Reconstruction of vehicle dynamics from inertial and GNSS data

Scuola di Scienze
Corso di laurea in Informatica

Supervisor:
Chiar.mo Prof.
Stefano Sinigardi

Co-Supervisor:
Dott. Alessandro Fabbri

Candidate:
Federico Bertani

I sessione
Anno Accademico 2017-2018

ABSTRACT

The increasingly massive collection of data from various types of sensors installed on vehicles allows the study and reconstruction of their dynamics in real time, as well as their archiving for subsequent analysis. This Thesis describes the design of a numerical algorithm and its implementation in a program that uses data from inertial and geo-positioning sensors, with applications in industrial contexts and automotive research. The result was made usable through the development of a Python add-on for the Blender graphics program, able to generate a three-dimensional view of the dynamics that can be used by experts and others. Throughout the Thesis, particular attention was paid to the complex nature of the data processed, introducing adequate systems for filtering, interpolation, integration and analysis, aimed at reducing errors and simultaneously optimizing performances.

La raccolta sempre più massiccia di dati provenienti da sensori di varia natura installati sui veicoli in circolazione permette lo studio e la ricostruzione della loro dinamica in tempo reale, nonché la loro archiviazione per analisi a posteriori. In questa Tesi si descrive la progettazione di un algoritmo numerico e la sua implementazione in un programma che utilizza dati provenienti da sensori inerziali e di geo-posizionamento con applicazioni a contesti industriali e di ricerca automobilistica. Il risultato è stato reso fruibile tramite lo sviluppo di un add-on Python per il programma di grafica Blender, in grado di generare una visualizzazione tridimensionale della dinamica fruibile da esperti e non. Durante tutto il lavoro di Tesi, particolare attenzione è stata prestata alla complessa natura dei dati trattati, introducendo adeguati sistemi di filtraggio, interpolazione, integrazione ed analisi, volti alla riduzione degli errori e alla contemporanea ottimizzazione delle prestazioni.

I'm personally convinced that computer science has a lot in common with physics. Both are about how the world works at a rather fundamental level. The difference, of course, is that while in physics you're supposed to figure out how the world is made up, in computer science you create the world. Within the confines of the computer, you're the creator. You get to ultimately control everything that happens. If you're good enough, you can be God. On a small scale.

Linus Torvalds

Please visit `https://github.com/federicoB/bachelor_thesis` **to find an updated version of this document**

1

INTRODUCTION

Vehicle dynamics reconstruction is a topic with much interest in automotive companies, car manufacturers and especially insurance makers. Being able to analyze car behavior in real time can give an enforced picture about driving style, road assessment, car occupancy and so on. Besides, the case of very important events, like a car crash, to have a clear picture of the dynamics prior to the accident can help determine responsibilities and liabilities. With the tool i will describe in this thesis, dynamics of a car can be represented visually using data coming from a vehicle black box. This thesis will in fact analyze techniques for dynamic reconstruction based on inertial and GNSS data used in a related software project.

The Physics of the City Laboratory inside the Physics of Complex Systems Group has been involved for years in the development of solutions related to human mobility, based both on pedestrian and means of transport. For this reason, collaborations have been opened with numerous Italian companies: in the insurance business (for example Unipol Gruppo S.p.A.), in the telematics branch (for example Octo Telematics Ltd), and in the industrial branch (for example Meta System S.p.A. and TEXA S.p.A.). From these collaborations, various activities were born: inertial sensors development and validation, product integration for the creation of "black boxes" for cars and study of techniques for the validation of collected data and reconstruction of the dynamics through the recorded data itself.

My thesis work has been inserted in the activities carried out by the group for these last two points. The data come in fact from prototype devices made by Meta System and collected directly on-board via a serial connection between the devices themselves and a system based on Raspberry Pi.

The whole on-board processing chain, serial transmission and registration on RPi device has been validated by the group during the years of activity and has led to the development of numerous tools used indirectly in this Thesis.

The data collected and used for the project can be distinguished in these classes:

- **inertial** data was created by accelerometers and gyroscopes, which measures respectively linear accelerations and angular velocities in a local reference frame.

- **GNSS** (Global Navigation Satellite System) was created by an electronic receiver. GNSS receivers provide latitudes, longitudes and altitudes. One of the most famous GNSS is the American GPS (Global Positioning System) but during experiments also Russian GLONASS and Chinese BeiDou signals were received.

All these sensors can be bundled in a box, as in our case, and fitted on a vehicle. This box transmits data to a remote server using conventional cellular networks.
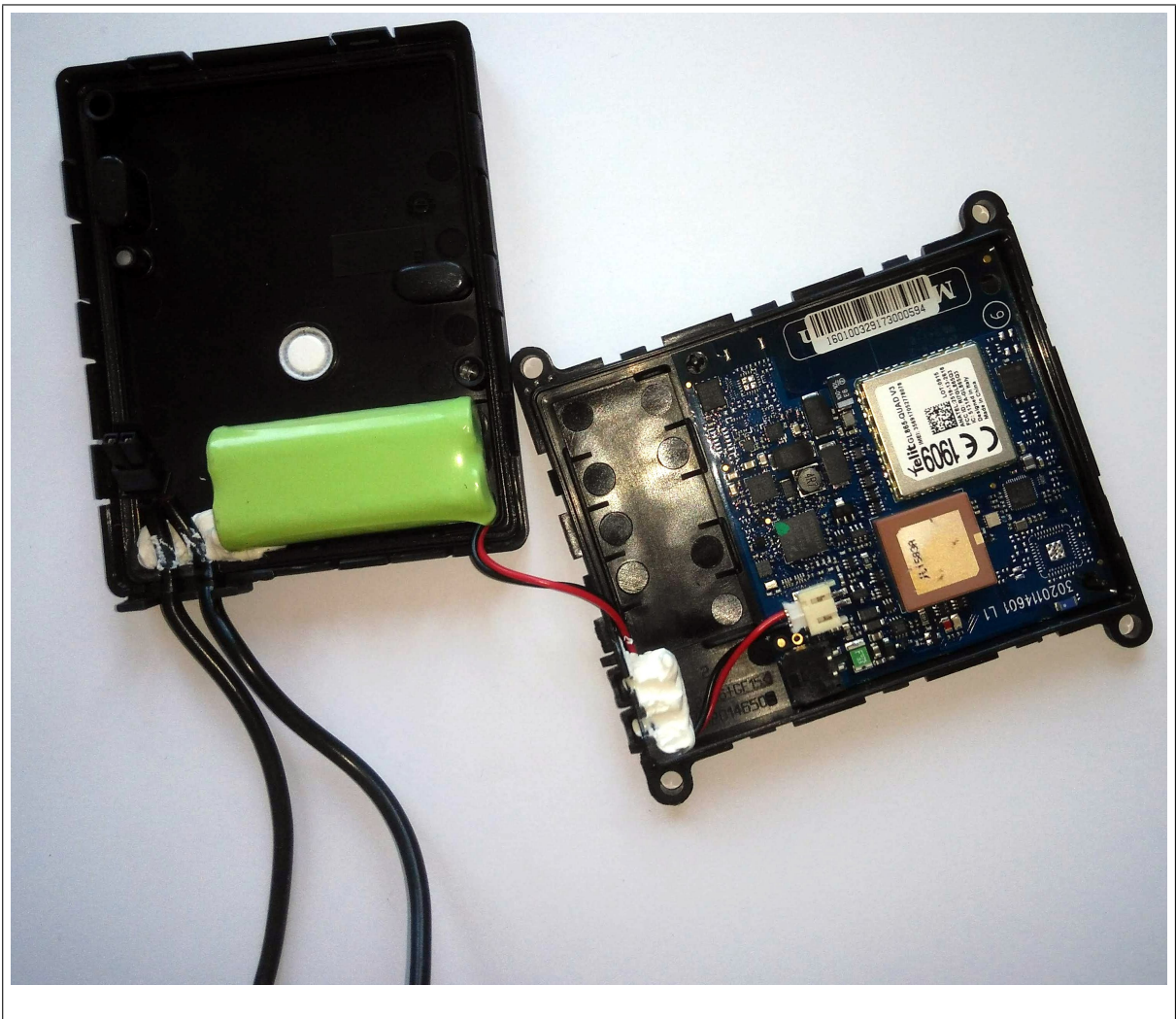

Figure 1.1: One of the sensor box used to collect data

The experimental work will consist of verifying the possibility of creating a software capable of recreating a vehicle dynamic visually realistic. The validation of results will be, in a first initial

phase where effectiveness of the program will be low, exclusively visually empiric, because of the need of rapid development and testing times. In a second and later phase, when quality of output will become more difficult to evaluate empirically, results will be compared to pictures and video footages captured in the data acquisition phases.

Aspiring software solution may face some challenges: sensor data gathering and transmission, correction of bad alignment, integration numerical error, precision reinforcement with multiple sensor fusion, representation of reconstructed trajectory.

Input date format is specified in a Physycom(PHYsics of COMplex SYstem) group's Github online repository [1].
I searched for an universal standard file format for inertial and GNSS data but I didn't find previous work.

