Visual-inertial Odometry using synthetic data

Davide Saia, 374446 Maria Pennicchi, 367301

Master's Degree in Robotics
Engineering Department
Prof. Paolo Valigi

***Abstract*—Visual inertial odometry, is one of the most used methods for making pose estimation, either in the aerial or ground scenario. The combination of both sensors can be made thanks to using the Kalman filter. It is not always simple to recover suitable data for testing, so in this project, we want to make visual-inertial odometry using synthetic data.**

***Index Terms*—visual-inertial odometry, Kalman filter, drivingScenario, synthetic data, trajectory, sensor fusion.**

I. Introduction

VISUAL-INERTIAL ODOMETRY represents a cornerstone technology in autonomous robotics for estimating the pose of a vehicle. By combining visual and inertial data, visual-inertial odometry (VIO) achieves improved robustness and accuracy in diverse operational contexts, including aerial and ground navigation. The integration of visual and inertial sensors through techniques like the Kalman filter allows for efficient sensor fusion, enhancing the reliability of pose estimation.

In this work, we explore the use of synthetic data for visual-inertial odometry, aiming to facilitate repeatable experiments and provide a modular framework for testing navigation algorithms, eliminating the need for a costly real-world setup. Synthetic data generation circumvents the challenges associated with obtaining suitable real-world datasets, enabling controlled evaluations and the testing of novel methodologies.

The project focuses on the utilization of MATLAB's drivingScenarioDesigner tool to create synthetic scenarios, define trajectories, and simulate sensor data. Both the smoothTrajectory and waypointTrajectory methods were employed to analyze their impact on pose estimation accuracy. These approaches were integrated with a sensor fusion technique using a Kalman filter to enhance the overall performance of the system. The methods were compared under varying levels of trajectory perturbation to evaluate the system's robustness. This study demonstrates the utility of synthetic data and emphasizes the role of sensor fusion in improving pose estimation. By integrating visual odometry and inertial measurements, our approach highlights the benefits of multiple sensors for achieving precise localization in dynamic scenarios.

II. State of the Art

The development of robust and accurate localization and navigation systems has been a focal point in robotics research. Recent advancements highlight the critical role of sensor fusion, synthetic datasets, and lightweight algorithms to address the challenges posed by dynamic environments and resource-constrained platforms.

Sensor fusion techniques are essential for enhancing pose estimation and object tracking accuracy. Troncoso et al. demonstrated the integration of stereo vision and inertial measurement unit (IMU) data to mitigate challenges posed by dynamic objects and environmental noise[1]. Their work highlights the role of Kalman filters in combining data from multiple sensors for robust pose estimation. Geneva et al. presented a two-stage system combining offline mapping with online localization to minimize drift and computational overhead. Their EKF-based framework achieves centimetre-level accuracy by fusing inertial, odometry, and Lidar data, showcasing its applicability in real-world scenarios[2]. Similarly, Dahal et al. proposed a hybrid sensor fusion architecture combining Lidar and Radar data for extended object tracking[3]. Their method improves the accuracy of obstacle state estimation, particularly in high-curvature road scenarios, by leveraging curvilinear road coordinates and Unscented Kalman Filters (UKF). Their work integrates road geometry into tracking routines, enabling robust state estimation even under occlusion or noisy sensor conditions. This provides detailed obstacle representations, including shape, orientation, and kinematics, which are critical for tasks like lane-keeping and trajectory planning.

In this last article, synthetic data generated through the MATLAB Driving Scenario Designer is essential for validating the algorithm prior to real-world trials, which are rarely feasible. The use of synthetic datasets has revolutionized the testing and development of autonomous systems, particularly in robotics. These datasets address key challenges of real-world data, such as high collection costs, complex labelling, and difficulty in simulating rare or hazardous scenarios. Song et al. highlights their utility in enabling reproducible testing of perception algorithms, such as semantic segmentation and depth estimation, in fully controlled environments[4]. Future efforts aim to enhance realism, expand scenario diversity, and incorporate dynamic interactions, driving the development of robust and trustworthy autonomous systems.

Finally, we want to mention an article regarding educational robotics. In fact, Pinto et al. demonstrated the value of using simulation-based scenarios alongside real-world experiments to teach advanced localization concepts using the Extended Kalman Filter (EKF)[5]. By integrating theoretical lessons with practical tasks on LEGO NXT robots, the study highlighted the importance of simulation tools like SimTwo, which allow students to explore and refine localization algorithms in controlled environments before transitioning to physical tests. This approach not only enhances the understanding of probabilistic methods such as EKF but also provides an accessible and scalable framework for teaching complex robotics concepts, bridging the gap between theory and hands-on application.

