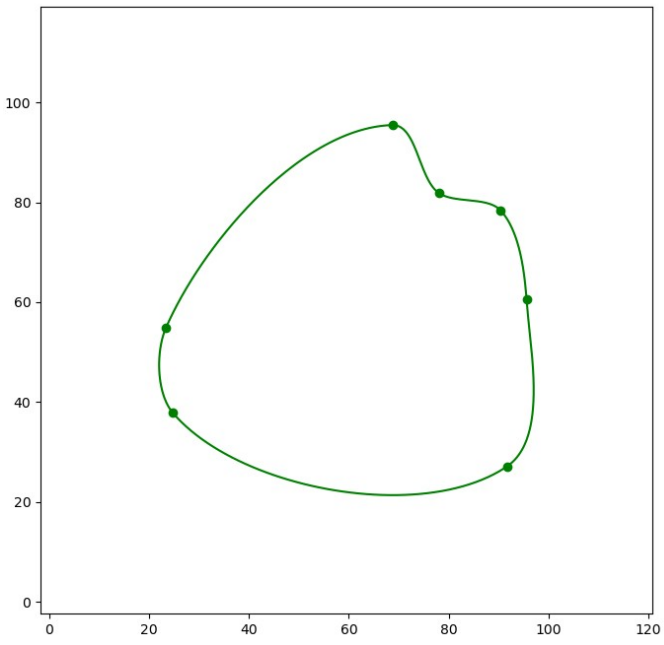

Figura 4.8: Plot 2

```
263     #Ottiene i punti equidistanti lungo la curva del percorso con
         equally_space_curve e li plotta
264     tck, equidistant_u, equidistant_point_samples =
         equally_spaced_curve(x, y)
265     xmin, xmax, ymin, ymax = plt.axis()
266     plt.xlim([xmin, xmax])
267     plt.ylim([ymin, ymax])
268     plt.plot(equidistant_point_samples[0][0][0],
         equidistant_point_samples[1][0][0], marker="x", markersize=10,
         markeredgecolor="red", markerfacecolor="blue")
269     plt.plot(equidistant_point_samples[0],
         equidistant_point_samples[1],   marker="o", markersize=2,
         markeredgecolor="blue",    markerfacecolor="blue")
270     plt.show(block = False)
271     input("Press Enter to continue...")
```

Successivamente, lungo l'intera curva andiamo ad ottenerci l'insieme di punti equidistanti tramite la funzione descritta in precedenza.

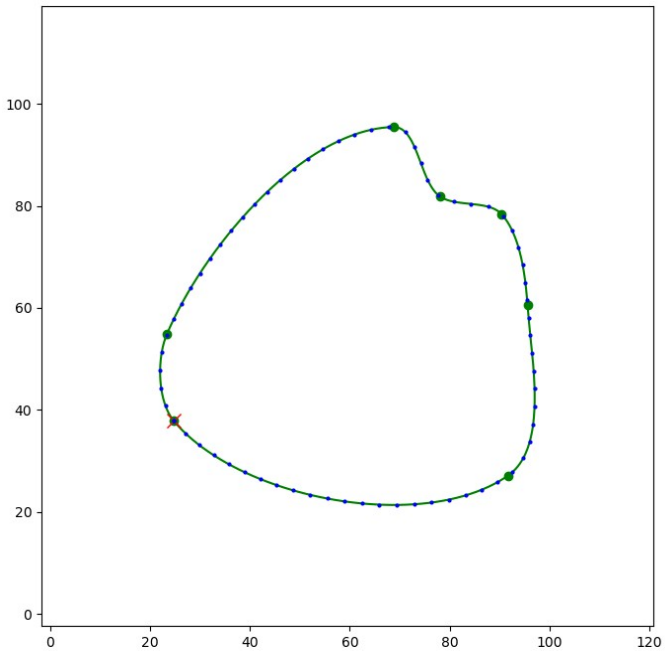

Figura 4.9: Plot 3

```
273         #Genera e plotta i punti dei coni
274          points, len_left, len_right, points_list_left,
            points_list_right =   generate_track(equidistant_u,
            equidistant_point_samples, tck, distance)
275          plt.show(block = False)
276          input("Press Enter to continue...")
278          node()
```

Infine, per ogni punto del percorso destro e sinistro andiamo a visualizzarci reciprocamente i coni blu e gialli che delimitano il tracciato.