Briefly formats supported by the software are all the following combinations:

	inertial	**inertial+GNSS**
interpolated	inertial	fullinertial
non interpolated	unmodified-inertial	unmodified-fullinertial

The *full-inertial* format is the more complete one, containing both accelerometer, gyroscope and GNSS data. Indeed, the program will give a better output when provided with a *full-inertial* input file, as it has more data to improve reconstruction. Records can be interpolated, in this case have always data from all sensors, instead of only from one of them, as they can have different frequencies.
In the case input data isn't interpolated, the software will take care of doing it.

The following table shows *full-inertial* data format in the order in which it is presented in the input files.

---

[1]https://github.com/physycom/file_format_specifications/blob/master/formati_file.md

Dataset Column Index	Name	Type	Unit of measure	Notes
0	Timestamp	Float	Seconds	From 1/1/2000 UTC+1
1	Latitude	Float	Degrees	From -90 to +90
2	Longitude	Float	Degrees	From -180 to +180
3	Altitude	Float	Meters	from sea level
4	Heading	Float	Degrees	Measured from magnetic north, from 0 to 360
5	Speed	Float	Km/h	
6	Acceleration X	Float	g-unit	Inertial
7	Acceleration Y	Float	g-unit	Inertial
8	Acceleration Z	Float	g-unit	Inertial
9	Angular speed X	Float	Degrees/Second	right-handed reference system
10	Angular speed Y	Float	Degrees/Second	right-handed reference system
11	Angular speed Z	Float	Degrees/Second	right-handed reference system
12	Acceleration module	Float	g-unit	
13	Relative time	Float	Seconds	From first record

I decided to visualize the reconstructed trajectory animated on the 3D modeling software Blender. The decision is motivated by the fact that Blender is a popular open source software and has an API to interact with. Additionally, it's already used inside the research group and one of its member is a 3D-artist specialized on it.

Road vehicle crashes reconstruction has already been a case study, with both inertial and GNSS data, for example in the work of S. Tadic [10].
None of the work I found ever dealt with computational problems, neither provided a 3D visualization of the reconstruction.

This Thesis consists of 9 chapters: after this introduction, in chapter 2 we will look to some mathematical notions necessary to comprehends techniques used in the project, then in chapter 3 I will write about what programming language I decided to use and what libraries
The chapters that follow are a description of the principal code parts: error reduction, vehicle rotation, numerical integration and the Blender add-on. Then in chapter 6 I included a part about software engineering practice I followed during the development of this project.
Finally, there's a conclusion chapter, with a recap of all the topics covered and a description of possible future improvements.

MATHEMATICAL BACKGROUND

In this chapter I will introduce some mathematical and physics notion used in the project. The first part of the chapter deals with numerical integration, a technique used to computationally calculate the integral of physics quantities like acceleration and velocity to get the trajectory traveled by the vehicle. The second part deals with quaternion, a number system useful to handle rotations in tridimensional spaces, more efficient when dealing with multiple composition of rotations than rotation matrices.

2.1   Numerical integration

Records in input dataset may not be equally spaced, which means that if t_i is a timestamp of a record at position i, then $\exists k, (t_k - t_{k-1}) \neq (t_{k+1} - t_k)$.

This implies that I can't use Simpson integration method, because it's a Newton-Cotes quadrature rule and it assumes that data points are equally spaced. [3]

Using a similar approach, interpolating with a second grade polynomial, I implemented an algorithm suitable for **irregularly spaced data.** [2]

Let's considering these consecutive points $(t_1, y_1), (t_2, y_2), (t_3, y_3)$ and a finite space of real functions, with its base of representation $\phi_0, \phi_1, ..., \phi_n$. Each function of the space can be defined as a linear combination of the basis:

$$\phi(x) = \sum_{i=0}^{n} a_i \phi(x)$$

Let's also consider the canonic form, where the basis are monomial $\phi_i = x^i \ \forall_{i=0}^{n}$

Indeed, every polynomial can be represented as a linear combination of monomials.

What follows is a linear equation system that requires a polynomial in the form $y = ax^2 + bx + c$

to pass through the points (t_i, y_i).

$$\begin{cases} at_1^2 + bt_1 + c = y_1 \\ at_2^2 + bt_2 + c = y_2 \\ at_3^2 + bt_3 + c = y_3 \end{cases}$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} \begin{pmatrix} t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \\ t_3^2 & t_3 & 1 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} t_1^2 & t_1 & 1 \\ t_2^2 & t_2 & 1 \\ t_3^2 & t_3 & 1 \end{pmatrix}^{-1} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Once a, b, c are found the integral of the parabola can be calculated as:

$$\int_{t_1}^{t_3} y(x)dx = \int_{t_1}^{t_3} ax^2 + by + cdx = \left[\frac{a}{3}x^3 + \frac{b}{2}x^2 + cx \right]_{t_1}^{t_3}$$

$$= \frac{a}{3}(t_3 - t_1)^3 + \frac{b}{2}(t_3 - t_1)^2 + c(t_3 - t_1)$$

Unlike classic Simpson method, this allow us to calculate also half of the area bound by the parabola, simply evaluating integral $\int_{t_1}^{t_2} y(x)dx$ or $\int_{t_2}^{t_3} y(x)dx$ instead of $\int_{t_1}^{t_3} y(x)dx$

2.2 Quaternions

Quaternions are an extension of complex numbers usually represented in the form:

$$q = a + xi + yj + wk$$

where a, x, y, w are real numbers and i, j, k are called *quaternion units*.

with the property $(ijk) = -1$

a is called *scalar* or *real* part of the quaternion, while x, y, w is the tridimensional *vector* or *imaginary* part. [6]

Quaternions have been formalized by the Irish mathematician William Hamilton in 1843.

If the scalar part is equal to zero then the quaternion is called *pure*.

2.2.1 Vector rotations with quaternions

Given a point $\boldsymbol{v} \in \mathbb{R}^3$, its rotation around versor $\boldsymbol{x} = \{a_x, a_y, a_z\}$ by an angle θ can be obtained in the following way:

$$p' = qp\overline{q}$$

where:

- p is a pure quaternion with the vector part equal to $\boldsymbol{v}$

- q is a quaternion representing rotation $q = e^{\frac{\theta}{2}(a_x i + a_y j + a_z k)} = \cos\left(\frac{\theta}{2}\right) + (a_x i + a_y j + a_z k)\sin\left(\frac{\theta}{2}\right)$

- $\overline{q}$ is the conjugation of quaternion q definied as
  $\overline{q} = (s, -\boldsymbol{v}) = s - xi - yj - wk = e^{-\frac{\theta}{2}(a_x i + a_y j + a_z k)} = \cos\left(\frac{\theta}{2}\right) - (a_x i + a_y j + a_z k)\sin\left(\frac{\theta}{2}\right)$

- p' is a pure quaternion with its vector part equal to vector $\boldsymbol{v}$ rotated

2.2.2 Product of quaternions

Given two quaternion q_1 and q_2 defined as:

$q_1 = (s_1, \boldsymbol{v_1}) = s_1 + a_1 i + b_1 j + c_1 k$

$q_2 = (s_2, \boldsymbol{v_2}) = s_2 + a_2 i + b_2 j + c_2 k$

then their product can be defined as:

$q_1 q_2 = (s_1 s_2 - \boldsymbol{v_1} \cdot \boldsymbol{v_2},\ s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_1} + \boldsymbol{v_1} \times \boldsymbol{v_2})$

2.2.3 Quaternion product isn't commutative

$$q_1 q_2 \neq q_2 q_1$$

is equivalent to

$$(s_1 s_2 - \boldsymbol{v_1} \cdot \boldsymbol{v_2},\ s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_1} + \boldsymbol{v_1} \times \boldsymbol{v_2})$$

$$\neq$$

$$(s_1 s_2 - \boldsymbol{v_1} \cdot \boldsymbol{v_2},\ s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_1} + -(\boldsymbol{v_1} \times \boldsymbol{v_2}))$$

by the anticommutative property of vector product [6]
$\boldsymbol{a} \times \boldsymbol{b} = -(\boldsymbol{b} \times \boldsymbol{a})$

2.2.4 Comparison with rotation matrices

Matrices can be used to perform rotation. [5] A quaternion can be associated with a rotation matrix and vice versa. [6]

Aside from memory usage, where quaternions are more efficient, the time complexity must take into account the conversion between rotation representations, as from axis-angle in our case, and actual context of usage as explained in the following.

Performance can be evaluated at a high-level by counting the number of operations needed to perform conversions and rotations. Operations like additions or subtractions, multiplications, divisions and trigonometric function like sines and cosines.

In the following analysis I will leave out performance optimization that computer systems can apply in some situations, like for example the simultaneous calculations of sine and cosine

performed by modern machine. Just looking at number of operations should give a general idea of difference between rotation representations.

Axis-angle to matrix conversion requires 13 additions, 15 multiplications and 2 trigonometric calls.

Axis-angle to quaternion conversion requires 4 multiplications and 2 trigonometric calls.

Rotating a vector using a rotation matrix requires 5 additions and 9 multiplications.

Rotating a vector using a quaternion, optimizing considering its scalar part zero, requires 17 additions and and 24 multiplications.

So it results that rotation matrix are more efficient in rotating a vector, but the real advantage of quaternions are rotation compositions. Indeed, composing two quaternions require only 12 additions and 16 multiplications while matrices, would have required 18 additions and 27 multiplications.

