POLITECNICO DI TORINO

Master's Degree Thesis in
Computer Engineering

Prediction of road users behaviour and dynamic motion control in ROS

Academic advisor: Massimo Violante
Candidates: Assunta Petti
Luigi Tutisco
Internship tutor: Simone Nava

A.A. 2018/2019

Index

1. Introduction

1.1 Abstract

An automated driving system is a combination of various components that are capable to perceive, make decisions and operate an automobile: everything is done by electronics and machinery, the human driver leaves all the responsibilities to the system.
This system introduces automation into road traffic and includes handling of the vehicle, destination and awareness of surroundings.

The focus of this thesis is to implement a dynamic motion planning algorithm in order to control continuously a vehicle from a start to a goal position avoiding collisions with known obstacles, both static and dynamic (people on the streets, other cars, bikes and so on).
As the title explains, the first goal is to predict the trajectory of other road users so that the second goal can be achieved: the vehicle is able to evaluate a plan to reach a goal.
This plan takes into consideration the actual position of the obstacles and their future positions in order to understand if a certain path could lead to collision before even starting the movement, and so generating the best trajectory that ensures safety, comfort and efficiency.

The software has been developed in ROS (Robot Operating System) which allows access to the largest ecosystem and community currently existing in the field of robotics.

1.2 Overview

This work is based on the thesis of two students. They studied a way to build a platform for research studies or for other students' thesis. This platform makes it easier to implement new ideas for autonomous driving, having the possibility to access car data recorded from a customized BMW 5 Series vehicle.

When the work for this thesis starts, a static motion planner is already implemented. It takes as input the goal that has to be reached and evaluates a path using as information only static obstacles. Once the path is evaluated, it does not change until the vehicle reaches the goal.
This basically means that, if there are moving obstacles, their current position could change in such a way that the original path could lead to a crash.

The practical problems to be solved are:

- create a costmap that can handle dynamic objects: a costmap is a 2D grid that highlights obstacles. In the current implementation, only static obstacles and the temporary position of dynamic ones are mapped. The improved costmap will be then sent to the motion planning node that will understand how the vehicle should move;
- predict future trajectory of other road users and include it in the costmap;

- handle collisions: the ego-vehicle will have to stop or at least to slow down if it recognizes that the trajectory of another road user can be safety critical for its movement; then it will have to recompute the path to reach the destination.

A block diagram that explains the decision process that the ego-vehicle has to take is shown in Fig. 1:

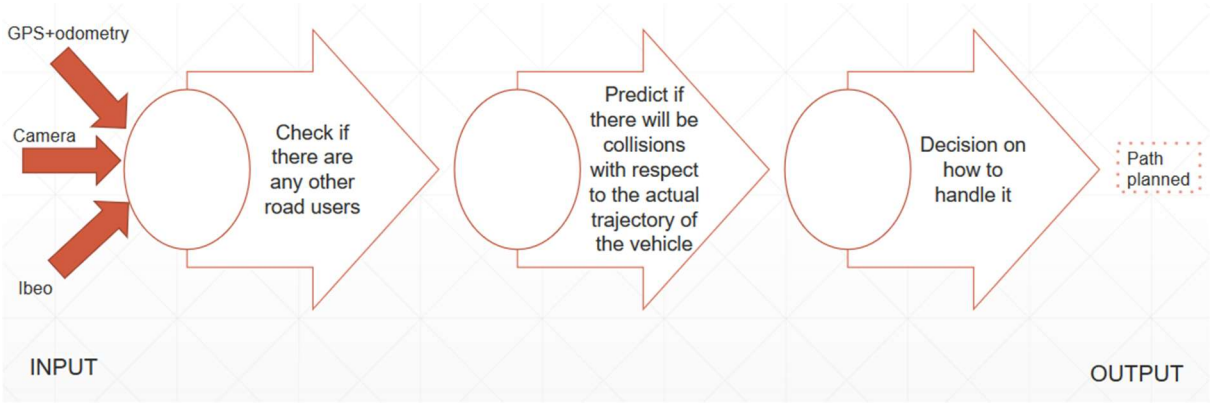

Fig. 1: decision process the ego-vehicle has to take

- The inputs of the process come from the sensors; their task is to accumulate as much information as possible on the surrounding environment. Then the information from sensors has to be mapped inside the costmap: cells corresponding to obstacles, both static and dynamic, have to be marked as *not-traversable.*

  Due to the fact that the costmap is, as a matter of fact, a map of the environment divided in cells which have a cost information, the not-traversability can be done setting a high cost for a group of cells.
  - The **cost** is a value that goes from 0 to 255; the software that handles the costmap is anyway capable of representing only three levels of cost. Specifically, each cell in this structure can be either free, occupied, or unknown. When a cell is recognized as occupied, a “lethal obstacle” cost is assigned to it and inflation can be performed.
  - **Inflation** is the process of propagating cost values out from occupied cells that decrease with distance out to a user-specified inflation radius.
  - The **group of cells** has to represent the actual obstacle that is in the surroundings of the ego-vehicle. The sensors are capable to evaluate width and length of the obstacles, then the costmap is capable to compute the cells that have to be taken into consideration for those specific dimensions.

- A trajectory prediction algorithm has to forecast the **future positions** that the **moving objects** are about to take. This information has also to be mapped inside the costmap. The prediction is done using a Kalman filter and applying a specific motion model.

  - The motion model has the objective to model the system of the actual moving obstacle.
    Numerous motion models (with different degrees of complexity) exist.

Linear motion models assume a constant velocity (CV) or a constant acceleration (CA). Their linearity of the state transition allows a good propagation of the state probability distribution.
Despite of these properties, however, these models can prove to not handle all the possible profiles of motion, in some cases also having a significant error when used within the Kalman filter to predict future positions of objects.

Another model then has been identified: it is not a classical model but it seemed really promising. The constant jerk (CJ) model (the jerk is the derivative of the acceleration) can show to have the best performances among the considered models, and, once applied within the Kalman filter, it can prove to well predict the trajectories of the obstacles.

- The result of these predictions has to be that, in addition to information from the sensors, every obstacle in the surroundings of the ego-vehicle has also its predicted path. For each object this information can be displayed on the costmap marking as occupied also the cells corresponding to the future positions of obstacles.
  To have then a better visualization of the actual path that has been predicted, the cells can be joined to form a polygonal chain.

- After all possible information is obtained, a motion planning algorithm can safely evaluate a route from the start to the goal and compute speed commands for the vehicle depending on the chosen path. This path constitutes the output of the decision process.

  - A few algorithms are studied that have to take into account the dynamic constraints to which a car is subjected and its non-holonomicity[1] property. To be noticed that implementing only this last property is an error because it happens that the algorithm works **but** only for differential-drive robots with turning radius[2] equal to 0. Basically it becomes suitable for robots that are capable to rotate on themselves, so not for cars.

    This constitutes a great limitation because it happens that when a destination is chosen such that the yaw angle of this one is perfectly perpendicular to the actual one the ego-vehicle has, the car can result to be stuck and no movement can be produced, even if the goal could be simply reached performing a few maneuvers.

    This leads to the implementation of an algorithm that can take into account the possible turning radius of a vehicle. In this way, what it is desired to have is an accurate motion model of the car and, as output, the right planned path that has to be followed.

[1] A robot is **holonomic** when the number of controllable degrees of freedom is equal to the total degrees of freedom. If not, it is non-holonomic.
[2] The **turning radius** of a vehicle is the radius of the smallest circular turn that the vehicle is capable of making, shown in Fig. 2.

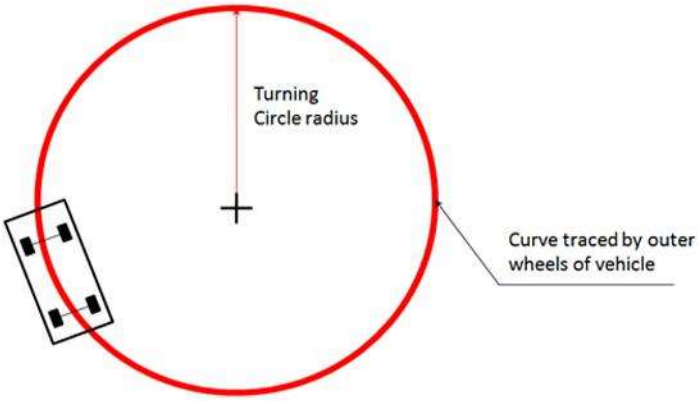

Fig. 2: visualization of a possible turning radius of a vehicle

- Since dynamic obstacles have to be handled, the implemented motion planner has to be a dynamic one: this means that it has to be capable to re-evaluate the best path to reach the goal depending if new information happens to be on the map. In fact, if a moving object appears in the surroundings of the ego-vehicle, or a previously recognized one moves away from the previous position, it has to be evaluated if this will have an impact on the actual path the ego-vehicle is taking. If so, a new one has to be found.

The expected result is thus that the vehicle is capable to evaluate a safe path to be taken and to react if a new information about an obstacle arrives: it has to be capable to slow down or to stop the motion and re-evaluate the best path if necessary.

1.3 Objective Software

This work would not have been possible without the help of our company Objective Software Italia SRL, proud part of Luxoft: this is a global consulting partner that offers end-to-end digital solutions to solve clients' complex business challenges. It boasts expertise in automotive, financial services, healthcare, life sciences, telecommunications, and other industries.
With headquarters in Zug, Switzerland and 42 global locations, it operates in 21 countries across five continents, providing clients with robust global delivery platform.

They gave us the opportunity to work on the platform and to collaborate with very smart people that helped and taught us.

Fig. 3

2. Description of the platform

2.1 ROS Robot Operating System

The platform is based on ROS, **Robot Operating System**: this is a collection of software frameworks designed for robot software development [1]. ROS is not an operating system, anyway it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionalities, message-passing between processes, and package management.
Despite the importance of reactivity and low latency in robot control, ROS, itself, is *not* a real-time OS (RTOS), thus leading to the creation of ROS 2.0.

Fig. 4

- In ROS a wide range of different programming languages can be used, like C++, Python, Java and others.
- ROS is not an integrated development environment: it can be used with most popular IDEs but also with a text editor and the command line, without any IDE.

ROS has two levels of concepts: the Filesystem level and the Computation Graph level.

The Filesystem level is composed of:

- **Packages**: Packages are useful units that organize software in ROS and constitute the most atomic item that can be built and released in ROS. A package may contain ROS runtime processes (*nodes*), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together.
- **Package Manifests**: Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.
- **Repositories**: A collection of packages that have a common VCS[3] system. Packages which share a VCS share the same version and can be released together. Repositories can also contain only one package.
- **Message (msg) types**: descriptions that define the data structures for messages sent in ROS.
- **Service (srv) types**: descriptions that define the request and response data structures for services in ROS.

[3] **Version control system** is the system managing changes to documents, computer programs, large web sites, and other collections of information.