III. Project Objective

The aim of this project is to estimate the pose of a vehicle using synthetic data. To make odometry, we want to use the Kalman filter, which has the property of **sensor fusion**. The purpose of using synthetic data is to make repeatable experiments and generate a modular framework for navigation tools and test algorithms. For pose generation we used the **drivingScenarioDesigner** Matlab tool, which allows us to define either the scene where to make the estimation and specify the *ego Vehicle* pose, what we want to estimate. To make an estimation we use an **insfilterErrorState** object, already present in Matlab. We want to make Visual inertial Odometry because is used a lot in a lot of fields of application, and we want to focus on the importance of making better estimations thanks to the usage of multiple sensors. In our project, we want to compare the estimation made by the same filter but in two different situations:

- smoothTrajectory
- waypointTrajectory

Both provide a way to cross a desired path, but the way the trajectory is followed is the main difference, the second one is more perturbed.

IV. Experiment Description

In this section, we will describe how we generated synthetic data and how we used them to estimate poses with an error state Kalman filter. The importance of generation of synthetic data allows students, workers and great minds to evaluate and test filter or algorithm performance. In particular in Matlab that is possible with using of **drivingScenarioDesigner** tool.

A. drivingScenarioDesigner

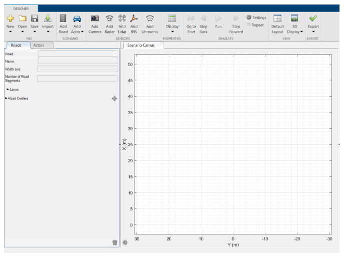
DrivingScenarioDesigner[6] is a tool Matlab that allows the creation of custom scenarios. The GUI is simple and easy to use. Write on the Matlab terminal the command:

```
>> drivingScenarioDesigner
```

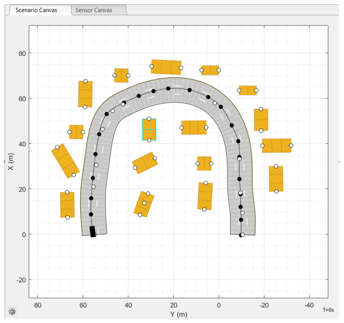
The main elements are:

- Roads
- Actors

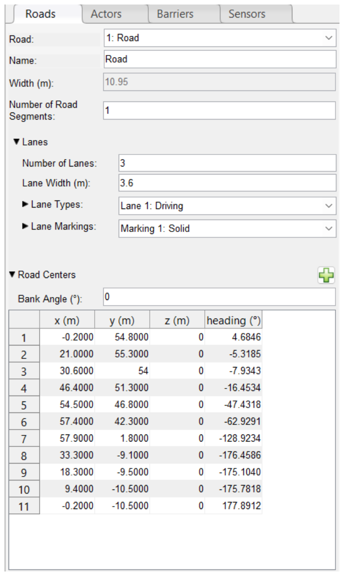
At the start, it is possible to add a road into a blank scenario canvas. It is possible to create roads with different shapes. As actors, the scenario deals with elements like cars, trucks or pedestrians, so it is possible to generate complex driving cases.



In our work, we generate a generic road with a curve to make a general trajectory as possible and as an ego vehicle a car, which initially has the shape of a parallelepiped. We took inspiration from the MathWorks web page[7]. To make a more complex scenario is possible to add barriers and guardrails. Everything in the scenario has a proper mesh, applicable with a flag in Matlab code. To create and modify roads, you need just



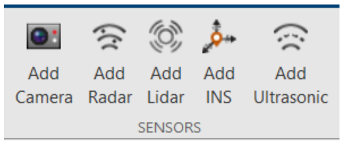
a click, to create a continuous road you have to click and the points generation goes on until the press of the enter button. In the side window, it is shown the coordinates of the road with the relative center, editable too. It is also possible to add lanes and make roads more realistic. An extraordinary possibility is