What follows is a comparison analysis of the number of operations required to rotate n vectors by a cumulative n angles represented as axis-angle notation. Those abbreviations are used: A: Additions / subtractions, M : Multiplications, F: trigonometric Functions

- rotation matrices: $((13A + 15M + 2F) + (5A + 9M) + (18A + 27M))n = (36A + 51M + 2F)n$

- quaternions: $((4M + 2F) + (17A + 24M) + (12A + 16M))n = (29A + 44M + 2F)n$

PROGRAMMING LANGUAGE AND LIBRARY CHOICES

3.1 Python

For this project, the Python programming language was chosen. [9]

Python is increasingly being used by the scientific community for its large collection of packages. It allows to link normal Python code to C / C++ extension when performance optimization is critically important.

In addition, Blender only offer a Python API.

All benchmarks showed are based on standard Python implementation CPython.

3.2 Python scientific stack

- **Numpy** is a data structure library for n-dimensional arrays. Instead of creating an object for each integer like vanilla Python, with related garbage collector overhead, it handles array in contiguous memory space like C. Another big performance advantage is through vectorized operations, that permit to modify multiple array elements with a syntax like they were a single value. For example, if `arr` is an array of n elements, to multiply each element for the constant value 3, one doesn't need to use a cycle that iterate on array, but can just write `arr * 3`. This is not just syntactic sugar, but it offers to the interpreter additional informations, like that the operation would be on all elements without interruption from side effects, implying a performance improvements thanks to of optimized cache usage.

- **Pandas** is a package for data analysis. Its main data type is called `DataFrame` and repre- sents a two-dimensional dataset.`DataFrame` can be easily created from `csv` files, offering various methods. It is possible to complete a project like this only with pandas, as it uses

numpy underneath, but it offers a very expressive interface that needs a lot of logic beneath with a non negligible overhead. For this reason I reduced the Pandas usage and called it only when strictly necessary.

- **Scipy** offers many mathematical routines on top of numpy arrays. I used mainly trigonometric functions like arctangent, cosine and sine. It has big precision constant for example for Pi and offers vectorized routines for cross and dot products.

- **Matplotlib** is the most used python plotting library, most of the other plotting libraries are based on it trying to abstract and elevate its complex interface. Nevertheless one can create very appealing charts also with low-level Matplotlib methods, especially it offers full customization.

3.3   Quaternion libraries

My Project initially started with **numpy-quaternion** [1] but then moved to **pyquaternion** [11] because of the following reason:

- **moble/quaternion** didn't work on Windows [1], as stated in the readme. This was a blocking issue because the 3D artist of the group was using a Windows workstation.

- **pyquaternion** has a better documentation and more high-level methods

On the downside, pyquaternion has performance disadvantages. As an example the following code using **numpy-quaternion**

```
from numpy_quaternion import quaternion as Quaternion
quaternions = np.array(
[np.exp(Quaternion(*np.asarray(delta_theta)) / 2) for delta_theta in delta_thetas]
)
```

runs in 0.665 milliseconds
while the following code using **pyquaternion**

```
from pyquaternion import Quaternion
quaternions = np.array(
[Quaternion.exp(Quaternion(vector=delta_theta)/2) for delta_theta in delta_thetas]
)
```

runs in 24.6 milliseconds, both with an array of 100 records

---

[1] as per commit 03d627 the library works on Windows but it was fixed too late for this work

4

Input data from sensors doesn't represent real measurement of vehicle physics quantities. Instruments have a measuring error, each one with different possible reasons. This chapter deals with techniques used in the project to reduce that error, whether it was experimental or systematic.

4.1 Stationary time detection

Most of the solutions used to remove errors are based on the assumption of a vehicle state, in particular the stationary one.

This can't be detected by a near zero acceleration along all axis because vehicle could be moving with constant speed. So integrated speed should be used, but integrating acceleration vectors that need corrections can bring next code logic to make mistakes as well as being a waste of time, because integration should be recalculated after having applied corrections.

One solution can be to use non-directional speed from GNSS data by the measure device in the box. But this speed is under effect of Kalman filters to avoid measurement errors The filter is implemented directly inside box firmware, but reacts late to changes. One can differentiate numerically the GNSS positions to get a more precise speed and, on top of that, calculate stationary times. The choice of using speed subject to Kalman filters or the one derived should be based on GNSS sensor precision. Especially when vehicle is stopped, its error can be relevant as showed in the following picture.

Figure 4.1: Green points represent GNSS positions. On top-left GNSS error can be noticed in a situation where vehicle was stopped during experiment

To try to handle input data where vehicle never really stops, the algorithm for stationary time detection searches into dataset with an increasing threshold, until enough periods are found.

4.2 Gyroscope drift

Gyroscopes have a drift and an offset that is unavoidable. [4]
Offsets and their drifts can be measured when vehicles are stationary. Average value of angular velocities in stationary moments is an approximation of gyroscope offset. During motion, a gyroscope exposed to heat increases its drift. So I made sure drift removal was applied continuously at each stationary time, removing the drift detected onwards.
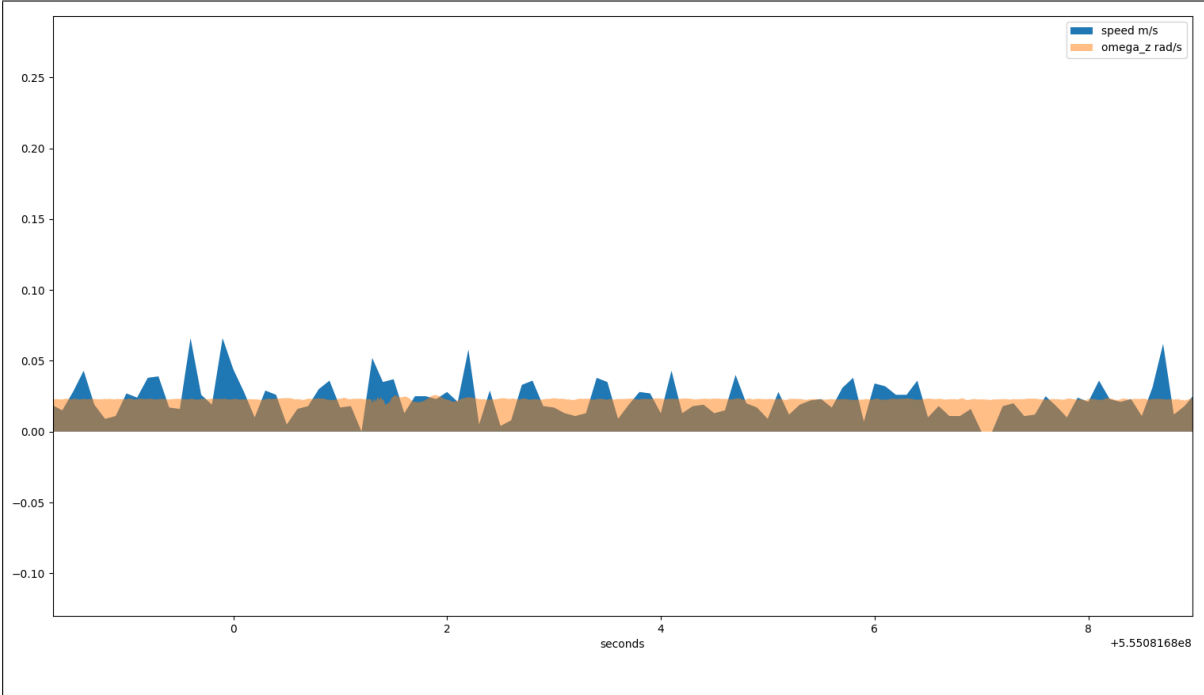
Figure 4.2: Data from stopped vehicle. Linear speed is near zero but gyroscope detects a non negligible angular speed

4.3   Noise reduction

Sensors are subject to noise both electronic (from the board itself or the electric supply) and mechanical (from their setup and the vehicle). In the data analyzed for this project I found spikes without any correlation to real events. There are various techniques to remove them, the one I used is the rolling average.

Given a series s, the *centered* rolling average with a window size w is defined as:

$$v_i = \frac{1}{w} \sum_{j=0}^{\frac{w}{2}} v_j \sum_{j=\frac{w}{2}}^{w} v_j$$

where v_i is the element of series s', that is s with centered rolling average applied. The choice of a value for window size is important. A value too small can lead to a noise left too high, while a windows size too large value can lead to *flattening* and reduction of measured value. For example with a value too high, an acceleration can be measured even before it actually started and overall highest value will be lower than in reality.