The *Computation Graph level* is the layer that takes care of ROS processes and how they cooperate with each-other to evaluate data:

- **Nodes**: Nodes are processes that perform computation. ROS is built to be modular at a fine-grained scale; a robot control system usually comprises many nodes.
- **Master**: The ROS Master provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- **Messages**: Nodes communicate with each other by passing messages. A message is simply a data structure, comprising typed fields.
- **Topics**: Messages are sent with a transport system with publish/subscribe semantic. One node *publishes* a message into a given topic. Another node that is interested in the content of that message *subscribes* to the topic and stays there waiting for messages to be published. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers do not know about the existence of each other. In this way the production of information is decoupled from its consumption.
- **Services**: The publish/subscribe model is very useful when a communication many-to-many is required; it is not suited anyway for request/reply communication that is one-to-one. This is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.
- **Bags**: Bags are collections of messages from ROS topics that can be saved and re-played, very useful when testing algorithms.

Within this architecture it is possible to do decoupled operations since every element has a name. Nodes, topics, services, all have names, thus making possible to have a distributed system without any handling complexity.
The relations between these entities is shown in Fig. 5.

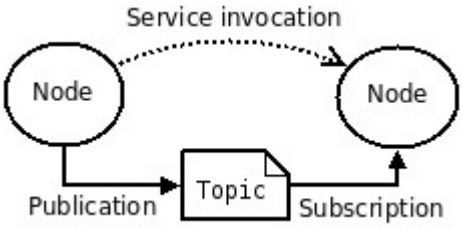

Fig. 5

2.2 Odometry

The car is equipped with proprioceptive sensors needed for odometry.

Proprioceptive sensors are largely used in robotics to measure the internal state of the robot, namely: the position and the joint speed, and the torque at the joint [2].
Odometry is the use of this internal state to measure the changes in position over time, taking into account that measuring time is easy using the internal clock of the embedded computer.
Odometry is a form of *localization*: it is used in robotics by some legged or wheeled robots to estimate their position relative to a starting location, the robot must determine its position in the environment. In odometry position is determined by measuring the change from the robot's known initial position, while localization refers to the determination of the position of a robot relative to the known positions of other objects such as landmarks or beacons.

A disadvantage of odometry is that the measurements are indirect, relating the power of the motors or the motion of the wheels to changes in the robot's position. This can lead to significant errors since the relation between motor speed and wheel rotation can be very nonlinear and vary with time. Furthermore, wheels can slip and skid so there may be errors in relating the motion of the wheels to the motion of the robot.

Here there are some of the most useful computation that can be done using proprioceptive sensors:

- distance from speed and time: when a relation between motor power and velocity *v* has been determined, the robot can compute the distance moved by *s* = *vt*. If it starts at position (0, 0) and moves straight along the *x*-axis, then after *t* seconds its new position is (*vt*, 0).

- odometry with turns: suppose that the robot turns slightly left because the right wheel moves faster than the left wheel.
  In Fig. 6, the blue dot is the left wheel, the red dot is the right wheel, and the black dot is the center of the robot which is halfway between the wheels. The *baseline b* is the distance between the wheels, and *dl*,*dr*,*dc* represent the distances moved by the two wheels and the center when the robot turns.
  dl and *dr* can be measured computing the number of rotations counted by the wheel encoders[4].

  If the radius of a wheel is *R* and the rotational speeds of the left and right wheels are ω_l , ω_r revolutions per second, respectively, then after *t* seconds the wheel has moved:

  $$d_i = 2\pi R\omega_i t \;\; i = l, r$$

  The task is to determine the new pose of the robot after the wheels have moved these distances but it can be done using a geometrical approach.

[4] Position sensors that provide an electrical signal proportional to the displacement (linear or angular) of a mechanical member with respect to a reference position. There are two main types of encoders: absolute and incremental.

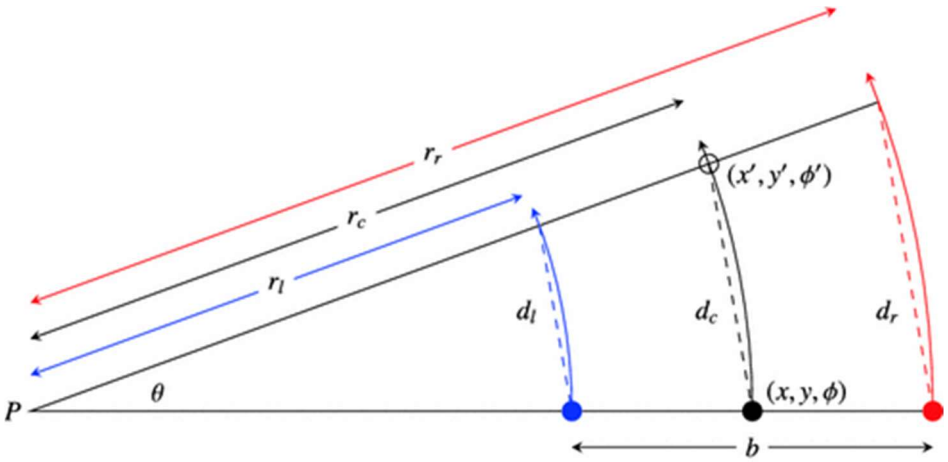

Fig. 6: model of a car that is turning

To compensate for the errors introduced by odometry and to construct a map of the surrounding environment, heteroceptive sensors are used that are able to measure the position of the robot with respect to the external environment.
The most commonly used ones are: the force sensors, the tactile sensors, the proximity sensors and the vision sensors. Their use plays a central role for the control of the interaction with the environment, for the avoidance of obstacles, for the location of mobile robots and for navigation in unknown environments.

2.3 Ibeo sensors

The platform for which the work of this thesis is developed is equipped with laser scanners and cameras as heteroceptive sensors.
The laser scanners are tools that can measure at very high speed the position of hundreds of thousands of points, which define the surface of the surrounding objects. The result of this relief is a very dense set of points that is called a "**point cloud**".

There are several parameters useful for defining and evaluating the characteristics of a laser scanner instrumentation:

- range: maximum distance that the scanner can measure;
- speed: number of points acquired in every second;
- accuracy: degree of conformity of a quantity measured with respect to the real value;
- precision: ability of the instrument to return the same value in subsequent measurements;
- laser class: this is the danger of the laser beam emitted by the instrument; it goes from class I (completely harmless) to class IV (very dangerous).

The sensors used are LIDAR (Light Detection and Ranging) Ibeo sensors that are based on the Time of Flight method: in TOF laser scanners, the point cloud is generated by calculating the time taken by the laser beam to travel the distance from the emitter to the affected subject and vice versa, knowing that the speed of propagation of the laser beam is equal to that of the light .
Knowing the vertical and horizontal angle of the beam emission, it can be defined the coordinates of the measured point. These laser scanners are characterized by the ability to detect very distant data, even reaching objects at a radius of 6 km from the sensor. Anyhow LIDAR sensors in the automotive industry can reliably detect objects within ranges of up to 300 meters.

LIDAR sensors have a high angular resolution and a wide field of view which is invaluable for precise and reliable mid-range detection. Hence, the scanned data can be recorded and with the help of software tools, they can be used to create a model of the vehicle's surroundings. This model includes the position and velocity of other road users as well as information on the road infrastructure. Tracking algorithms, implemented on the platform, are then able to determine the shape, the yaw rate, heading direction, classification of objects (bikes, cars, people on the streets..) and many other attributes.
LIDAR sensors are able to detect free spaces in their entire field of view [3] and, as an active system, they are independent from any light conditions: in this way, they can be used at any time of the day.

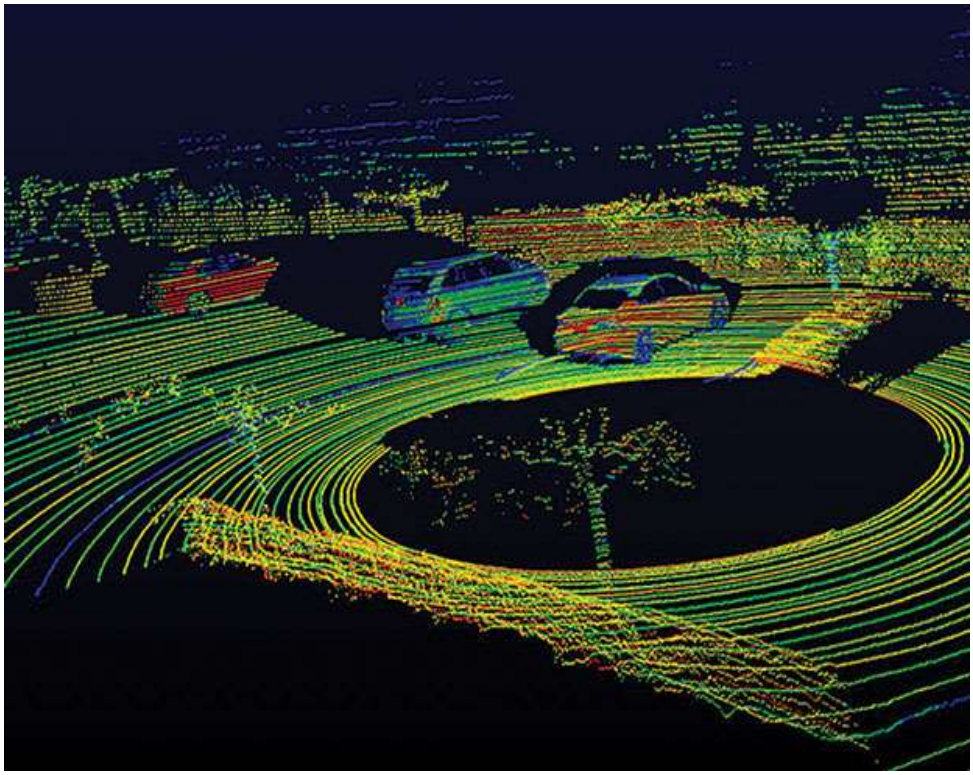

Fig. 7: Example of Ibeo sensor scanning result

2.4 Object detection

The objects the sensors recognize are point clouds so without information about the real shape of the objects or their classification. This is a post-process done using tracking algorithms.
The first step to analyze objects is clustering: **clustering** is the grouping of a set of objects in a way that elements in the same group (called a **cluster**) are more similar (in some sense) to each other than to elements in other groups (clusters).

Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data pre-processing and model parameters until the result achieves the desired properties.
The basic Euclidean clustering algorithm is used that, as the name suggest, is done taking into account the Euclidean distance from the center point of all clusters. A point is considered part of a certain cluster if the distance to its center point is the lowest with respect to the other clusters.