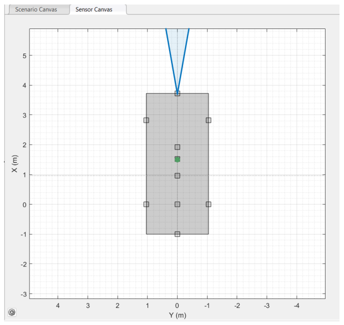
Road: 1: Road
Name: Road
Width (m): 10.95
Number of Road Segments: 1
▼ Lanes
Number of Lanes: 3
Lane Width (m): 3.6
► Lane Types: Lane 1: Driving
► Lane Markings: Marking 1: Solid
▼ Road Centers
Bank Angle (°): 0

	x (m)	y (m)	z (m)	heading (°)
1	-0.2000	54.8000	0	4.6846
2	21.0000	55.3000	0	-5.3185
3	30.6000	54	0	-7.9343
4	46.4000	51.3000	0	-16.4534
5	54.5000	46.8000	0	-47.4318
6	57.4000	42.3000	0	-62.9291
7	57.9000	1.8000	0	-128.9234
8	33.3000	-9.1000	0	-176.4586
9	18.3000	-9.5000	0	-175.1040
10	9.4000	-10.5000	0	-175.7818
11	-0.2000	-10.5000	0	177.8912

to set waypoints, the waypoints are scenario features, and their generation is similar to roads but it is important to insert an actor before, to insert forward waypoints, right-click on the actor inserted in the previous step and click on "*Add forward waypoints*", and as roads. click on the canvas to generate the points to follow and press the enter button when the desired trajectory is generated. To make easier the custom scenario it is possible also to edit every point of the canvas, in the side window, click on the point chosen and modify the coordinates of the point. After creating the scenario, it is possible to add other vehicles, and sensors and configure them using a proper side window. There are four type of sensors:



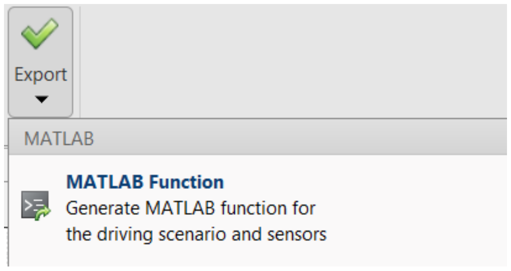
Each sensor is customizable and it is possible to set where the sensor is located on the vehicle. This choice is taken in the adding phase, but it is also possible to edit the location in a secondary phase. To choose which sensor to edit, just click on



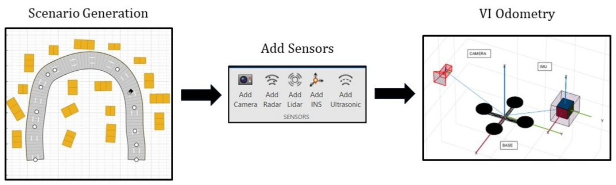
the side window in the **Sensors** topic, it is possible to configure the settings, parameters and placement. Each added sensor can be disabled, to test and evaluate algorithms or performance on different sensors.

B. *Project Function*

In our project, we use a lot the driving scenario function and edit the helper function for our scope. In particular *Designer* allows exporting the generated scenario as a Matlab function, the function supplies Matlab code with a helpful description to replicate the scenario made in the Designer. The functions in the code describe road and vehicle placement with also the waypoints the vehicle has to follow. After the tool details,


Export
MATLAB
MATLAB Function
Generate MATLAB function for the driving scenario and sensors

in our project, we used only some parts of the functions supplied, we custom that for our scope, in particular, we used synthetic data for making odometry estimation using a **camera** sensor, and **imu** sensor. The project is based on *Visual-inertial Odometry*. In this section, we describe the functions created and their use. The architecture of the project can be divided into three main parts:



*1) **createDrivingScenario()**:* this function has the task of using the scenario generated by the tool Matlab *drivingScenarioDesigner*, and define also the placement and the waypoints of the **egoVehicle**. It is also defined as an **estVehicle**, this vehicle is the same as the car we want to estimate, but the difference is in the updating of its pose that is linked to the filter. At the end of the function we set the trajectory used for filter estimation, we used the function smoothTrajectory[8] provided by the *drivingScenarioDesigner*, the function takes as input:

- vehicle object
- waypoints
- speed

The speed array dimension is $1 \times N$ where N is the number of waypoints chosen for the desired path.
To compare the trajectory made by *drivingScenario* we used the function waypointTrajectory[9]. The **waypointTrajectory** takes as parameters:

- waypoints
- sampleRate
- Velocities
- ReferenceFrame
- JerkLimit

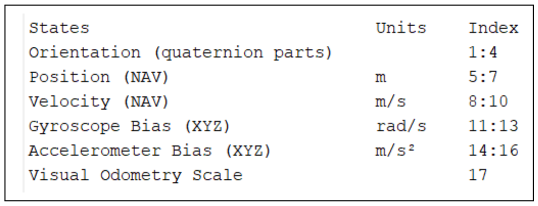
The waypoints are passed as a matrix $N \times 3$ where N is the number of waypoints chosen in the generation phase. Waypoints are editable, in our work we focused on 2D coordinates and the last columns are all zeros. The **JerkLimit** is passed as alternatives to **TimeOfArrival** or **GroundSpeed**, as jerk Matlab encode how the acceleration changes in an instant.

*2) **createSensors()**:* this function is a custom function. We do not use the sensors provided by the scenario because they are difficult to work with, less customizable, and require estimating the car's position after the experiment is complete. Instead, we create custom parameters using existing MATLAB functions. In particular, the function takes as a parameter the scenario that has the feature of **SampleTime**, because we can also simulate it. This feature is useful to calculate the sample rate that we set equal to 100. We define and set the visual odometry model parameters, as found on MathWorks[10]. These parameters model a feature-matching and tracking-based visual odometry system using a monocular camera. The scale parameter accounts for the unknown scale of subsequent vision frames of the monocular camera. The other parameters model the drift in the visual odometry reading as a combination of white noise and a first-order Gauss-Markov process. We define and set the second sensor that we used, the IMU sensor model containing an accelerometer and gyroscope using the *imuSensor* System object[11]. The sensor model contains properties to model both deterministic and stochastic noise sources. The property values set here are typical for low-cost MEMS sensors.

*3) **filterInitializer():*** In this work to make sensors fusion we use an insfilterErrorState[12]. The insfilterErrorState object implements sensor fusion of IMU, GPS, and monocular visual odometry (MVO) data to estimate pose in the NED (or ENU) reference frame. The filter uses a 17-element state vector to track the orientation quaternion, velocity, position, IMU sensor biases, and the MVO scaling factor. The insfilterErrorState object uses an error-state Kalman filter to estimate these quantities. The function has the task of initializing the filter element in particular:

- State
- StateCovariance

the function takes in input the filter object and the trajectory already defined in IV-B1, we extract the initial $[pos, orientation, velocity]$ from the trajectory and we use it to initialize the state of the filter. We also tuning the covariance based on gyroscope bias and visual-odometry scale factor, we tune as to low confidence scenario, indeed we set high values of covariance.


```
States                              Units    Index
Orientation (quaternion parts)               1:4
Position (NAV)                      m        5:7
Velocity (NAV)                      m/s      8:10
Gyroscope Bias (XYZ)                rad/s    11:13
Accelerometer Bias (XYZ)            m/s²     14:16
Visual Odometry Scale                        17
```

*4) **preallocateDataSimulation():*** At this point, we have created the scenario, and the sensors to estimate the pose of the car, we have also initialised the filter parameters, and we need to initialize data to set up the simulation loop in terms of structure and sample dimensions. The function takes as parameters:

- numImuSamples
- numCameraSamples

these parameters are used to set the row's dimensions, the first specification initializes the **ground truth** data and the filter data are given in the output. The second specification is important for loading the visual odometry output data. The aim of this function is to give in output the ground truth values and the estimations.

*5) **updatePose():*** in this job we want to plot either the simulation and the estimation made by the Kalman filter, to update the pose of the ground Truth vehicle and the estimated vehicle. We use this function, it takes as input:

- vehicle
- position
- velocity
- orientation

As the vehicle passes the object we want to update the pose, that in the simulation loop we update both, and so that we call this function twice. To update the corresponding information about the **pose** and **velocity**, we also take velocity because the **UpdatePlots** function from a prebuilt scenario moves the vehicle with respect to the last update poses.