4.4   Correction of vertical alignment

Box vertical unalignment is corrected by looking at gravitational acceleration on stationary times. Since g is always presented by the accelerometer, it can be used to determine vertical axis. This

13

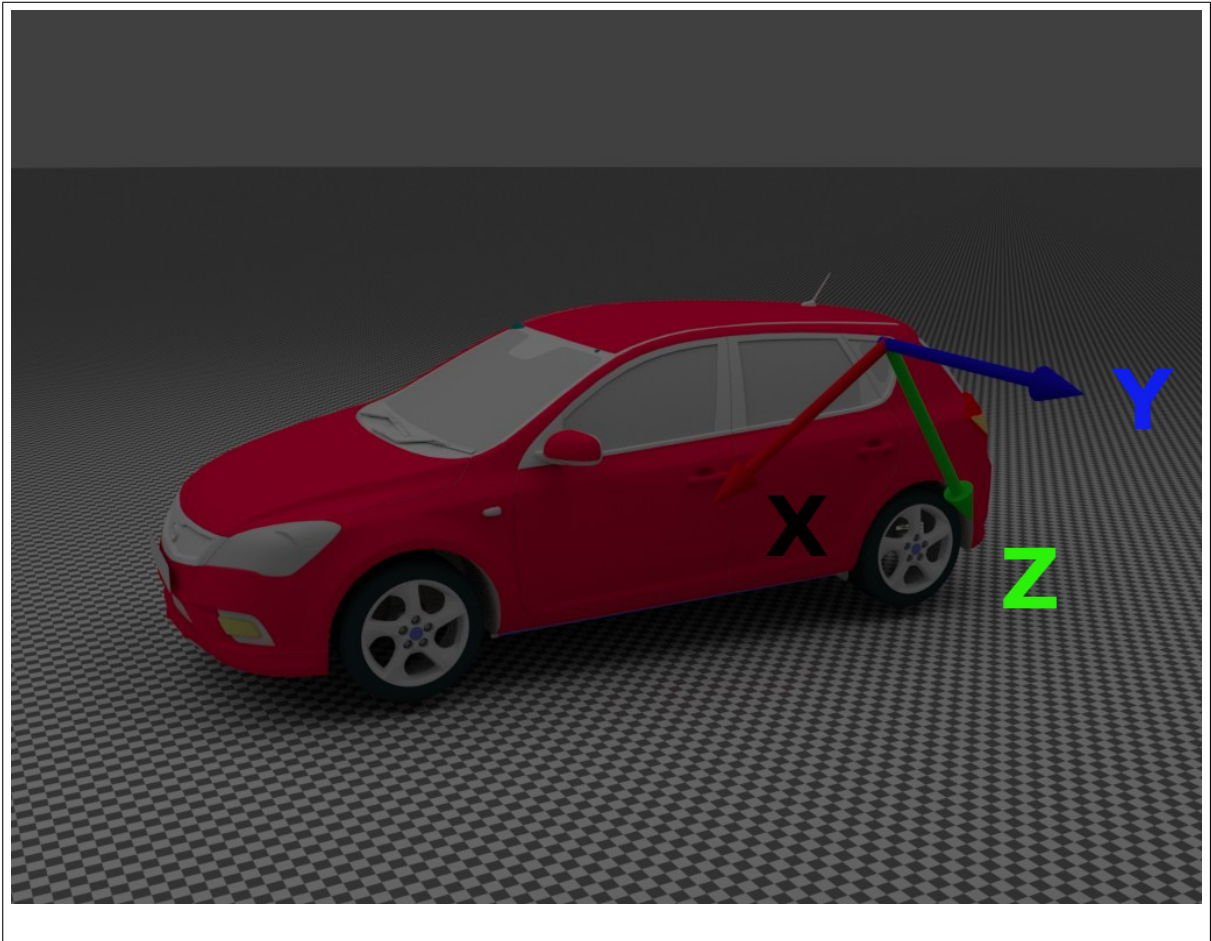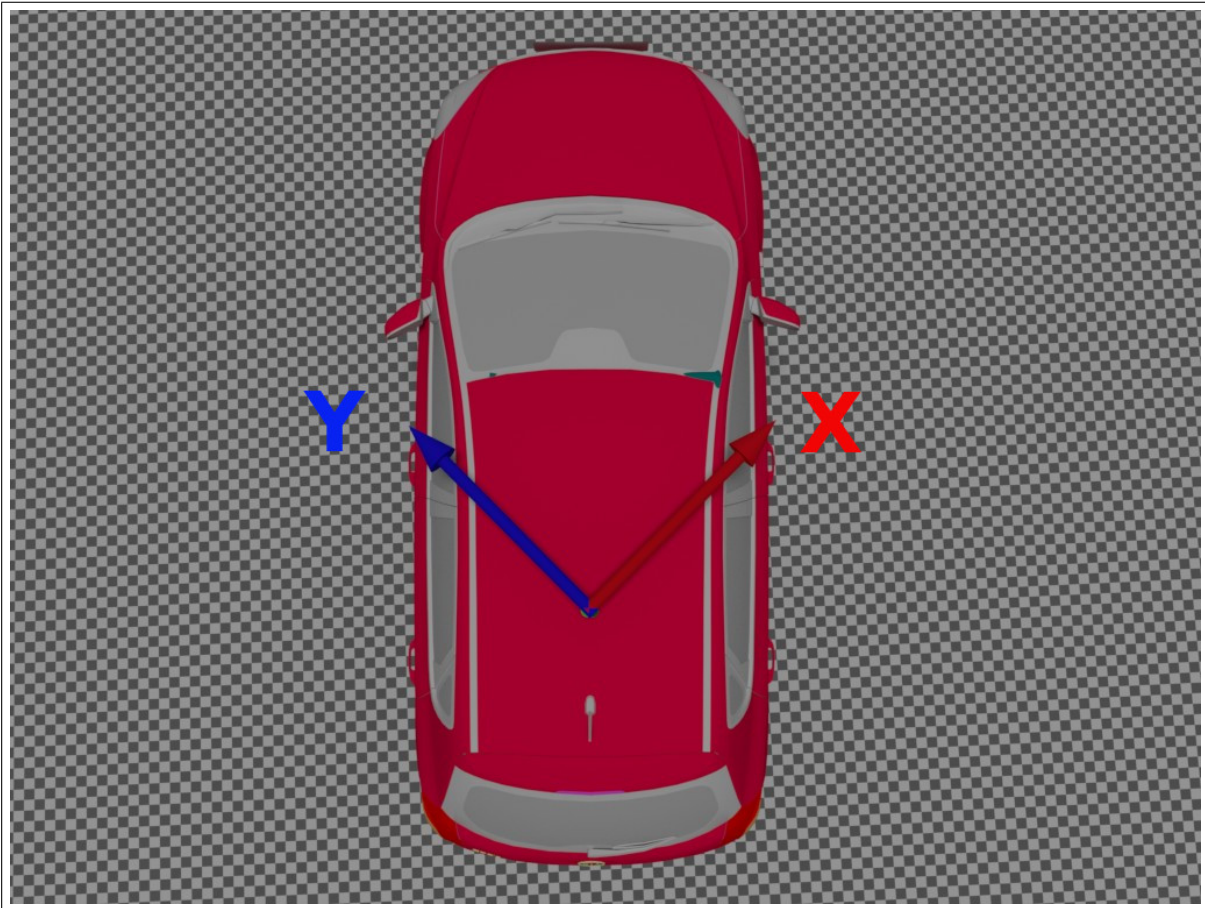technique only works assuming the vehicle rest on a flat surface.

Figure 4.3: Axes shows a sensor bad vertical alignment

Once obtained the rotation angle, the program rotates both local acceleration and angular velocities. After having done that, if in a following stationary time a bad vertical alignment is still found, the program prints a warning on the console to notify that the box become misaligned during motion.

4.5 Correction of horizontal alignment

Box horizontal unalignment is corrected by looking for situations where $a_x > 0$, $a_y > 0, \omega_z = 0$. If this state exists, then the car is sliding on ice without rotating on itself or there is an unalignment on the xy plane. The latter is more likely.

On unaligmnent, for example, an acceleration forward will be measured with both positive accelerations along x and y axis. Still in the situation described, the misalignment can be measured and corrected, still by rotating both local accelerations and angular velocities.

ROTATION WITH QUATERNIONS

This chapter deals with the usage of quaternions for rotations and in general with all of the project parts related to the transformation from a local frame of reference to a laboratory one.

5.1 From local to laboratory

The program calculates the change of angular positions by integrating angular velocities, which are described in a local reference frame. This means that for calculating angular position at time t_{i+1} we must take into account angular position at time t_i, so that it is reconstructed iteratively.
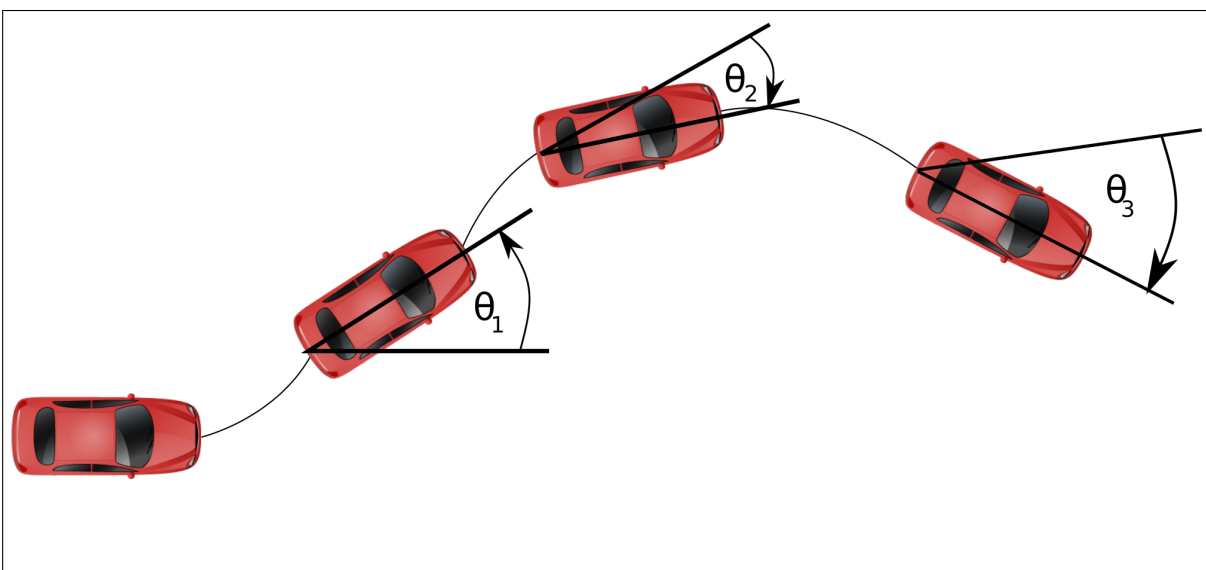