The algorithm executes this process:

1. Create an empty queue Q of points
2. For each point in the original point cloud, add the point to the queue Q and:
   (a) Extract a point P from the Q. Add it to cluster Ci
   (b) Add all the points within a certain radius from P to the queue Q
   (c) Go to 2a until the Q is empty. When empty, cluster Ci is completed
3. Repeat for each remaining point in the original point cloud

Euclidean clustering is performed at every LIDAR scan. Anyway, given two point clouds A and B and the two set of clusters CA and CB, it is not directly possible to define a correspondence between clusters of CA and CB.
Comparing the distance between the centroids of the clusters may not be enough (think about two cars moving close to each other). Also, sometime it is useful to know not only the position but also the velocity of a certain cluster of objects. The way to have this is to apply a Kalman filter to each cluster.

Each cluster will have its own state (position and velocity) and the information is updated with the output of every scan. When a new set of cluster is generated, the problem of assignment arises: first step is to compare each cluster to each previously tracked cluster for position and velocity compatibility. This comparison may result in a match or a rejection. In case of match, the state of the Kalman filter is updated with the new measurements.
This gives knowledge of the speed and direction of other cars relatively to the ego-vehicle.

Classification is performed in a probabilistic way, using the Hungarian Algorithm. By defining a matrix of matching likelihood, the Hungarian Algorithm chooses the most probable classifications. A bounding box around the cluster is generated, and that is used for estimating the full space occupied by the vehicle, with some extra margin.
Thanks to the Kalman filter, it is possible to know the relative speed of each detected box [4].

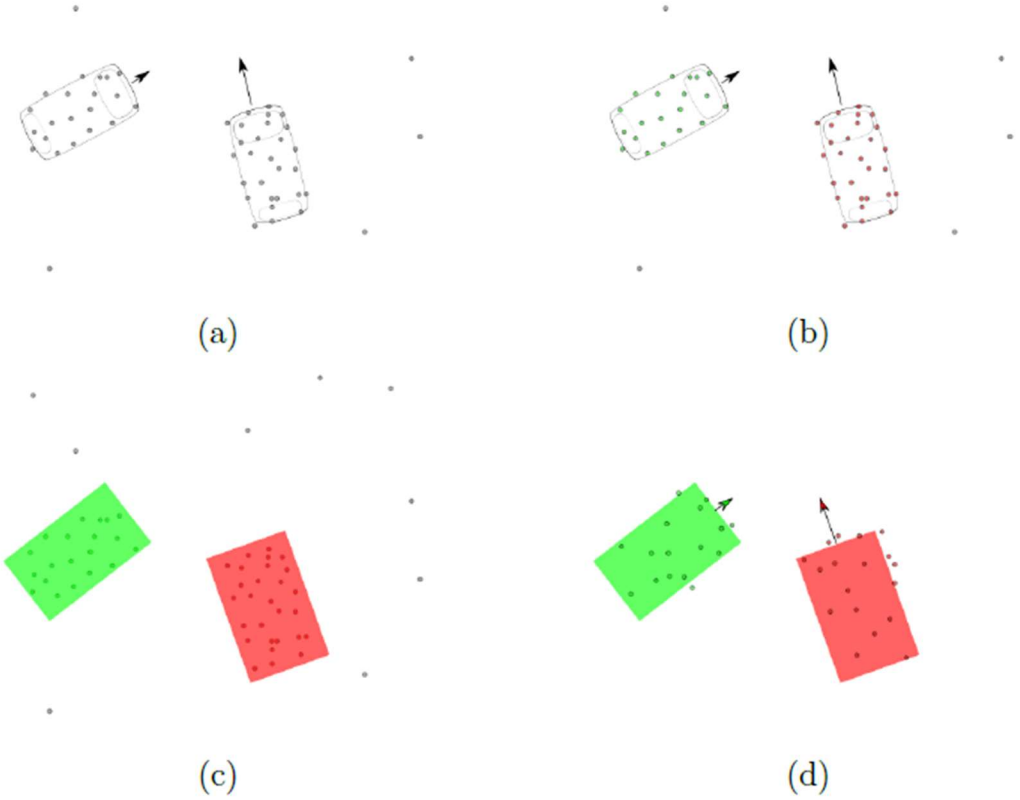

Fig. 8: Kalman Filter tracking.
After clustering is performed (a and b), instances of the filter are initialized (c).
State is then updated based on the data of new point clusters (d).

2.5 Navigation

2.5.1 Costmap

In the actual implementation, the platform doesn't let the vehicle move freely.
Collisions with objects are effectively considered, even if only static ones: it can be said that a *static* motion planner is implemented based on a 2D occupancy map, or costmap.
2D because, in the automotive case, vehicles move mostly horizontally and height is not usually a problem. The map assigns a value proportional to the 'traversability' of each cell. Such value is used for soft-constraints or favour some trajectory against others.

Since the platform is ROS based, the costmap_2d package is used [5]. This package provides a configurable structure that maintains information about where the robot should navigate in the form of an occupancy grid.

The costmap uses sensor data and information from the static map to store and update information about obstacles in the world through the costmap_2d::Costmap2DROS object.

Usually, in order to use a class or a function in another class it is necessary to include the header file. Here instead the pluginlib package of ROS can be used: it provides an alternative method that allows referencing a class/function without including the header file. Using pluginlib, the costmap package does not know the layer classes but at runtime it can use function from each of these classes: pluginlib exports some features from layer classes and makes these visible to the costmap.

The costmap automatically subscribes to sensors topics over ROS and updates itself accordingly. Each sensor data is used to either mark (insert obstacle information into the costmap), clear (remove obstacle information from the costmap), or both.
A marking operation is an operation that increases the cost of a cell.
A clearing operation consists of ray-tracing through a grid, from the origin of the sensor outwards, and decreasing the cost of the cells no more recognized as occupied.

While each cell in the costmap can have one of 255 different cost values, only three levels can be represented. Specifically, each cell in this structure can be either free, occupied, or unknown:

- Columns that have a certain number of occupied cells are assigned a costmap_2d::LETHAL_OBSTACLE cost;
- Columns that have a certain number of unknown cells are assigned a costmap_2d::NO_INFORMATION cost;
- Other columns are assigned a costmap_2d::FREE_SPACE cost.

The costmap changes at the rate specified by the update_frequency parameter. Each cycle, sensor data comes in, marking and clearing operations are performed and cost values are assigned as described above.

Sensor data has to be inserted into the costmap and to do so the Costmap2DROS object makes extensive use of tf. tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure and keeps track of all these frames over time, letting the user transform points, vectors, etc between any two coordinate frames at any desired point in time.
A robotic system typically has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc., as shown in Fig. 9.
tf assumes that all transforms between the coordinate frames specified by the global_frame parameter, the robot_base_frame parameter, and sensor sources are connected and up-to-date [6].

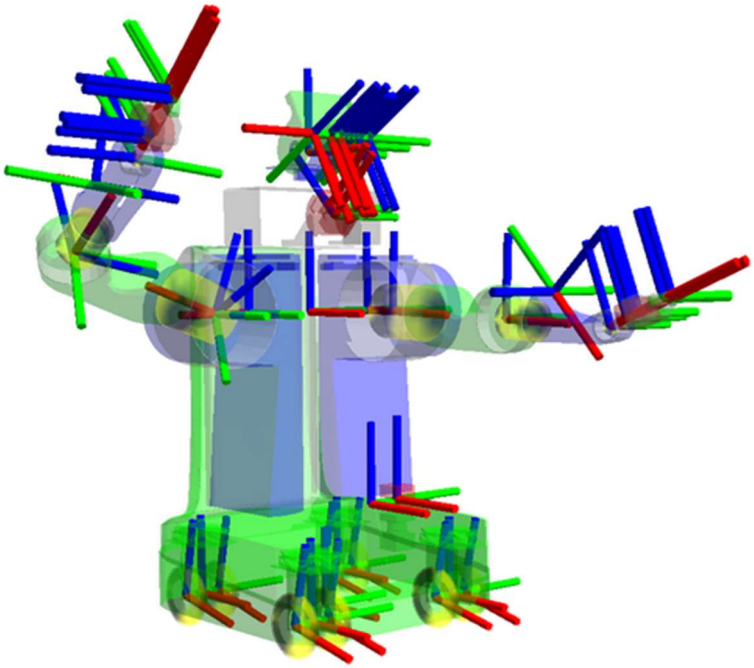

Fig. 9: a representation of a tree of reference frames

The costmap is a powerful method with which the bounding boxes representing the road users, generated by the detection pipeline described in the previous paragraph, can be marked as non-traversable. This is the first step to let the ego-vehicle recognize obstacles.

2.5.2 A*

A* (A star) path planning algorithm is already implemented on the platform, provided by Autoware for navigation.
A* is an informed search algorithm, or a best-first search: it is formulated in terms of weighted graphs in a way that it considers to have the starting position as a node of the graph; from there it has to find a path to a defined goal, always a node, having the smallest cost (least distance travelled, shortest time, etc.). At the beginning it initializes a tree of paths from the start node to the destination, then extends the tree until the termination criterion is satisfied.
More information about the algorithm is given later in this document.

The bearing of the vehicle during traversal is not considered in the simple A* but the algorithm implemented in Autoware can be seen as three dimensional: each cell is identified by a (x, y, θ) tuple. This way, the generated path has also a yaw variable associated to each position. The heuristic algorithm is also slightly modified in order to take into account reverse and turns [4].

The actual implementation only evaluates the path, to be undertaken, **once** at the beginning of the evaluation of the path. Simulations have shown that, if a moving obstacle happens to be in the middle of the path, the ego-vehicle does nothing to avoid it, as can be seen in Fig. 10.

This is the main reason that led to the definition of this thesis.

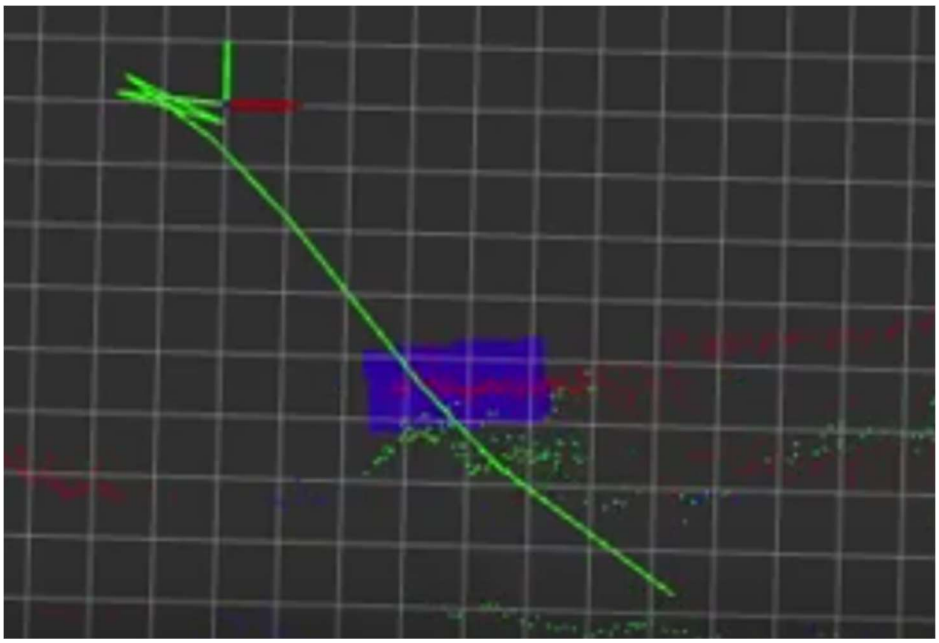

Fig. 10: the blue rectangle is the moving obstacle. It happens to be in the middle of the path that the ego-vehicle is supposed to follow. A crash will occur.