*6) **visualOdometryModel():*** It is the corresponding function of the simulation **VO** model, we choose to simulate the estimation as a function of parameters, which we passed in input we the initial pose and orientation. In this function, the *drift* of the camera and the output parameters which are the estimated values of the vehicle pose are calculated. To calculate the drift we used this formula:

$$drift = randn(1, 3) * sigmaN + b;$$

where b is the bias computed as follow:

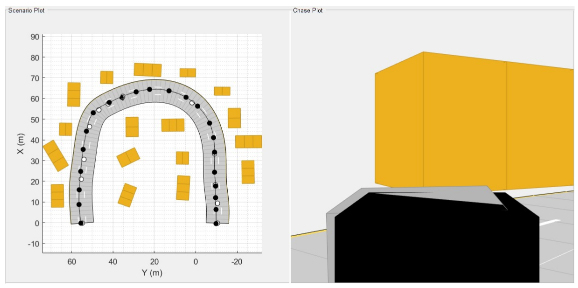
$$b = (1 - 1/tau). * b + randn(1, 3) * sigmaA;$$

We estimate the pose and the orientation as a function of scale factor and bias

$$posVO = scaleVO * pos + drift;$$
$$orientVO = orient;$$

C. Simulation Loop with smoothTrajectory

We want to simulate the scenario created in the **drivingScenarioDesigner** to set up and start the simulation we use the Matlab function **advance**, which starts the scene and places either the vehicles, the roads and the buildings. The time of simulation is set as a function of the speed of the *egoVehicle*. The simulation loop is based on the estimation of the ground truth pose, to recover the initial state, we use the function **state**, which takes as a parameter the vehicle that we want to take info about. The next step is to convert the motion quantities from the vehicle frame to the IMU frame and make the measure. We use the function **predict** to predict by the *insfilterErrorState*. We use the IV-B6 to estimate the estimation made by the camera and the use the function **fusemvo** to correct the filter estimation. In the end, the pose estimated can be recovered by the function **pose**, we use the pose estimated to update the visualization of the estimated vehicle pose and we plot it.



It is possible to set it using the meshes and improve the visualization, the meshes are provided by Matlab, but it is also possible to define it by yourself, with a mesh object. The plots used in our project are:

- plot
- chasePlot

In the first plot, we insert also the waypoint marker, which is crossed by the vehicles, in the second plot, we want to give a vision of the scene from the vehicle's point of view.

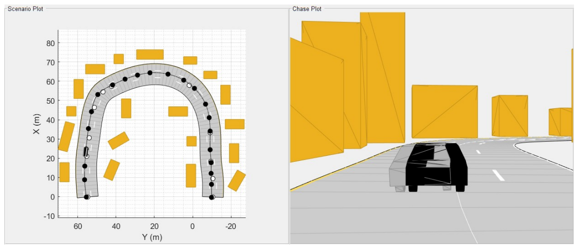


Fig. 1. Plot with meshes

D. Simulation Loop with waypointTrajectory

To set up the simulation loop, we had to set various parameters, especially:

- number of seconds to simulate
- number of samples of the sensors

We tested at different times of simulation; we chose to simulate over 120 seconds to give the vehicle to finish the entire path. The number of samples is determined by the **sampleRate** and the seconds of the simulation, which are about the number of IMU samples. For camera samples, we estimate around one sample per four IMU samples. After defining and allocating all structures we set the *loop*, which iterates on the number of IMU samples. At the start is generated the true trajectory is given by the **waypointTrajectory** in the IV-B1 function. We passed the initial pose to start generating gyroscope and accelerometer measures, and **predict** the next step based on the just calculated measures. The peculiarity of **insfilterErrorState** is to make a prediction on the ins measure and correct the estimation with the *VO* measure through the function **fusemvo**. The loop uses the **visualOdometryModel** IV-B6 to make the estimation and populate the estimation array made by the VO model. Suddenly the prediction made by the filter and the Visual odometry model, we exploit the function **fusemvo** which is specifically for the monocular visual odometry model, this function helps to correct the estimation of the filter and fuse the estimation of the two sensors. At this moment we can extract from the filter the estimated values and save them in the corresponding variables. We use the ground Truth values and the estimated one to update the pose of the two vehicles in the scenario and update the scene to show the fidelity of the estimation. We want to plot the scenario simulation during the pose updating either in the 2D case or the **chasePlot**[13]. The first plot gives a dimensionality vision of the scene and the target path we want the vehicle to follow.