Figure 5.1: Angle composition in vehicle motion

One of the first error I committed was to consider the rotation composition as the sum of angles on each axis. I was using the same integration routine used for linear dimensions, which performs the cumulative sum on each axis independently, i.e. the sum of the areas. When composing multiple rotations, for example when having a rotation around vertical versor $\hat{z}$ by an angle θ_z, besides adding θ_z to the vertical rotation, one must move rotation axes that belongs to the horizontal xy plane.

So I divided the integration routine in two functions, one that returns an array of areas, the other that calls the first and then does the cumulative sum of the array of areas. I've done this also for keeping compatibility with others modules that were using the previous single function.

Then, once I had the angle variation vector $\langle \Delta\theta_x, \Delta\theta_y, \Delta\theta_z \rangle$ the associated quaternions can be created and multiplied together. Essentially, the two quaternion libraries I used in the project overload mathematical operators so compositions and rotations become more straightforward.

For quaternion composition, initially I was using `numpy.cumprod()` [8] but this function was performing multiplications in reverse order, with respect to what I needed for my application. For example, calling `cumprod()` on a vector $\langle a, b, c, d \rangle$ will result in $\langle a, ab, abc, abcd \rangle$ while I needed $\langle a, ba, cba, dcba \rangle$

So I wrote some custom code to make quaternion composition, using `functools.reduce()`, since it's faster than using a `for` cycle.

```
quaternions = reduce(lambda array, element: [*array, element * array[-1]],
                     quaternions,
                     [initial_quaternion])
```

The reason is that lambda use local scoped variables, that can be kept in register and cache, and that interpreter can preload next items.

5.2 World frame of reference

There exists an additional frame of reference, similar to the laboratory one described before but with an additional rotation in the horizontal plane. This frame of reference has y-axis oriented towards geographical north and x-axis to east. This is critically important for mixing GNSS data that is already in this frame of reference, with integrated inertial data.

To find the angle between GNSS positions and east, the program finds the first position that is at least 10 meters away from the initial one, then calculates the angle of this vector.

Assuming that the first motion of the vehicle is in the forward, direction given the previous calculated angle, all accelerations should be rotated.

Instead of rotating all accelerations and angular positions, the program sets only the initial

position with respect to the world frame of reference. Since the following angular positions are calculated from the first, no additional operations are needed, preventing the execution of $O(n)$ operations, i.e. the rotation of all records.

6

TRAJECTORY INTEGRATION

Initially I coded both the square, the trapezoid and the parabola method, to analyze better suited my necessities. [3] The last one, from now on, will be called PaIS (Parabola integrator for Irregular Spaced data),

A quadrature formula of grade n provides the exact integral value of a polynomial of grade $\leq n$, which means that it isn't always necessary to use a formula with a high grade.

I verified experimentally the difference in error between integration methods by creating synthetic trajectories.
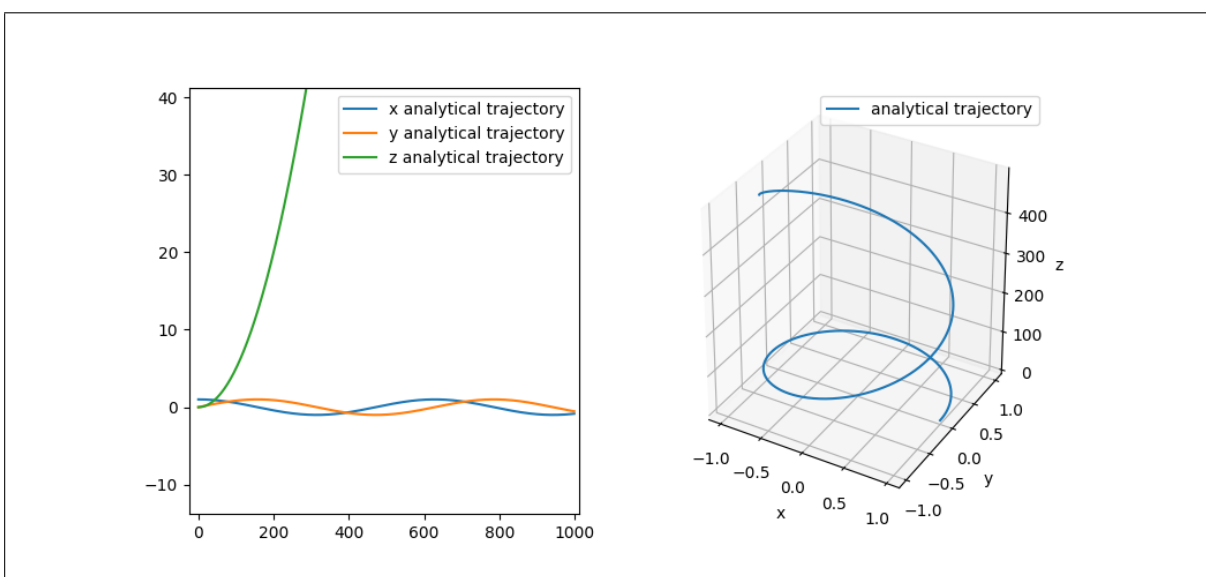

Figure 6.1: Trajectory created to evaluate integration error

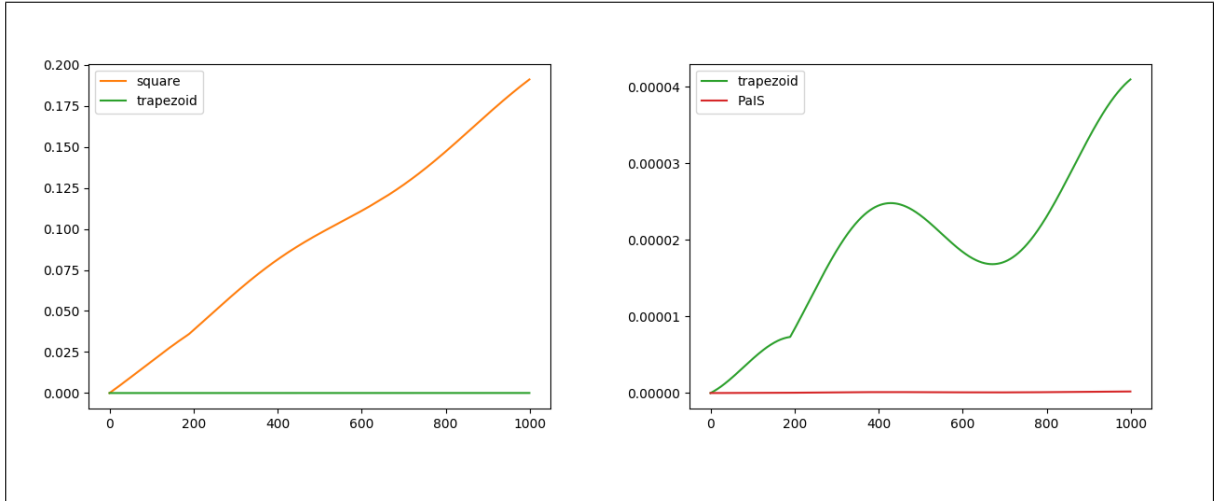then I compared the integrated one with the analytical.