3. State of the art

3.1 Levels of autonomous driving systems

There are five different levels of automation for what regards **autonomous driving systems**:

0) No automation

The driver controls the vehicle without any help from the system, it does not interfere.

1) Driver assistance

The driver is in control of the vehicle, but the system can modify the speed and steering direction of the vehicle. BMW Personal CoPilot driver assistance systems are an example of this, with the help also of the Active Cruise Control with Stop&Go function, which independently adjusts the distance to the car in front of you. Then there is the Collision and Pedestrian Warning with City Brake Activation, which automatically brakes to prevent collisions.

2) Partial automation

The driver must be able to control the vehicle if corrections are needed, but he is no longer in control of the speed and steering of the vehicle. This is actually done by BMW remote-controlled parking function that is capable to pull the cars into tight spots without a driver, by Tesla´s autopilot feature and by the DISTRONIC PLUS system created by Mercedes-Benz.

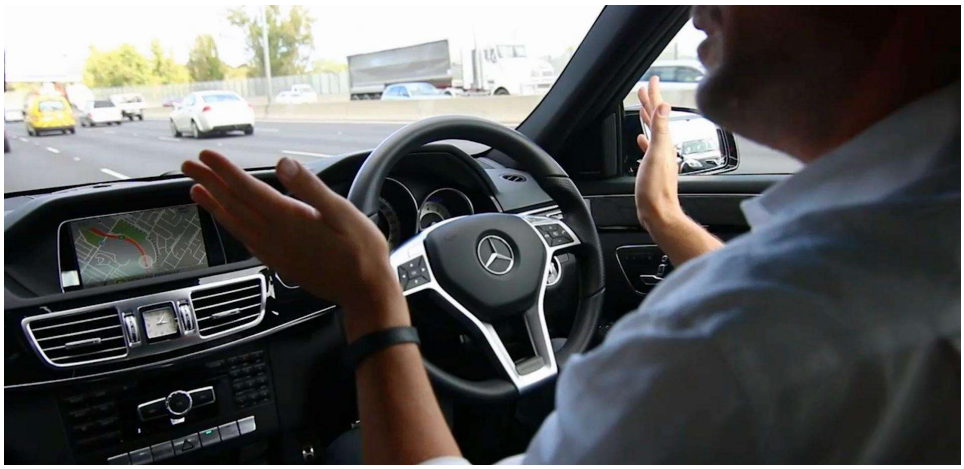

Fig. 11: Mercedes Benz - Distronic Plus

3) Conditional automation

The system is completely capable to control vehicle speed and steering and to monitor the environment, but the driver must be able to take over control within a few seconds for exceptions on the road, such as at construction sites.

4) High automation
The system is in complete control of the vehicle and human presence is no longer needed even if a human driver can still request control, and the car still has a cockpit. Basically the car can handle the majority of driving situations independently, also highly complex urban driving situations like the sudden appearance of construction sites.

The driver must anyway remain capable to drive and to take control if needed. The driver could have the chance to sleep temporarily but, if a warning alarm is ignored, the car has to enter a safety condition, for example by pulling over.
An example of a system that falls into this category is the Google car, Fig. 12.

Fig. 12: Google car

5) Full automation
The system is in complete control of the vehicle and human presence is no longer needed. Drivers do not have to be fit to drive and don't even need to have a license. The car performs all driving tasks – there isn't even a cockpit. Everyone in the car is a passenger. At this level, a high attention on safety has to be put and also, within populated areas, speed has not to reach high values. Currently, there are no driving systems at this level. [7]

3.2 Motion planning

In this environment, the **dynamic motion planning** assumes a really important role. Its main goal is to produce a continuous motion that connects a start configuration and a goal configuration while avoiding collisions with known obstacles, both static and dynamic (people on the streets, other cars, bikes and so on), and so generating the best trajectory that ensures safety, comfort and efficiency.

Examples of decisions that has to be made are avoiding walls and not falling down stairs. A motion planning algorithm would produce the speed and turning commands that have to be sent to the robot's wheels.

The set including position and orientation of the robot according to a fixed coordinate system is termed the *configuration* vector. Consequently, the set of all the configurations of the vehicle constitute the *configuration space*.

A **complete** motion planner is such that, in finite time, puts as output a solution or at least reports that there is none. The performance of a complete planner is assessed by its computational complexity.
Resolution completeness establishes that the planner is guaranteed to find a path if the resolution of an underlying grid is fine enough. As resolution it has to be considered the length of one side of a grid cell: the greatest the resolution, the higher the computational complexity.
Probabilistic completeness establishes that the more the algorithm runs the more the probability that the planner fails to find a path, if one exists, asymptotically approaches zero. The computational complexity depends on the rate of convergence.
Incomplete planners do not always produce a feasible path when one exists. Sometimes incomplete planners do work well in practice.

Here there are a few important motion planning algorithms.

3.2.1 Dynamic window approach

The **dynamic window approach** is an online collision avoidance strategy for mobile robots. It derives directly from the dynamics of the robot and deals with the constraints imposed by limited **velocities** and **accelerations** of the robot.
It consists of two main phases, the first consists in **generating a valid search space**, and the second one in **selecting an optimal solution** in the search space.

The **dynamic window** is a portion of the path on which to concentrate the major effort in optimization: the goal of optimization is to select a heading and velocity that brings the robot to the destination, avoiding obstacles along the way.

The **search space** is constructed from the set of velocities which produce a safe trajectory (that allow the robot to stop before colliding), taken from the set of velocities the robot can achieve in the next time slice due to its dynamics ('dynamic window').

The **optimal velocity** is selected to maximize the robots clearance, which is the distance from obstacles, maximize the velocity and obtain the heading closest to the goal.

These are the steps that the algorithm does:

1. First, the desired velocity to the goal can be calculated based on the ego-vehicle current position, and the destination. (e.g. go fast if the goal is far away, slow if it is close).
2. Select the allowable velocities (linear 'v', and angular 'ω') given the vehicles dynamics, e.g. allowable v ranges between: $[v - at, v + at]$, likewise for angular velocity.
3. Search through all the allowable velocities.
4. For each velocity, determine if a collision would be possible to happen along the trajectory computing distances from obstacles.
5. Determine if the distance to the closest obstacle is within the robots braking distance. If the robot will not be able to stop in time, disregard this proposed robot velocity.
6. If the velocity is 'admissible', the values required for the objective function can be now calculated. In the case of this thesis, the robots heading and clearance.
7. Calculate the 'cost' for the proposed velocity. If the cost is better than anything else so far, set this as best option.
8. Finally, set the robots desired trajectory to the best proposed velocity.

A pseudo-code of these steps is listed in Fig. 13.

```
BEGIN DWA(robotPose,robotGoal,robotModel)
   desiredV = calculateV(robotPose,robotGoal)
   laserscan = readScanner()
   allowable_v = generateWindow(robotV, robotModel)
   allowable_w  = generateWindow(robotW, robotModel)
   for each v in allowable_v
      for each w in allowable_w
      dist = find_dist(v,w,laserscan,robotModel)
      breakDist = calculateBreakingDistance(v)
      if (dist > breakDist)  //can stop in time
         heading = hDiff(robotPose,goalPose, v,w)
         clearance = (dist-breakDist)/(dmax - breakDist)
         cost = costFunction(heading,clearance, abs(desired_v - v))
         if (cost > optimal)
            best_v = v
            best_w = w
            optimal = cost
    set robot trajectory to best_v, best_w
END
```

Fig. 13

The **search space** of the possible velocities is reduced in three steps:

- **Circular trajectories**: The dynamic window approach considers only circular trajectories (curvatures) uniquely determined by pairs (v,ω) of translational and rotational velocities. This results in a two-dimensional velocity search space.

- **Admissible velocities**: A pair (v,ω) is considered admissible, if the robot is able to stop before it reaches the closest obstacle on the corresponding curvature. The restriction to admissible velocities ensures that only safe trajectories are considered.
- **Dynamic window**: The dynamic window restricts the admissible velocities to those that can be reached within a short time interval given the limited accelerations of the robot.

The objective function, the *navigation function,* that has to be maximized, is:

$$G(v,\omega) = \sigma(\alpha \cdot heading(v,\omega) + \beta \cdot dist(v,\omega) + \gamma \cdot vel(v,\omega))$$

- $heading(v,\omega)$: **Target heading** is a measure of progress towards the goal location. It is maximal if the robot moves directly towards the target.
- $dist(v,\omega)$: **Clearance**, it is the distance to the closest obstacle on the trajectory. The smaller the distance to an obstacle the higher is the robot's desire to move around it.
- $vel(v,\omega)$: **Velocity**, it is the forward velocity of the robot and supports fast movements.

The function σsmoothes the weighted sum of the three components and results in more side clearance from obstacles. [8]

As a result, the possible sets of velocities are:
- V_d : the set of the velocities for the dynamic window, which contains only the velocities that can be reached within the next time interval
- V_a : the set of velocities (v,ω) which allow the car to stop without colliding with an obstacle, so depending on the accelerations for braking, the admissible velocities
- V_s : the space of all possible velocities
- $V_r = V_s \cap V_a \cap V_d$: resulting search space for the velocities

There is however a problem with DWA, as shown in Fig. 14: sometimes the robot, or the car in this case, does not slow down in time and the result is that it is really difficult to enter a passage. A solution could be to use the 5-D planning that plans in the full $< x, y, \theta, v, \omega >$configuration space using A* and considers the robot's kinematic constraints [9]. The search space though is too large to be explored in real-time so a solution could be to restrict the full search space to ”channels" with subgoals to be reached, as shown in Fig. 15.