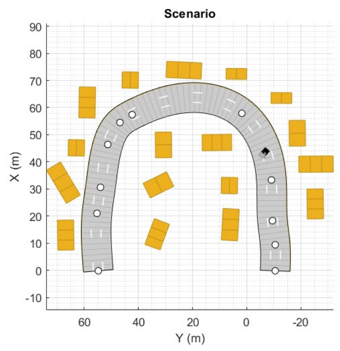


Fig. 2. Scenario plot

The second plot is from the vehicle's point of view, in brief as the vehicle moves.

Fig. 3. Chase plot

V. Issues

We met some issues during the project flow, the main three problems are:

1) create a generic scenario
2) make a coherent trajectory
3) correct tuning of the filter

A. Create Generic Scenario

We want to make a scene with roads and buildings as general as possible, we try a lot to modify and edit the roads generated following the trajectory that we made in the second phase. The Matlab tool IV-A provides several functions, with different scopes and purposes. In the project the aim is to make a simple road with a generic curve that can be crossed in two different ways, it is just to edit the vehicle placement and trajectory. It is a simple and efficient tool, we work a lot to understand how to edit single roads and palaces.

B. Trajectory

As described in IV-C, the **drivingScenarioDesigner** provides the trajectory made by the *smoothTrajectory* function. This function is simple to use, but editable only through the waypoints, and speed vector. We set the number of waypoints as a function of the shape of the roads and the speed as low as the simulation time desired.

$$speed = 5; \quad m/s$$
$$numWaypoints = 22;$$

We had more problems with the second method because to estimate the pose and generate a synthetic scenario we had to choose and design the right trajectory, the **drivingScenarioDesigner** tool gives as default a *smoothTrajectory* object, it is not possible to save in a variable and call its items, at least the trajectory given is an example of the path made by **waypointTrajectory**. The trajectory made by this function is a bit different, both take as input the waypoints array, but the peculiarity of this function it is also possible to edit the speed to follow the path and specify the *jerk* limit. The waypointTrajectory System object defines a trajectory that smoothly passes through waypoints. The trajectory connects the waypoints through an interpolation that assumes the gravity direction expressed in the trajectory reference frame is constant. The problem was which is the best number of waypoints and what velocities, we made a trade-off between them, because more waypoints to follow, more interpolation and lower speeds are necessary to make the trajectory as smooth as possible. We balanced the number of waypoints and we chose regardless of the shape of the roads, we did a lot of experiments, and we found the best solution with low speed and enough waypoints:

$$speed = 2; \ m/s$$
$$numWaypoints = 22;$$

C. Filter Tuning

The filter used in this project is a particular application of the Kalman filter sensor fusion property. In particular, this filter uses a combination of a vision sensor and an ins sensor, as the main prediction the filter exploits the inertial part and corrects the estimation with the Visual odometry side. The visual Odometry part is linked to the model defined in the IV-B6 section. The main problem met, was about the setting of the parameters of the **accelorometer** and the **gyroscope**. Both are defined by:

- Measumerent Range
- Resolution
- NoiseDensity

After several research, we found and set more generic parameters for a low-cost MEMS sensor. This type of choice is based on the data found. The solution applied was to make it more editable and maintain such modularity for the entire project flow. Knowing the parameters for other sensors, it is possible to modify the script and customize the simulation IV-B2.

VI. Results

In our work, we want to estimate the pose of a vehicle using synthetic data, from a generated scenario using the **drivingScenarioDesigner** Matlab tool. The Visual-Inertial Odometry, exploit the Kalman filter sensor fusion. The estimation made from the two sensors is weighted thanks to the corresponding covariance matrix and the state matrix. The prediction phase and the fusion phase are the most important part and the aim of this project. The peculiarity of Visual Odometry is the scale issue, the pose is estimated in the function of the feature detected and the drift caused by the usage of the vision sensor, indeed the estimation made by the camera is a bit similar to the combination of both, but overscaled because, with an only one vision sensor, the global scale is not possible to recover. The IMU sensor gives a better fidelity estimation and merge it with the camera the final estimation is the best, indeed we plot the difference between the two estimations and highlight them.