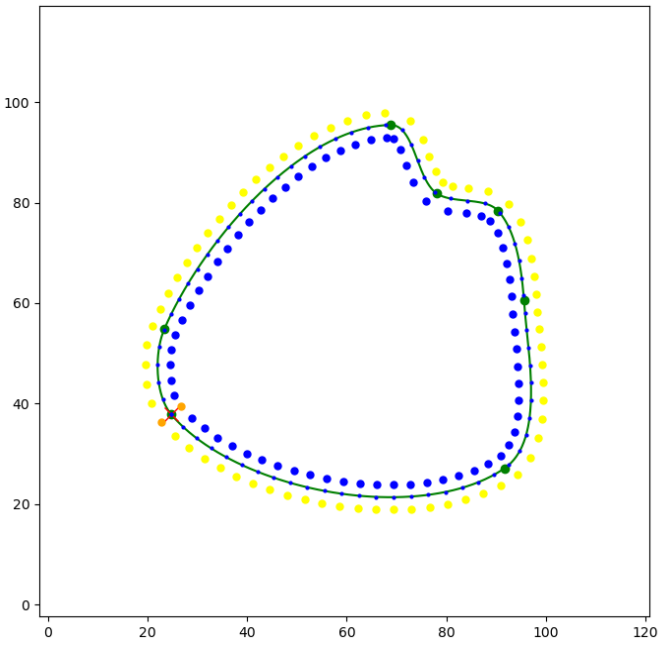

Figura 4.10: Plot 4

Nella seguente figura possiamo vedere la generazione di un tracciato casuale su Gazebo.