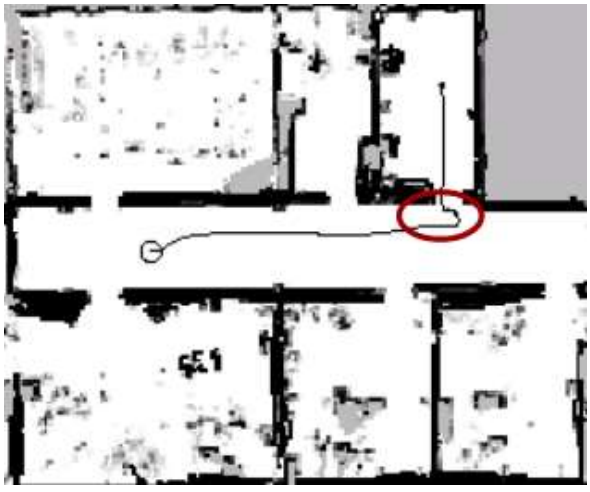

Fig. 14

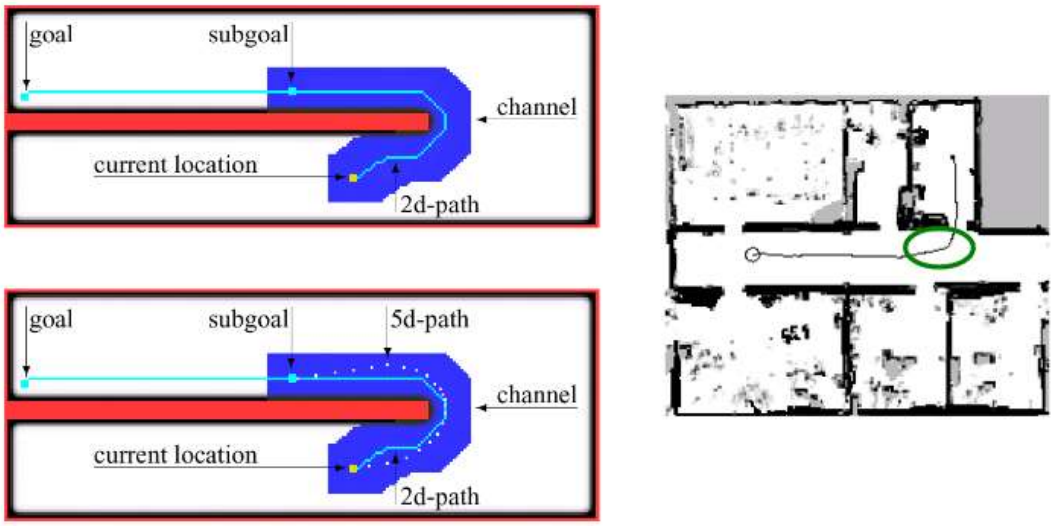

Fig. 15

3.2.2 Nearness Diagram Navigation

The **Nearness Diagrams** (ND) are the visual tools used to analyze the relations between the robot, the obstacle distribution, and the goal location. [10]

The PND (ND from the central Point) represents the nearness of the obstacles from the robot center and the RND (ND from the Robot bounds) represents the nearness of the obstacles from the robot boundary. The algorithm identifies gaps first; then, from these gaps, regions are obtained. Finally, one region is selected, the free walking area.

1. **Gaps**: can be identified in the obstacle distribution as discontinuities in the PND.
   A discontinuity exists between two adjacent sectors (i, j) if $| PND_i - PND_j |$ > 2 R, where R is the radius of the robot. The only interest is in the robot diameter (2 R) because that is the gap in which the robot fits.
   For a discontinuity between two sectors (i, j), if, for instance, $PND_i > PND_j$, it can be differed between a rising discontinuity from j to i and a descending discontinuity from i to j.
2. **Regions**: Two contiguous gaps form a region. Regions are valleys in the PND.
   Let S = {0, …, n-1} be the set of all sectors. A valley is a nonempty set of sectors of S, V = {l,...,r}, that satisfies the following conditions:
   (a) There are no discontinuities between adjacent sectors of V (i.e., there are no discontinuities within the valley)
   (b) There are two discontinuities in the extreme sectors of V (l and r)
   (c) At least one of the previous discontinuities is a rising discontinuity from l or from r where the rising discontinuities identify potential gaps to drive the robot within the region.

Fig. 16 shows the four regions (REGION 1, REGION 2, REGION 3 and FREE WALKING AREA) identified as valleys in the PND. These valleys do not have inside discontinuities [condition (a)], and they have a discontinuity in both extremes [condition (b)]. Furthermore, each valley has at least one rising discontinuity [condition (c)].

3. **Free walking area**: The region that has no discontinuities, so identifiable as "navigable", closest to the goal, and so that has to be taken. The process of selecting the free walking area is the following: first all valleys are identified; then the valley with the rising discontinuity closest to s_{goal} is selected; next, it is checked whether the candidate region is "navigable" and if so it is selected as free walking area; if it is not "navigable", another valley is selected and the process is repeated until a "navigable" region is found, or no region exists.

   It is still needed to differ between a wide free walking area and a narrow one. If its angular width is greater than a given quantity is wide, if not, it is narrow. Then, since the number of sectors of a valley is the angular width of the region, a valley is wide if the number of sectors is greater than $s_{max} = \frac{n}{4}$ and narrow otherwise where n is the total number of sectors.

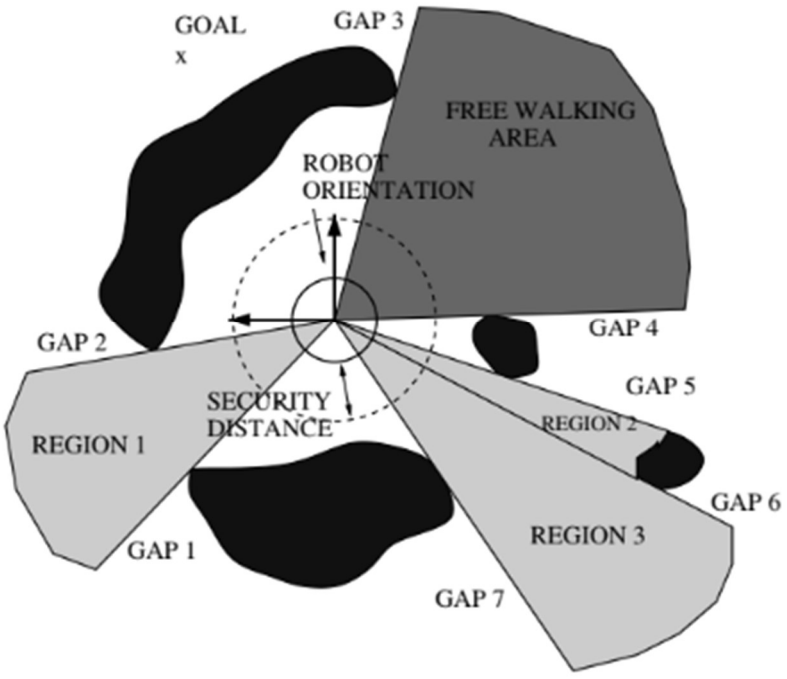

Fig. 16

A set of situations can be addressed. Each situation can be reached using criteria.
Criterion 1: Safety criterion: it is checked whether there are obstacles that exceed a security nearness in the RND (Low Safety), or not (High Safety). In Low Safety, the first two situations are obtained by applying the next criterion.
Criterion 2: Dangerous obstacle distribution criterion.

- **Low Safety 1 (LS1)**: The robot is in LS1 when the RND sectors that exceed the security nearness are only on one side of the rising discontinuity (closest to the goal sector) of the selected valley.
- **Low Safety 2 (LS2)**: The robot is in LS2 when the RND sectors that exceed the security nearness are on both sides of the rising discontinuity (closest to the goal sector) of the selected valley.

There are three situations in High Safety. The first one is obtained by applying the following criterion.

Criterion 3: Goal within the free walking area criterion.

- **High Safety Goal in Region (HSGR)**: The robot is in HSGR if the goal sector belongs to the selected valley.

If not, the last situations are obtained by applying the next criterion.

Criterion 4: Free walking area width criterion.

- **High Safety Wide Region (HSWR)**: The robot is in HSWR when the selected valley is wide.
- **High Safety Narrow Region (HSNR)**: The robot is in HSNR when the selected valley is narrow.

Next, a few other steps have to be done to evaluate the actions associated with each situation. The objective is to find simple control laws that produce the desired navigation behavior in each situation. Each action computes a motion command, divided in translational and rotational velocity.

3.2.3 Virtual Force Field (VFF)

The **Virtual Force Field (VFF)** method is an obstacle avoidance method, real time, for fast-running vehicles. [11]

A. The VFF method is based on a **2D Cartesian histogram grid** C for obstacle representation. Each cell (i,j) in the histogram grid holds a certainty value, c_{ij}, that represents the confidence of the algorithm in the existence of an obstacle at that location. The histogram grid is built and updated in a way such that it is incremented only one cell for each range reading. This results in a histogramic probability distribution, in which high certainty values are obtained in cells close to the actual location of the obstacle.
B. There is a window of cells always centered around the vehicle, defining a square region of C. This region is called the "**active region**" (denoted as C*), and cells that momentarily belong to the active region are called "**active cells**" (denoted as c_{ij}*).

Each active cell exerts a virtual repulsive force F_{ij}toward the robot. The magnitude of this force is proportional to the certainty value c_{ij}* and inversely proportional to d^{x}, where d is the distance between the cell and the center of the vehicle, and x is a positive real number. At each iteration, all virtual repulsive forces are summed up to form the **resultant repulsive force** F_r.

Simultaneously, a **virtual attractive force** F_t of constant magnitude is applied to the vehicle, "pulling" it toward the target. The summation of F_r and F_t yields the resulting force vector R as can be seen in Fig. 17.

The computational heart of the VFF algorithm is thus consisting in computation and summation of force vectors. This method is real-time because each reading from sensor is recorded and added to the histogram grid as soon as it is received; each one takes to the computation of the vector R immediately as they become available.
In practice, sensor data influence steering control thus resulting in fast reflexive behavior imperative at high speeds.

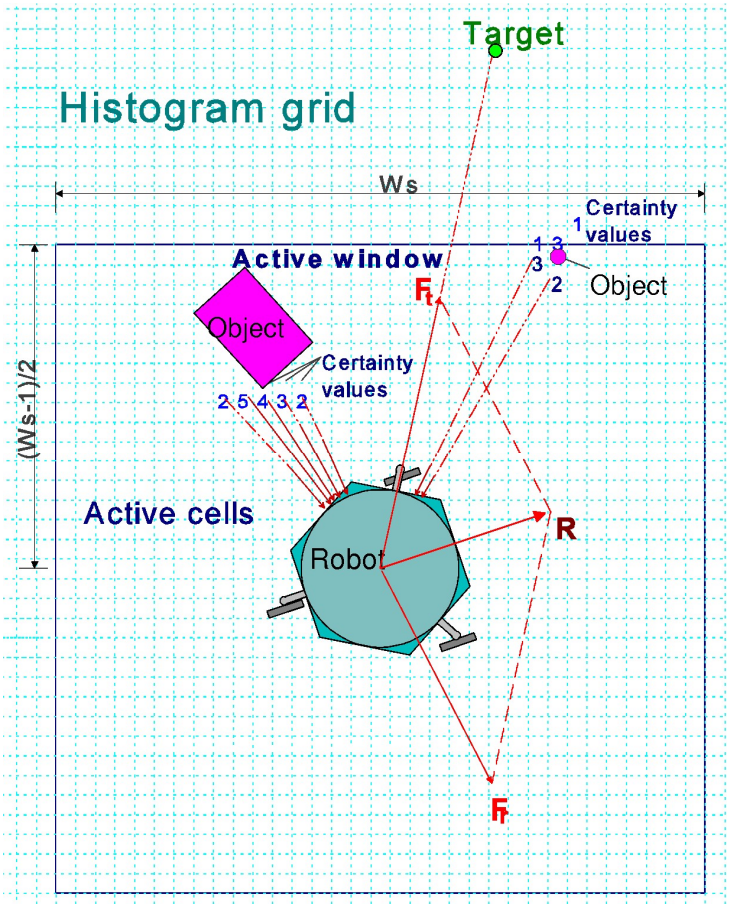

Fig. 17

Anyway this algorithm has some problems:

- sometimes the robot does not pass through a doorway, because the repulsive forces from both sides of the doorway result in a force that pushes the robot away
- the robot's momentary position is mapped in the histogram grid; when the robot moves, drastic changes in the resultant R may be encountered
- when travelling in narrow corridors two situations may occur:
  - the robot travels along the centerline between the walls: the motion is stable

- the robot travels slightly toward one side of the centerline: a strong repulsive force arises from a wall pushing the robot across the centerline, then the same happens with the other wall; the motion is unstable.