Figure 6.2: Absolute error from analytical trajectory

Numpy offers a compact way to write operation even on large arrays. This is for example the trapezoid method in one line.

```
def trapz_integrate_delta(times, vector):
    return (((vector[:,:-1] + vector[:,1:]) * (times[1:]-times[:-1])) * 0.5).cumsum()
```

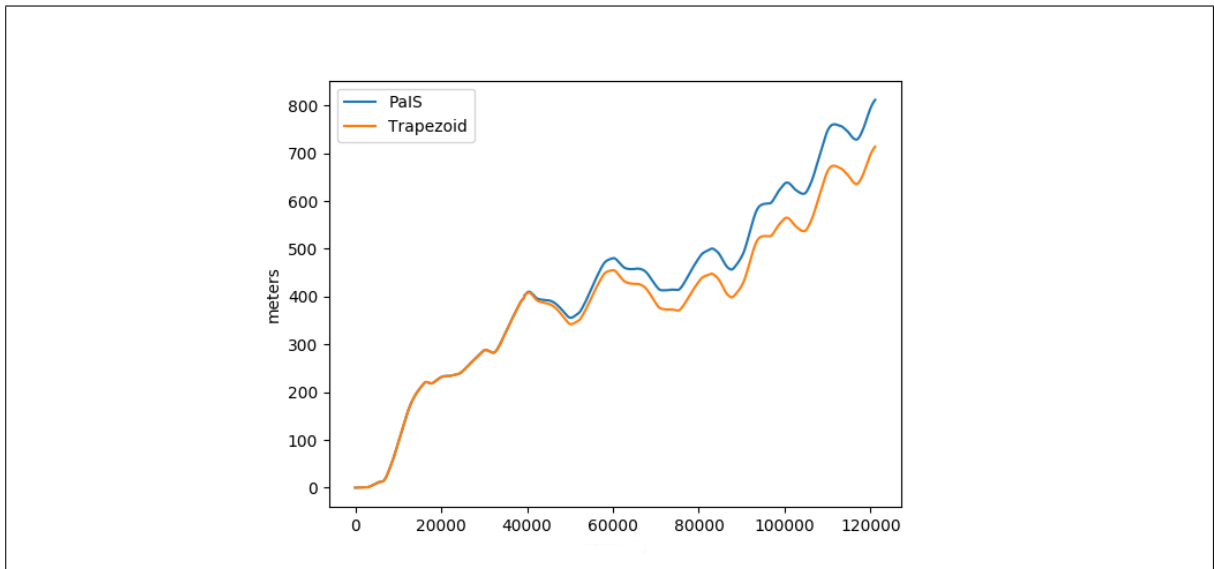Difference between PaIS and trapezoid can be seen after integrating for 40 thousands steps.

Figure 6.3: Difference between trapezoid and PaIS method on real world data

I decided to use the PaIS integrator because measures in discrete time produce a C^0 function, while physical quantities of a vehicle in continuous time are at least C^1.

The errors were still too large, the integrated trajectory diverges after some time by moving away too much from the real trajectory. So I improved integrated trajectory using GNSS data, which is more precise but has lower frequency and has difficulties on finding orientation. In this way, the output trajectory benefits from both measures, using their respective advantages and reducing each other disadvantage. Accelerometer and gyroscope are really good in small time frames, but due to integrated quantities have precision problems on long time spans. On the other hand commercial GNSS sensors don't have very precise positioning but they keep working steady for much times. I started from resetting the integrated position to the GNSS one every t times, but this was causing an edgy and irregular behavior that wasn't realistic. I proceeded to implementing a contiguous weighted average, so that GNSS component is always present in the output trajectory but its impact is limited. Especially when GNSS signal is low,the position completely wrong, in the order of tens of meters. Giving a low weight to GNSS, like 1 on 99, avoid problems caused by low signals but still helps with correcting numerical integration error.
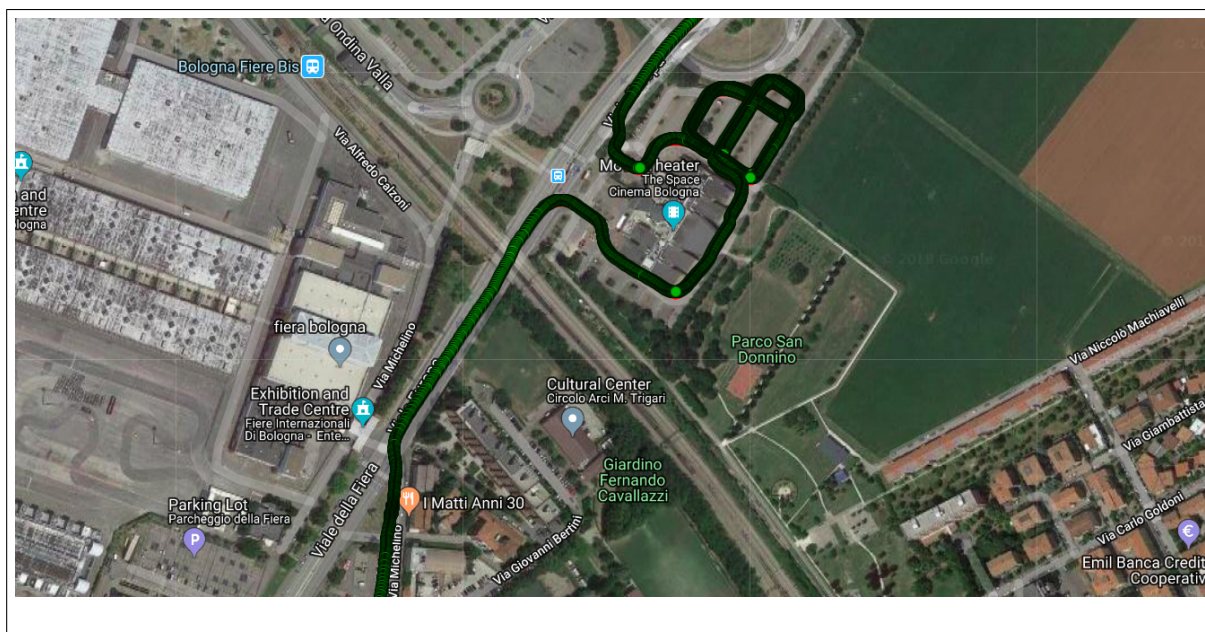

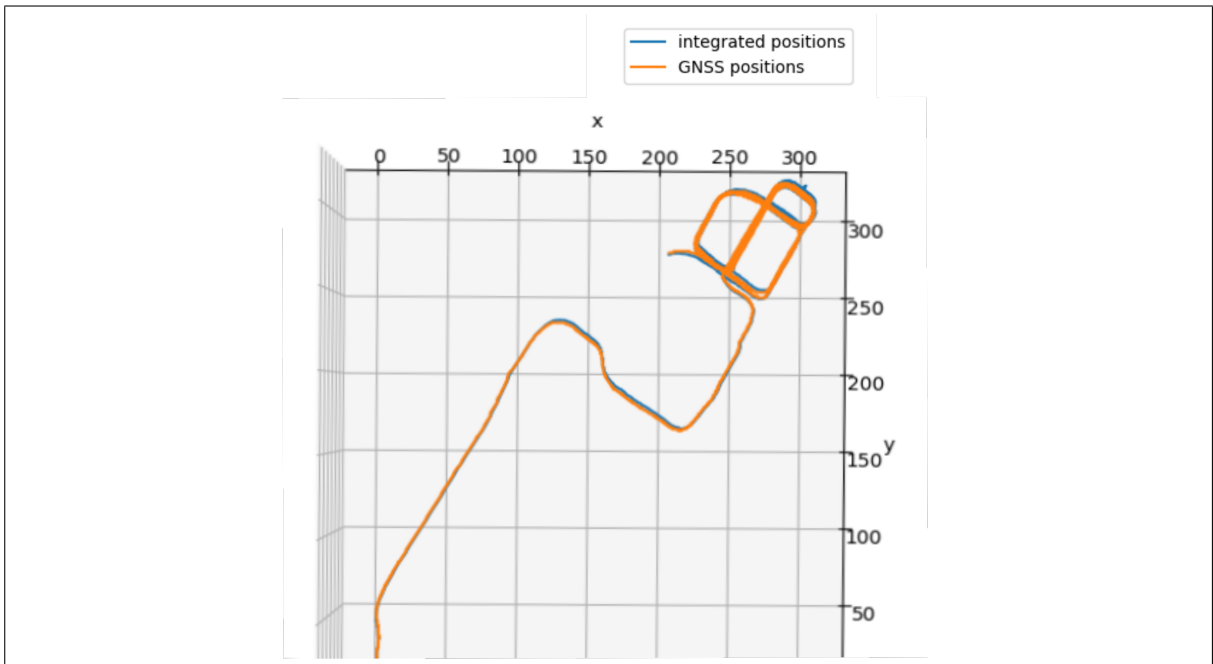**Figure 6.4:** Path traveled by a car with a box installed, in a Bologna parking

Figure 6.5: The same path from data registered by sensors and elaborated by the program
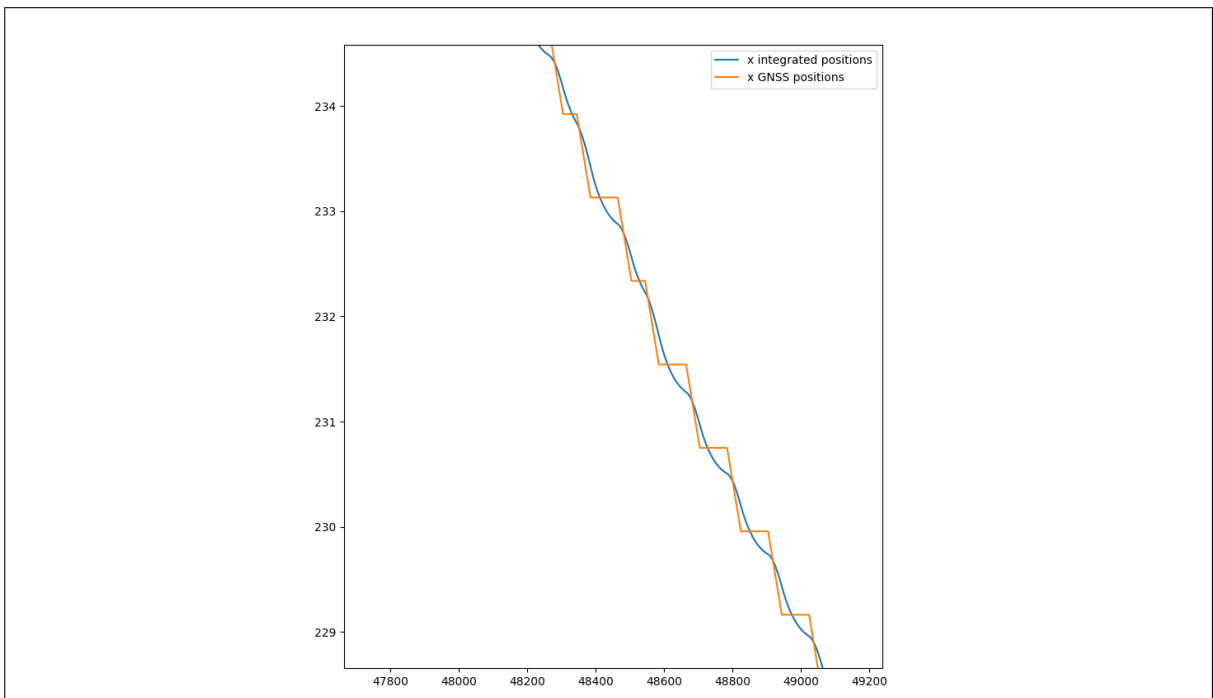