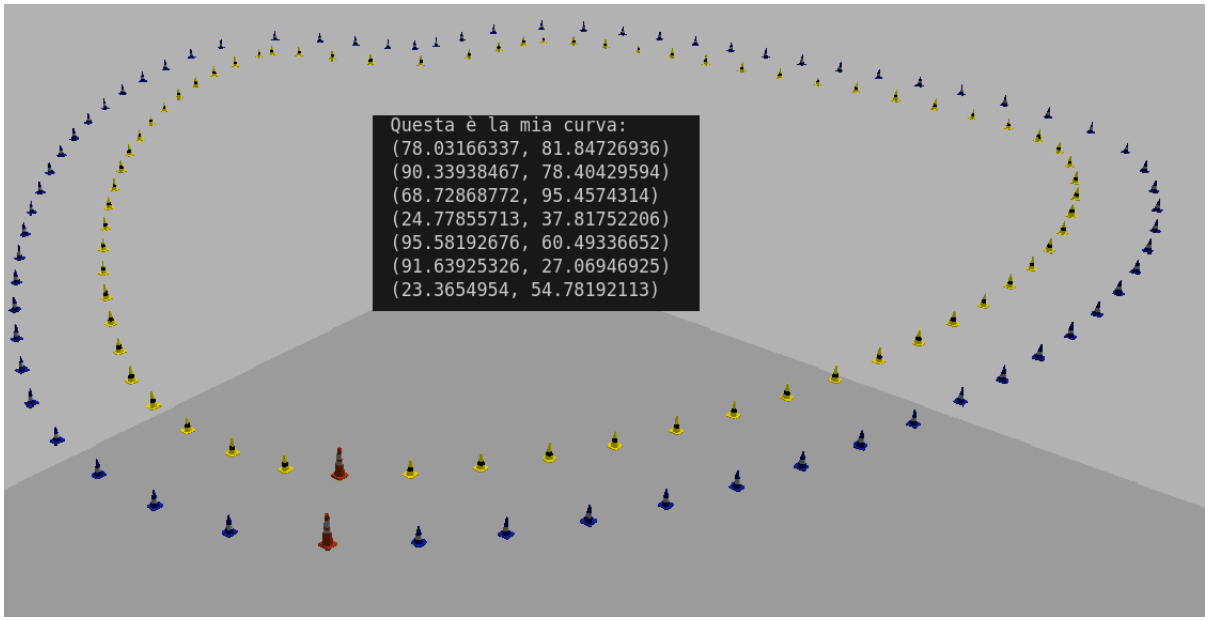

Figura 4.11: Tracciato casuale

Capitolo 5

Conclusione

Il lavoro di questa tesi ha rappresentato un processo impegnativo e stimolante, pieno di sfide e opportunità di miglioramento. Nel corso del progetto, abbiamo studiato e applicato un insieme di concetti matematici fondamentali per la realizzazione dei tracciati simulativi, tra cui l'uso delle curve di Bézier e i polinomi di Bernstein. Abbiamo testato il nostro algoritmo di guida autonoma sui tre tracciati realizzati. Questo test ci ha permesso di verificare l'efficacia e la robustezza di tale script, rivelando risultati molto positivi. Inoltre, abbiamo testato anche gli algoritmi per la percezione dei coni utilizzando un sensore LIDAR, ottenendo rilevazioni accurate dei coni in modo da migliorare la capacità di guida del veicolo in autonomo.

Nonostante i risultati ottenuti siano positivi, il progetto offre ampissimi margini di miglioramento e ulteriori possibilità di sviluppo. Di seguito, sono elencate alcuni sviluppi futuri che potrebbero essere implementati per perfezionare il tutto:

- *Sviluppo di nuovi tracciati simulativi*: Creare nuovi scenari e tracciati simulativi che vanno a rappresentare situazioni stradali più complesse e reali, per testare ulteriormente le capacità del nostro veicolo.

- *Integrazione di sensori aggiuntivi*: Aggiungere e integrare nuovi sensori, come telecamere e radar, per migliorare la percezione dell'ambiente circostante del nostro veicolo e aumentare la precisione di rilevazione degli ostacoli.

- *Test in ambienti reali:* Effettuare test sul campo in ambienti reali per validare i risultati ottenuti in simulazione.

Questa tesi rappresenta non solo un contributo al progetto ZED, ma anche un passo avanti nella ricerca e sviluppo di veicoli autonomi elettrici, offrendo soluzioni che possono essere applicate in contesti universitari e industriali.

Riferimenti bibliografici

[1] Gazebo, il simulatore che permette di sviluppare robot
URL: https://systemscue.it/gazebo-il-simulatore-che-permette-di-sviluppare-robot/8437/

[2] Cos'è il Robot Operating System e come può essere significativo nell'industria 4.0
URL: https://farelettronica.it/cose-il-robot-operating-system-ros-e-come-puo-essere-significativo-nellindustria-4-0/

[3] Github ZED Unime
URL: https://github.com/ZancleEDrive

[4] Python: cos'è, a cosa serve e come programmare con Python
URL: https://www.ai4business.it/intelligenza-artificiale/python-tutto-cio-che-ce-da-sapere-su-uno-dei-piu-popolari-linguaggi-di-programmazione/

[5] NumPy: the absolute basics for beginners
URL: https://numpy.org/doc/stable/user/absolute_beginners.html

[6] xml.etree.ElementTree - The ElementTree XML API
URL: https://docs.python.org/3/library/xml.etree.elementtree.html

[7] Cos'è Matplotlib
URL: https://www.geekandjob.com/wiki/matplotlib

[8] Cos'è Sympy?
URL: https://www.andreaminini.com/python/moduli-python/sympy

[9] Cos'è SciPy
URL: https://www.geekandjob.com/wiki/scipy

[10] Le funzioni trigonometriche: concetti fondamentali
URL: https://www.letuelezioni.it/blog/funzioni-trigonometriche-concetti-fondamentali#:~:text=Le%20sei%20funzioni%20trigonometriche%20principali,lato%20opposto%20e%20l'ipotenusa.

[11] Curve di Bézier
URL: http://www.mat.unimi.it/users/alzati/Geometria_Computazionale_98-99/apps/bezierizer/teoria.html