3.2.4 Vector-Field-Histogram

Starting from the shortcomings of the VFF method, **Vector-Field-Histogram** arises, taking care of the most problematic key point: whenever the resultant force F_r has to be computed, a bunch of data is reduced in only two information, direction and magnitude of F_r. An excessive data reduction occurs. Consequently, detailed information about the local obstacle distribution is lost.

The VFH method employs a two-stage data reduction technique, rather than the single-step technique used by the VFF method [11]. Thus, three levels of data representation exist:

- The highest level holds the detailed description of the robot's environment. In this level, the two-dimensional Cartesian histogram grid C is continuously updated in real-time with range data sampled by the on-board range sensors. This process is identical to the one described for the VFF method.
- At the intermediate level, a one-dimensional polar histogram H is constructed around the robot's momentary location. A transformation maps the active region C* onto H.
- The lowest level of data representation is the output of the VFH algorithm: it defines the reference values for the drive and steer controllers of the vehicle.

The algorithm selects the most suitable sector from among all polar histogram sectors with a low polar obstacle density, and the steering of the robot is aligned with that direction.

Usually there are two or more candidate valleys (areas of safe travel for the vehicle) and the VFH algorithm selects the one that most closely matches the direction to the target k_{targ} . Once a valley is selected, it is further necessary to choose a suitable sector within that valley. At first, the algorithm measures the size of the selected valley. Here, two types of valleys are distinguished, namely, wide and narrow ones. The sector that is nearest to $k_{targ}$is denoted k_n and represents the near border of the valley. The far border is denoted as k_f. The desired **steering directio**n q is then defined as $q = \frac{k_n + k_f}{2}$, both for wide and narrow valleys, so to maintain a course centered between obstacles.

The robot's ability to pass through narrow passages and doorways results directly from this, from the ability to identify a narrow valley and to choose a centered path through that valley. The VFF method lacks this property.

Also this method achieves the **elimination of the fluctuations** in the steering control: with the averaging effect of the polar histogram, k_n and k_f (and consequently q) vary only mildly between sampling intervals. Thus, the VFH method does not require a low-pass filter in the steering control loop and is therefore able to react much faster to unexpected obstacles.

The robot's maximum speed V_{max} can be set at the beginning of a run. The robot tries to maintain this speed during the run unless forced by the VFH algorithm to a lower instantaneous speed V, which is determined in each sampling interval.
The smoothed polar obstacle density in the current direction of travel is computed. If it is greater than 0 it indicates that an obstacle lies ahead of the robot. Large values of density mean that a large obstacle lies ahead or an obstacle is very close to the robot. Either case is likely to require a drastic change in direction, and a reduction in speed is necessary to allow the steering wheels to turn into the new direction.

3.2.5 Timed Elastic Band

For simple Elastic Band algorithm, the path to be followed is treated as an elastic band that is able to change its shape. Forces acting on the elastic band are computed by taking the gradient of the potential energy at discrete path points [12].

The repulsive forces on the elastic band are produced by obstacles in the vicinity of the path. In total, three types of forces are acting on the elastic band, as shown in Fig. 18.
The internal force f_{int}has to model the spring force; the external force $f_{ext}$is due to obstacles. Also, to constrain the motion along the elastic band, a constraint force $f_{constr}$is introduced with a direction tangential to the elastic band.
The path is represented by particles that are subjected to the total force:

$$f_{tot} = f_{int} + f_{ext} + f_{constr}.$$

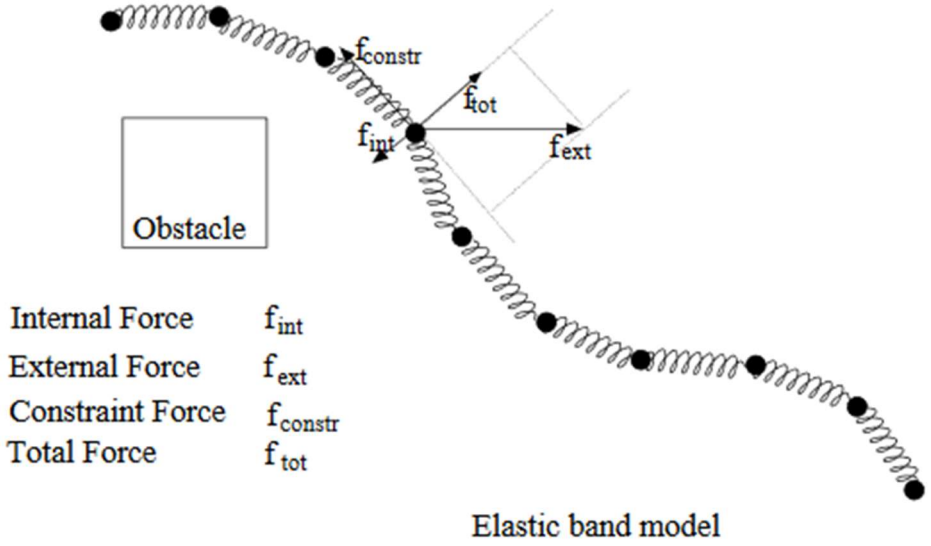

Fig. 18: Representation of an elastic band as a series of particles with springs in between.

This algorithm has been adapted to work with car-like robots and its name has become Timed Elastic Band (TEB) because time becomes part of the objective function.
To be noticed that as "configuration" of the car it is intended **s** = $[\, x, y, \beta \,]$ where x,y are the 2D coordinates of the car on a map of the environment and βrepresents the yaw angle of the car, angle with respect to the z-axis.

The overall objective of the TEB algorithm is to steer the car-like robot from a start s_s to a desired goal configuration s_f in minimum time [13].
The underlying optimization problem has as result a vector composed of the discretized sequence of n robot configurations $(\, s_k \,)_{k=1,2,\dots,n}$where k stand for discretized instants of time.
The TEB approach incorporates temporal information directly into the optimization problem and thus accounts for the minimization of transition time under kinodynamic constraints. Let $(\Delta T_k \,)_{k=1,2,\dots,n-1}$ denote a sequence of strictly positive time intervals ΔT_k□R+. EachΔT_kdescribes the time required to transit from $s_k$tos_{k+1}.
The TEB optimization problem is formulated as a nonlinear program (NLP):

$$\min \sum_{k=1}^{n-1} \Delta T_k{}^2$$

and it is subject to the following constraints:

- $s_1 = s_s$ and $s_n = s_f$
- $0 \leq \Delta T_k \leq \Delta T_{max}$
- $h_k(s_{k+1}, s_k) = 0$: constraint that has to satisfy the kinematics equation of the car
- $r_k(s_{k+1}, s_k) \geq 0$: constraint that impose a minimum turning radius
- Constraints to limit velocity and acceleration:

  $$v_k(s_{k+1}, s_k, \Delta T_k) = [v_{max} - |v_k|, \omega_{max} - |\omega_k|] \geq 0$$
  $$\alpha_k(s_{k+2}, s_{k+1}, s_k, \Delta T_{k+1}, \Delta T_k) = a_{max} - |a_k| \geq 0$$

  Special cases occur at k=1 and k=n−1 for which $v_1$and$v_{n-1}$are substituted by the desired start and final velocities (v_s, ω_s) and (v_f, ω_f) respectively.
- Constraint to optimize clearance from obstacles: the distance between configuration s_k and the obstacle perimeter is quantified in a continuous metric space, e.g. the Euclidean metric. Let $\delta(s_k, O)$describe the minimal Euclidean distance between obstacle O and pose s_k. A minimum separation δ_{min} between all obstacles and configuration s_k is defined by the inequality constraint:

  $$o_k(s_k) = [\delta(s_k, O_1), \delta(s_k, O_2), \dots, \delta(s_k, O_R)] - [\delta_{min}, \delta_{min}, \dots, \delta_{min}] \geq 0$$

  with R equal to the number of obstacles in the surroundings.

Solving nonlinear programs with hard constraints is computationally expensive. Therefore, the exact nonlinear program (NLP) is transformed into an approximate nonlinear squared optimization problem in which constraints are incorporated into the objective function as additional penalty terms with weights for each term.
In theory, the solution of this last problem coincides with the actual minimizer of the nonlinear program (NLP) in case all weights tend towards infinity. Unfortunately, large weights introduce ill-conditioning of the problem such that the underlying solver does not converge properly due to inadequate step sizes. The TEB approach abandons the true minimizer in

favor of a suboptimal but computationally more efficiently obtainable solution with user defined weights.
Further experiments reveals that unit weights of 1 provide a reasonable point of departure, except for the weight associated with the non-holonomic kinematics which is a few orders of magnitude larger (≈1000).

The TEB approach defines a predictive control strategy in order to account for disturbances, map and model uncertainties and dynamic environments that guides the robot from its current pose towards a goal pose s_f: to achieve this, problem (NLP) is solved repeatedly at a rate faster than the robot control cycle rate.

3.3 Path planning

The algorithms described before are examples of a grid-based approach to motion planning, that overlay a grid on configuration space, and assume each configuration is identified with a grid point. At each grid point, the robot is allowed to move to adjacent grid points as long as the line between them is completely contained within the free space C_{free} (tested with collision detection). This discretizes the set of actions, and search algorithms (or **pathfinding algorithms**) are used to find a path from the start to the goal.

Pathfinding or **pathing** is an exploring algorithm that has as purpose to find the shortest route between two points, and in the meanwhile applying also other optimization criteria (cheapest path, fastest path, etc).
At its core, a pathfinding method is an exploring method in the sense that it searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached.

The major part of the pathfinding algorithms address the problem by exhausting all possibilities in the graph. These algorithms run in $O(|V| + |E|)$, or linear time, where V is the number of vertices, configurations the vehicle can assume, and E is the number of edges between vertices.
The more complicated problem is finding the optimal path. The exhaustive approach in this case is known as the Bellman–Ford algorithm, which yields a time complexity of $O(|V| \cdot |E|)$, or quadratic time.
It is not necessary though to examine all possible paths to find the optimal one. Algorithms such as A* and Dijkstra's algorithm strategically eliminate paths, either through heuristics or through dynamic programming. By eliminating impossible paths, these algorithms can achieve time complexities as low as $O(|E| \cdot log(|V|))$.