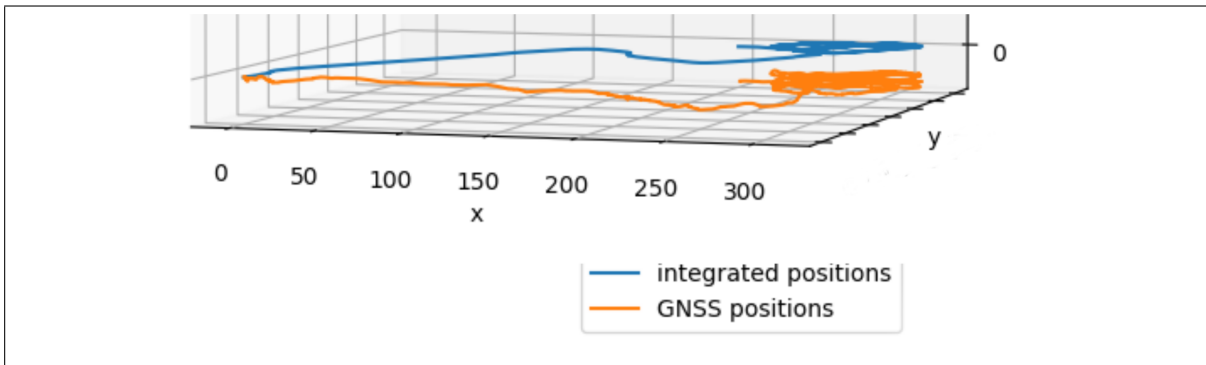**Figure 6.6:** Particular showing smoothness of integrated trajectory in respect of GNSS one

Figure 6.7: Note that GNSS and integrated path are distant on vertical axis. This is because altitude from GNSS system is extremely unreliable to determine vertical position. In my algorithm i Decided to neglect this information and only use the integrated accelerometer information.

7

Blender has a Python Application Programming Interface (API). Through the API, one can create a Python module and package it as Blender add-on. Add-ons are extension of Blender basic functionality, managed by a section of Blender settings and installable by file. As alternative to add-on to interact programmatically with Blender is possible to create a Python script and load it into the built-in text editor (and execute it) or give it as parameter while launching Blender from the command line. Add-ons provide versioning and authoring, remain activated between Blender files and don't need to be loaded each time.

7.1   Blender API

All the operations are made through bpy Python module. bpy can be subdivided additionally into:

- `bpy.context`: access to current active scene and elements.

- `bpy.data`: access to all objects in Blender document.

- `bpy.types`: built-in Python classes, usually extended.

- `bpy.props`: abstraction of a variable properties, having type `bpy.types.*Property`; if registered in `bpy.props`, it can be modified by graphical interface.

From `bpy.types` other classes used in the projects are:

- Operator: essentialy a function to manipulate objects. Every action that the user can do in Blender is coded like an `Operator`: like moving, modelling and sculpting objects.

- Panel: a section of the graphical interface.

7.2   Blender add-on anatomy

An add-on can be a single Python file or a zip containing an `__init__.py` file and other complementary files.

In both cases, the main Python file must have a `register()` and `unregister()` function implemented.

The `unregister()` takes care of calling `bpy.utils.unregister_class()`, for each class that needs to be loaded into Blender.

When loading a class, Blender performs sanity checks, making sure all required properties and functions are found, that properties have the correct type and that functions have the right number of arguments. `register()` function of registered class are called recursively and, for example, `Panels` are drawn on user interface at this moment.

The `unregister()` have a totally opposite task compared to `register()`.

It should call `bpy.utils.unregister_class()` on every register call. It's preferable to make the calls in the reverse order compared to `register()`.

Other objects defined in the Python add-on are:

- subclass of `Panel` for user-interface, a button to open a file system explorer dialog to select input dataset, an editable text field showing input dataset path and a button to launch dynamic reconstruction.

- `StringProperty` for storing input dataset file path and to share it between operators.

- subclasses of `Operator` for reaction to buttons clicks on user interface and calling related project functions.

Finally, a Python dictionary called `bl_info` is defined with add-on metadata as name, version and author. Those data are displayed in the list of add-ons in Blender settings.

7.3   Installation of dependencies

One particular challenge I faced was to have project library dependencies installed in blender internal interpreter.

There are three solution to this problem. Two solutions were discarded:

- Removing the Blender Python folder and Blender will fallback to system interpreter. But API supported Python version can be different to the system one and doing this is dangerous.

- Changing *Scripts path* inside Blender setting, to look to another directory with a specific structure. This directory must contains `add-ons,` `modules` and `startup` directories. Project dependencies should be installed in `modules` directory. There are various problems to this solution: it can be easily made non-working by a Blender update, removing the possibility

of changing *Scripts path*, also it can be against the final Blender user will, requiring all previous installed add-ons to be moved to new directory.

The solution I decided to follow and implement is to increase Blender interpreter capabilities by installing the **pip** package manager on it and then using *pip* to install the dependencies listed in the `requirements.txt` file. The whole process can be labeled as *bootstrapping*, as we start from a very small capability and we use it to have more at each step.

First step is to download a Python script which has *pip* binary included inside. To do this it is very useful the `get-pip.py` script. [7] Then, `get-pip.py` must be run by the Blender interpreter so it will be linked to it.

In the Blender Python directory a pip executable will emerge so project dependencies can be installed with it and modules will be linked to.

Including the code to do this process in the add-on is crucial to have a single `zip` to distribute and increase final user experience.

This part has been a bit tricky due to differences between operating system; both for file system structure and operating system interfaces to execute programs.

8

SOFTWARE ENGINEERING CONSIDERATION

This chapters deals with software engineering (SWE) topics: software design, quality measure and process management.

During SWE course I followed was focused on object oriented software design, that doesn't really fit for this project. The domain doesn't contain many objects to model and python classes introduce an overhead which is incompatible with performance required, taking into account input dimensions.

Anyway, even if I used a more functional approach, I still divided the codebase into modules using *responsibility-driven design*.

8.1 Requirements

After several discussions with my supervisor, I modeled the following functional requirements:

1. the software will support inputs in the format specified in the introduction chapter;

2. the software will use Blender API to animate a vehicle so that its linear and angular positions reflect the one from which the input data is measured;

Non functional requirements are:

1. performance: the software must at least elaborate a dataset of 100.000 records in less than five minutes;

2. user experience: the software must be easy to use also from a 3D artist point of view;

3. operating systems. the software must support of Windows and GNU/Linux ;

8.2 Structure

Satisfying functional requirement 1 is the main challenge of the project. Recreating a perfect reconstruction is very hard and requires many techniques as showed in the previous chapter. Following the methods previously illustrated the *responsibilities* of the software are:

1. input loading;

2. input parsing;

3. input format auto-detection;

4. noise reduction;

5. gyroscope offset drift reduction;

6. correction of vertical bad alignment;

7. correction of vertical bad alignment;

8. derivation of GNSS speed and acceleration;

9. moving from local to laboratory reference frame;

10. moving from laboratory to world reference frame;

11. integrate accelerations;

12. integrate velocities;

13. timestamps normalization;

14. dependencies auto-installation;

15. blender user-interface handling;