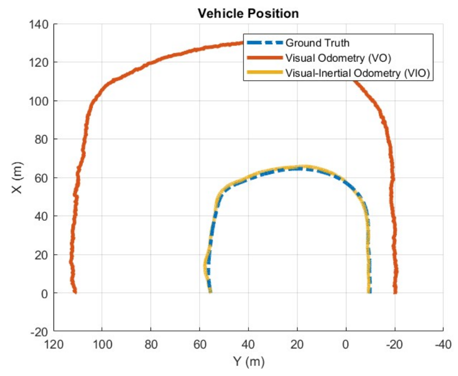


Fig. 4. Project Results

The second method as described previously has more perturbation, mostly after the curve, we want to highlight this situation, to make a sort of stress test for the filter. In the first phase during the straight road, the filter follows well the waypoint trajectory, after the curve, despite the low velocity, the path interpolation is less smooth as in the first method case, the vehicle makes sudden movements, this is caused because it not follow a specific motion model, but the only goal is to pass over the waypoints marked.

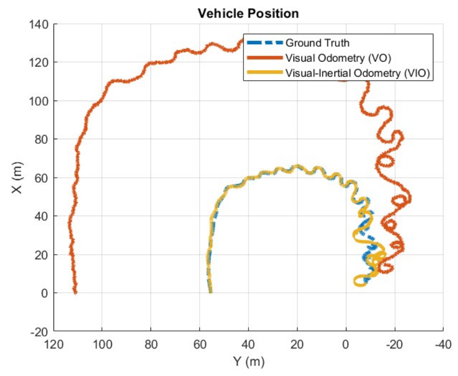


Fig. 5. Second methods results

As in the first method case, the estimation made by the **Visual Odometry** model is overscaled, it follows well the direction and the path, and these results are obtained not considering the bias of the camera, it is editable in the IV-B6 function.

References

[1] Juan Manuel Reyes Troncoso and Alexander Cerón Correa. “Visual and Inertial Odometry Based on Sensor Fusion”. In: *2024 XXIV Symposium of Image, Signal Processing, and Artificial Vision (STSIVA)*. 2024, pp. 1–5. DOI: 10.1109/STSIVA63281.2024.10637841.

[2] Patrick Geneva et al. “Versatile 3D Multi-Sensor Fusion for Lightweight 2D Localization”. In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 4513–4520. DOI: 10.1109/IROS45743.2020.9341264.

[3] Pragyan Dahal et al. “Extended Object Tracking in Curvilinear Road Coordinates for Autonomous Driving”. In: *IEEE Transactions on Intelligent Vehicles* 8.2 (2023), pp. 1266–1278. DOI: 10.1109/TIV.2022.3171593.

[4] Zhihang Song et al. “Synthetic Datasets for Autonomous Driving: A Survey”. In: *IEEE Transactions on Intelligent Vehicles* 9.1 (2024), pp. 1847–1864. DOI: 10.1109/TIV.2023.3331024.

[5] Miguel Pinto, António Paulo Moreira, and Aníbal Matos. “Localization of Mobile Robots Using an Extended Kalman Filter in a LEGO NXT”. In: *IEEE Transactions on Education* 55.1 (2012), pp. 135–144. DOI: 10.1109/TE.2011.2155066.

[6] MathWorks. *Driving Scenario Designer*. https://it.mathworks.com/help/driving/ref/drivingscenariodesigner-app.html. Accessed: 2025-01-15.

[7] MathWorks. *Create Driving Scenario Interactively and Generate Synthetic Detections*. https://it.mathworks.com/help/driving/ug/create-driving-scenario-interactively-and-generate-synthetic-detections.html. Accessed: 2025-01-15.

[8] MathWorks. *smoothTrajectory*. https://it.mathworks.com/help/driving/ref/drivingscenario.smoothtrajectory.html. Accessed: 2025-01-15.

[9] MathWorks. *waypointTrajectory*. https://it.mathworks.com/help/nav/ref/waypointtrajectory-system-object.html. Accessed: 2025-01-15.

[10] MathWorks. *Visual-Inertial Odometry Using Synthetic Data*. https://it.mathworks.com/help/driving/ug/visual-inertial-odometry-using-synthetic-data.html. Accessed: 2025-01-15.

[11] MathWorks. *Simulate Inertial Sensor Readings from a Driving Scenario*. https://it.mathworks.com/help/nav/ug/simulate-intertial-sensor-readings-from-a-driving-scenario.html. Accessed: 2025-01-15.

[12] MathWorks. *insfilterErrorState*. https://it.mathworks.com/help/nav/ref/insfiltererrorstate.html. Accessed: 2025-01-15.

[13] MathWorks. *Chase Plot*. https://it.mathworks.com/help/driving/ref/drivingscenario.chaseplot.html. Accessed: 2025-01-15.