Here there are a few important pathfinding algorithms.

3.3.1 A*

A* is an informed search algorithm, or a best-first search: it is formulated in terms of weighted graphs in a way that it considers to have the starting position as a node of the graph, from there it has to find a path to a defined goal, always a node, having the smallest cost (least distance travelled, shortest time, etc.). At the beginning it initializes a tree of paths from the start node to the destination, then extends the tree until the termination criterion is satisfied, Fig. 19.
At each iteration of its main loop, A* needs to determine which of its paths to extend.
It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal. Specifically, A* selects the path that minimizes:

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal.

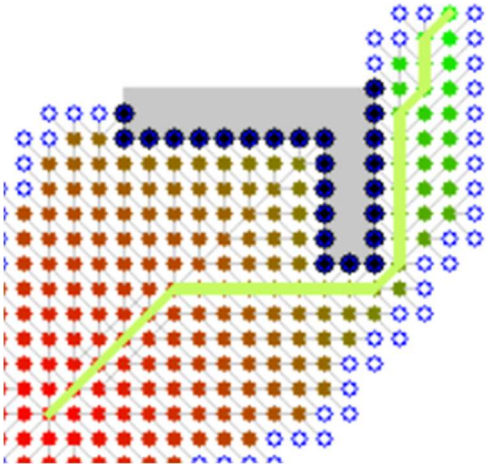

Fig. 19

A* uses a **priority queue** to select which node to expand that has the minimum cost. At each step, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue.
The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an **admissible** heuristic[5].
The algorithm described so far gives as output only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily modified so that each node on the path keeps track of its predecessor.

3.3.2 D* (Dynamic A*)

This algorithm, when it observes new map information (such as previously unknown obstacles), adds the information to its map and, if necessary, replans a new shortest path from its current coordinates to the given goal coordinates; the process continues until the goal is reached or it is determined that it is not reachable.
The basic operation of D* is outlined below. D* maintains a list of nodes to be evaluated, known as the "**OPEN list**".
Nodes are marked as having one of several states:

- NEW, meaning it has never been placed on the OPEN list
- OPEN, meaning it is currently on the OPEN list
- CLOSED, meaning it is no longer on the OPEN list
- RAISE, indicating its cost is higher than the last time it was on the OPEN list
- LOWER, indicating its cost is lower than the last time it was on the OPEN list

[5] A heuristic function is said to be **admissible** if it never overestimates the cost of reaching the goal, i.e. the cost it estimates to reach the goal is not higher than the lowest possible cost from the current point in the path.

The algorithm works by iteratively selecting a node from the OPEN list and evaluating it and then it propagates the node's changes to all of the neighboring nodes placing them on the OPEN list. This propagation process is termed "**expansion**".
A* follows the path from start to finish; D* instead begins by searching backwards from the goal node. Backpointers are introduced to make each node knowing which is the next one that leads to the target. When the start node is the next node to be expanded, the algorithm is done, and the path to the goal can be found by simply following the backpointers.

When an obstruction is detected along the intended path, all the points that are affected are marked as **RAISED** and again placed on the OPEN list. Then the RAISE state is propagated to all of the nodes' descendants, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the **LOWER** state to its neighbors. These waves of RAISE and LOWER states are the heart of D*.
The algorithm works only on the points which are affected by change of cost.

This algorithm has been used for Mars Rover prototypes and tactical mobile robot prototypes for urban reconnaissance.

3.3.3 RRT (Rapidly-exploring random tree)

RRT constructs a tree using random sampling in search space. The tree starts from an initial configuration, z_{init}, and expands to find a path towards a goal state say z_{goal}. The tree gradually expands as the iteration continue, see Fig. 20.
During each iteration, a random state z_{rand} is selected from configuration space Z and, if it lies in a free obstacle area, a nearest node $z_{nearest}$ is searched in tree according to a defined metric ρ. If $z_{rand}$is accessible to $z_{nearest}$, then the tree is expanded by connecting z_{rand} and $z_{nearest}$. Otherwise, it returns a new node z_{new} by using a steering function, thus expands tree by connecting z_{new} with $z_{nearest}$.

When an initial path is found, the process does not stop until a certain amount of time expires or a certain number of executions has been reached.
While RRTs are sometimes able to solve very hard problems quickly, their main drawback is that no guarantees are made on solution quality and, in practice, their solutions are often poor.

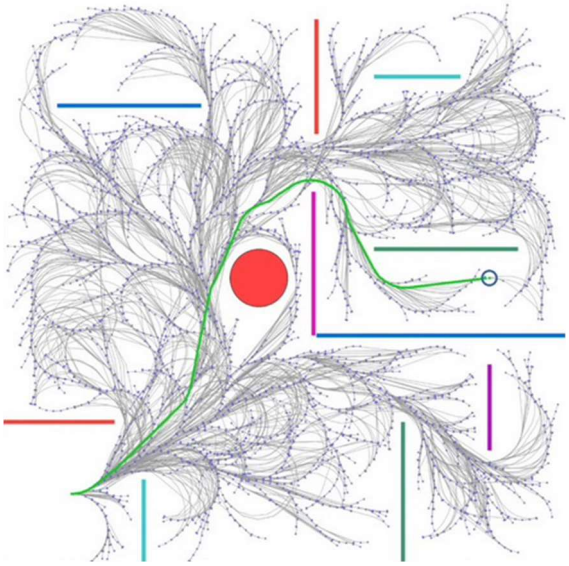

Fig. 20: the red dot is an obstacle, such also the colored rectangles; in green the path that has been evaluated to reach the destination on the right.

3.3.4 RTR* (Real time R*)

A planning problem P is defined as a tuple $\{S, s_{start}, G, A, \alpha, O, D\}$ where

- S is the set of states, where a state is the tuple $< x, y, \theta, v, t >$ corresponding to location, heading, speed, and the current time
- s_{start} is the starting state
- G is the set of goal states, where each state g is underspecified as $< x, y, \theta, v >$ because it is unknown when the robot will be able to arrive there
- A is the set of motion primitives available to the robot, that is a set of primitive actions that can be applied in a certain state.
- The function α is a primitive action that makes possible to pass from a state to another and has a duration of t_a
- O is the set of static obstacles whose locations are known and do not change
- D is the set of dynamic obstacles. This set can change as the robot acquires more observations of an obstacle.

When expanding a node s, the R* algorithm selects a random set of k states that are within some distance Δ of s. These states will form a sparse graph Γ that is searched for a solution. The edges computed between nodes in Γ represent actual paths in the underlying state space. If the algorithm does not find a solution within a given node expansion limit, it gives up, labelling the node as AVOID, and allowing R* to focus the search elsewhere. R* will only return to these hard to solve subproblems if there are no non-AVOID labelled nodes left.
R* solves the planning problem by carrying out searches that are much smaller than the original problem, and easier to solve. Note that R* finds complete paths to the goal on every search, and the time that this takes is not bounded [14].

To make it real time:

- RTR* terminates when the expansion limit has been reached, not when the goal is reached, this because it isn't sufficient to just reach the goal state. In domains with moving obstacles, it may be necessary to move off the goal state at some future time to get out of the way of an obstacle.
- In R*, if a node is labelled as AVOID, it is inserted back onto the open list. If the node is ever popped off of the open list again, another attempt is made at computing the path, this time with no node expansion limit. In RTR*, each time a search fails due to the expansion limit, the limit is doubled for that node the next time it is removed from the open list. Since the expansion limit for computing the path to a node is doubled each time, the total amount of extra searching that may need to be done is bounded by a constant factor in the worst case.
- RTR* has in memory information about the nodes in the sparse graph that are on the best path found. This allows the search to seed the sparse graph Γ with nodes that appeared promising on the previous iteration, a kind of path reuse. All other nodes in Γ not on the most promising path are discarded. If the costs of the graph have not changed much, then these nodes will most likely still be favorable and will be used by RTR*. If costs have changed, then RTR* recomputes a better path, possibly not even using the cached sparse nodes.
- One of the key insights of the R* algorithm is that dividing the original problem up into many smaller subproblems makes it generally easier to solve than solving the original. To make these problems easier, the goal condition can be relaxed to allow states within a certain radius of the actual goal state and with any heading and any speed to be considered a goal.

3.3.5 PLRTA* (Partitioned Learning Real-time A*)

PLRTA* partitions g and h values into static and dynamic portions. The static portion refers to only those things that are not time dependent. The dynamic portion refers to only those things that are dependent on time, such as the locations of the dynamic obstacles.

For each search node encountered, the static costs (g_s), dynamic costs (g_d), static cost-to-go (h_s) and dynamic cost-to-go (h_d) can be tracked.
The evaluation function is now:

$$f(n) = g_s(n) + g_d(n) + h_s(n) + h_d(n).$$

Partitioned Learning Real-time A* (PLRTA*) performs a limited lookahead A* search forward from the agent. In backtracking algorithms, **lookahead** is the generic term for a subprocedure that attempts to foresee the effects of choosing a node with respect to another. It then selects the minimum f node in the open list and labels it g_0. The planning iteration ends by taking the first action along the path from s_{start} to g_0. The algorithm goes on taking actions between nodes until the goal is reached.

Heuristic learning can be divided into separate steps for h_s and h_d. The main difference between the two learning steps is in the setup phase. The h_s of all nodes in the closed list can be set to ∞. h_s is then computed from the h_s of a successor state and the static cost g_s incurred by moving from its state to its successor.

For what regards the dynamic cost, it can be set h_d= 0. While this is very weak, it can be improved drastically during the search. Because dynamic and static costs are considered separately, h_d values can be learnt through g_d costs. These h_d values of a state can frequently change with respect to time, depending on the movements of dynamic obstacles and their predicted future locations.

The learning step of PLRTA* will always only raise the heuristic estimate of a state. However, there must be a way to lower a high cached heuristic value if the dynamic obstacle that caused the high value moves away. To accomplish this, the cached dynamic heuristic values of all states can decay over time linearly. This allows the algorithm to “unlearn” dynamic heuristic values that turned out to be overestimates, in fact after a while the value of the dynamic cost is back to zero and it is removed from the cache [14].

3.4 Costmap

There are a few other approaches to motion planning but it has been decided to use the grid-based one to have the chance of using a costmap.
A **costmap** is a grid in which every cell gets assigned value (cost) depending on the fact that an obstacle is present or not, so giving the chance of determining distance to objects.

Note: In Fig. 21, the red cells represent obstacles in the costmap, the blue cells represent obstacles within a certain radius from the robot, and the red polygon represents the footprint of the robot. For the robot to avoid collision, the footprint should never intersect a red cell and the center point of the robot should never cross a blue cell.