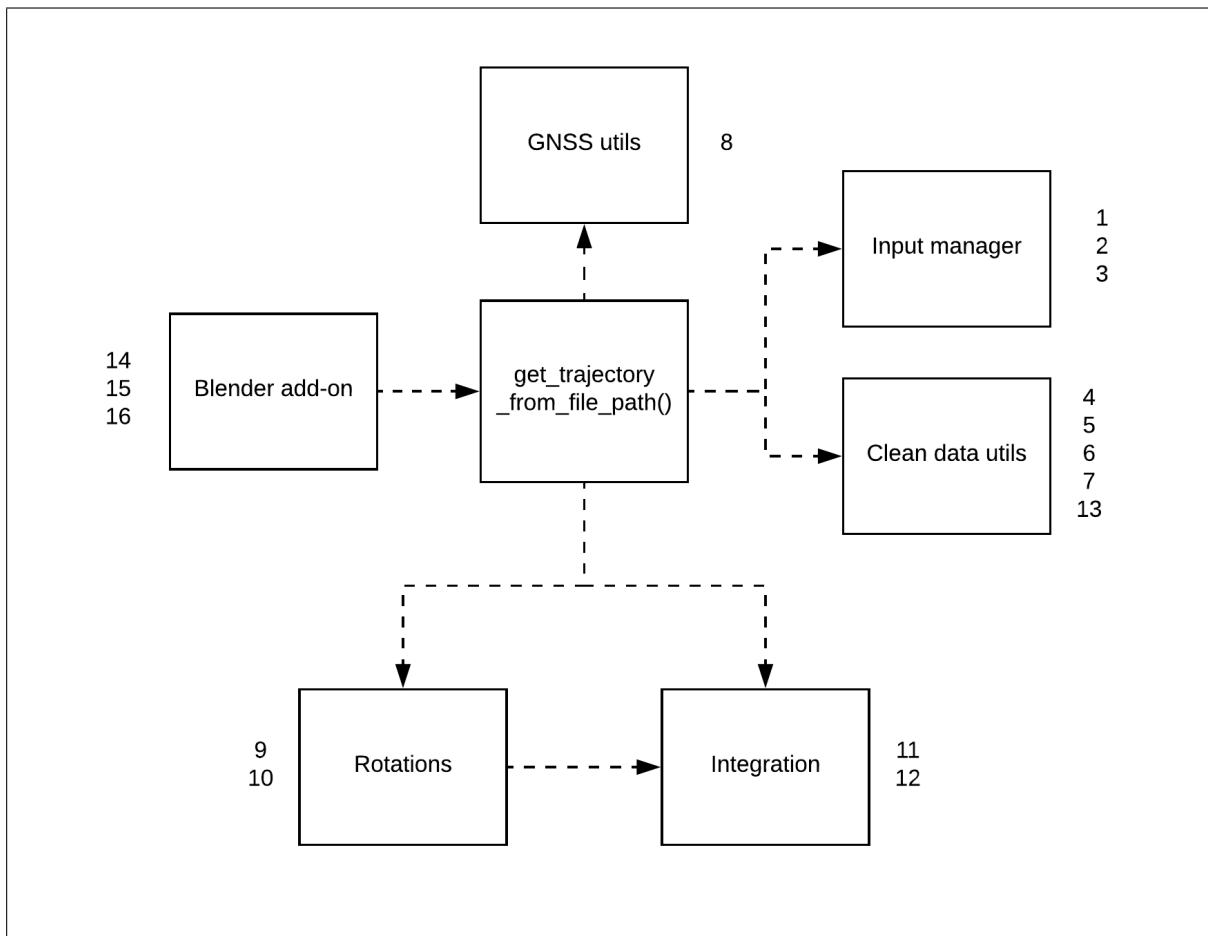16. creation of animation in Blender;

Figure 8.1: UML diagram showing project modules and their dependencies. The numbers nears each modules states responsabilities assigned to it.

8.3 Dependency management

Current software development practices suggest to *don't reinvent the wheel* and re-use as much as existing software as possible. Project stability is related to its dependencies' stability. Delegating responsibilities to other trusted programmers reduce project risks and increase overall quality. Specifying explicitly project dependencies is useful both for the programmer and for the user, in particular specifying which version of each dependency the software is assured to work. Python has various tools to handle this, the most popular are *pip+virtualenv* and *Conda*.

Virtual environments allow to isolate Python packages on which the project depends on from the global system ones. This allows to use different packages version.

Pip (Pip Install Packages) can install packages from a the PyPi repository, the largest one containing more than 143 hundred packages, parsing a list file containiting dependencies.

Conda is a package manager designed for every language, instead virtualenv is only for Python. It was created from the PyData community to overcome Pip limitations and doesn't use only

PyPi to retrieve packages. Instead it has customized version of python scientific packages like numpy, scipy and pandas, optimized for performance as Conda can work on a lower-level in target machine.

Initially, I was using Conda and I noticed a better performance, but when I created the Blender add-on code part that auto install dependencies I had to move back to virtualenv+pip because it was easier to install.

8.4 Unit testing

On the project I applied the principles of *unit testing*. I tried to create a test for every function I've written, sometimes even before the code itself to be tested, to make it minimal.

During this process I prepared a tool to produce syntetich trajectories, to overcome specific requirements of some tests. The tests I've written are incremental, some verify only small functionalities, others verify more overall features that use the small functionalities underneath. For example, I created a test with a complex trajectory similar to the one of a spring, to test integrator precision, then I created a circular trajectory, where body also rotate like a car in a roundabout to test, angular position integration and body rotation. In this way it is easier when a breaking change is introduced to debug and fix the problem.

Right now the test coverage, which is the percentage of code lines that are executed by automatic tests, of project's principal modules is 86%.

CONCLUSIONS

Thus thesis presented a software to reconstruct vehicle dynamics from inertial and GNSS data. I developed an algorithm to integrate physical quantities with a technique suitable also for not equally time separated records. Both acceleration and angular velocity are integrated; angular position is composed using quaternion performance advantages. Trajectory obtained from accelerometer and gyroscope is improved additionally by GNSS data, leading to a more precise and smooth path. The project has been implemented in Python, mainly take advantage of Numpy and Pyquaternion libraries functionalities. I created a standalone Blender add-on which auto-installs its dependencies and offers a graphical interface. Most of the code has been checked with automated testing.

9.1 Future Development

Several improvements can be implemented in following releases:

- the angular position reconstruction can still be improved significantly using some inference techniques (fusing data from other sensors, refining the analytical model of the dynamics) and also it tends to drift away due to numerical error. Cross-validating vehicle angular positions with accelerometer and GNSS can provide a better assessment on the error it has been accumulated and that must be corrected. Moreover, online maps API can be used to have an approximation of road slope;

- gyroscope offset can be additionally analyzed through unused data coming from the temperature sensor. In fact, it is well known that this erroneous behavior is strongly correlated with the sensor's thermal state;

- vertical positioning can be strongly improved by using barometer;

- data from sensors installed on different positions in the same vehicle can be merged, at the beginning using their known relative positions, and later even by self-calculating them;

- usage of approximation instead of interpolation for GNSS precise positioning can be used to reduce error when vehicle is in a stationary regime;

- to improve integration algorithm stability it is possible to explore different change of basis of second grade polynomial used;

- implementation of symplectic integration schemes which, due to their non-dispersive nature, can help improving the reconstruction;

- implementation of a new file format, based on HDF, to drastically improve I/O performances

- parallelize execution, expecially in delta integration routine where it can be distributed on an large number of cores.

RINGRAZIAMENTI

Vorrei ringraziare tutto il gruppo di Fisica dei Sistemi Complessi, in specifico Stefano Sinigardi, Alessandro Fabbri, Nico Curti e Raffaele Pepe per il supporto e per il bel ambiente lavorativo in cui sono stato.

Vorrei ringraziare la mia famiglia per l'aiuto che mi hanno dato e perché mi hanno permesso di studiare seguendo le mie passioni.

REFERENCES

[1] Mike Boyle.
numpy-quaternionn.
May 2018.
URL: https://github.com/moble/quaternion.

[2] Bruce Cameron Reed.
"Numerically Integrating Irregularly-spaced (x, y) Data".
In: *The Mathematics Enthusiast* 11.3 (2014), pp. 643–644.

[3] Giulio Casciola.
Dispensa di CALCOLO NUMERICO.
METODI NUMERICI PER IL CALCOLO.
Ufficio F2 Dipartimento di Matematica, P.zza di Porta S.Donato, 5 Bologna, Sept. 2017.

[4] Z. Diao, H. Quan, L. Lan, and Y. Han.
"Analysis and compensation of MEMS gyroscope drift".
In: *2013 Seventh International Conference on Sensing Technology (ICST)*.
Dec. 2013,
Pp. 592–596.
DOI: 10.1109/ICSensT.2013.6727722.

[5] David Eberly.
Rotation Representations and Performance Issues.
2016.

[6] Giuliana Figna.
Redazione di presentazioni e dispense. Un esempio: I quaternioni nella computer-graphics.
2014.
URL: http://amslaurea.unibo.it/6701/.

[7] *get-pip.py*.
Python Packaging Authority.
URL: https://bootstrap.pypa.io/get-pip.py.

[8] *numpy.cumprod()*.
Scipy.org.

URL: https://docs.scipy.org/doc/numpy/reference/generated/numpy.cumprod.html.

[9]   *python*.
        Python Software Foundation.
        URL: https://www.python.org/.

[10]   S. Tadic, R. Stancic, L. V. Saranovac, and P. N. Ivanis.
        "Vehicle Collision Reconstruction With 3-D Inertial Navigation and GNSS".
        In: *IEEE Transactions on Instrumentation and Measurement* 66.1 (Jan. 2017), pp. 14–23.
        ISSN: 0018-9456.
        DOI: 10.1109/TIM.2016.2619018.

[11]   Kieran Wynn.
        pyquaternion.
        Dec. 2017.
        URL: https://github.com/KieranWynn/pyquaternion.