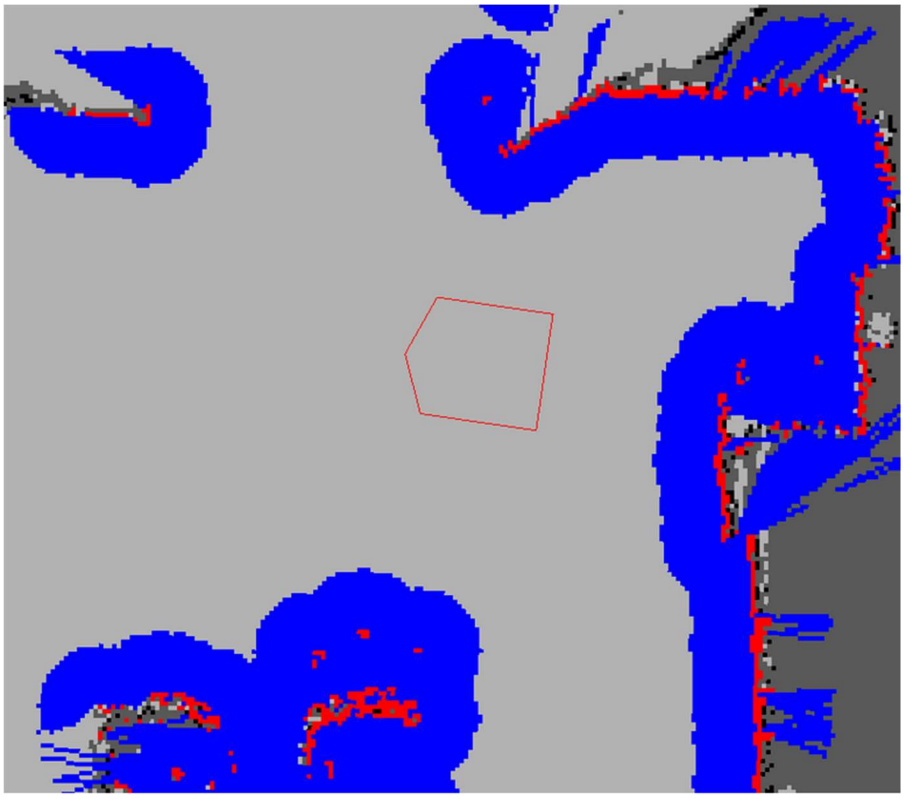

Fig. 21

The blue cells are the result of the inflation. **Inflation** is the process of propagating cost values out from occupied cells that decrease with distance out to a user-specified inflation radius [15].
For this purpose, 5 specific cost values can be defined as they relate to a robot.

- "Lethal" cost means that there is an actual obstacle in a cell.
- "Inscribed" cost means that a cell is less than the robot's inscribed radius away from an actual obstacle. A crash will verify if the robot center passes in a cell that is at or above the inscribed cost.
- "Possibly circumscribed" cost is similar to "inscribed", but using the robot's circumscribed radius as cutoff distance. Thus, if the robot center lies in a cell at or above this value, then it depends on the orientation of the robot whether it collides with an obstacle or not.

- "Freespace" cost is assumed to be zero, and it means that there is nothing that should keep the robot from going there.
- "Unknown" cost means there is no information about a given cell.
- All other costs are assigned a value between "Freespace" and "Possibly circumscribed" depending on their distance from a "Lethal" cell and the decay function provided by the user.

Better explanation is provided in Fig. 22.

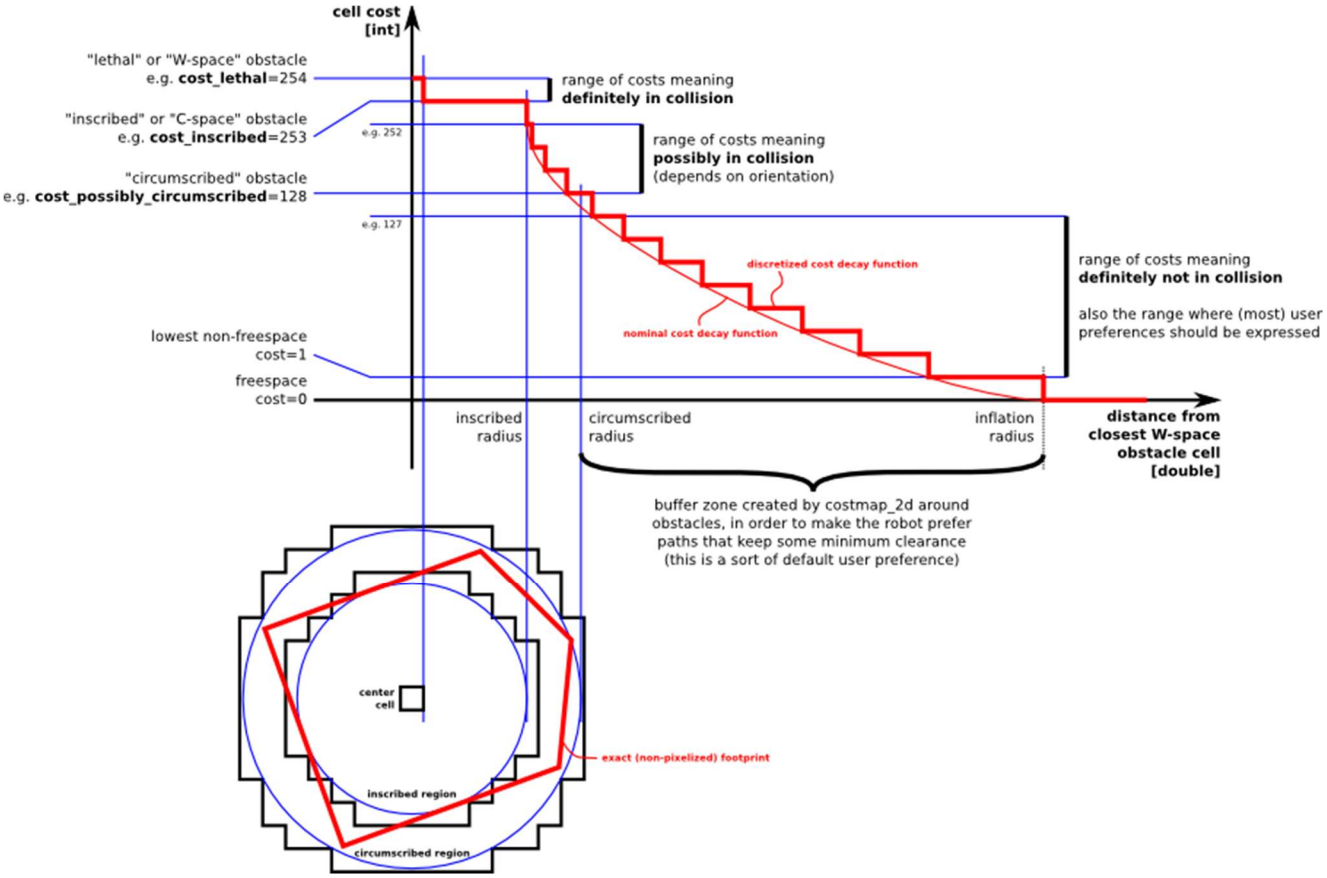

Fig. 22

To sum up, then, why the costmap has been used:

- path planning algorithms are the basis for motion planners because they evaluate a path that the vehicle should follow;
- to evaluate this path the costmap is useful because it takes into consideration the obstacles surrounding the vehicle, so the planner knows which configurations are not in the free space C_{free};
- once a path, is defined the motion planner can make a decision about the speed that has to be pursued by the vehicle, taking into account that a certain maximum speed should not be exceeded.

3.5 Trajectory prediction

The Motion Planning algorithms are commonly based on A* path planning algorithm. This one works in an environment in which no dynamic obstacles are present. This means that, in a real environment, which also includes dynamic obstacles, the path that it evaluates should not be undertaken.

To take that into consideration, it has been decided to mark new information in the costmap consisting in the **trajectory that the obstacles are about to take**. In this way a possible crash can be predicted and a different route can be defined.

Vehicle tracking is one of the most important data fusion tasks for Intelligent Transportation Systems (ITS). Especially for advanced driver assistance systems such as Collision Avoidance, Adaptive Cruise Control (ACC), Stop-and-Go-Assistant, a reliable estimation of other vehicles' positions is one of the most critical requirements.

3.5.1 Motion models

In order to increase the stability and accuracy of the estimation, the vehicles are mostly assumed to comply with certain motion models which describe their dynamic behavior.
From the data fusion point of view, the task is to estimate the parameters of the model – taking into account all available observations.

A few motion models exist that have different levels of complexity. *Linear motion models* assume a constant velocity (CV) or a constant acceleration (CA). Their linearity of the state transition allows a good propagation of the state probability distribution.
On the other hand, these models assume straight motions and are thus not able to take rotations (especially the yaw rate) into account.
Curvilinear models can be further divided by the state variables which are assumed to be constant. The simplest model of this level is the Constant Turn Rate and Velocity (CTRV) model. By defining the derivative of the velocity as the constant variable, the Constant Turn Rate and Acceleration (CTRA) model can be derived.

Both CTRV and CTRA assume that there is no correlation between the velocity v and the yaw rate ω. As a consequence, disturbed yaw rate measurements can change the yaw angle of the vehicle even if it is not moving. In order to avoid this problem, the correlation between v and ω can be modelled by using the steering angle Φ[6] as constant variable and derive the yaw rate from v and Φ. The resulting model is called Constant Steering Angle and Velocity(CSAV). Again, the velocity can be assumed to change linearly, which leads to the Constant Curvature and Acceleration(CCA) model.

[6] This angle is defined between the axis of motion and the direction of the front wheels.

The connections between all models described so far are illustrated in Fig. 23:

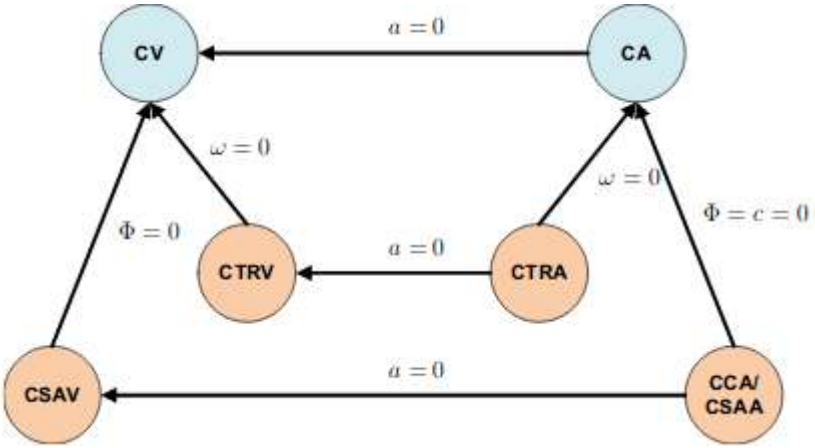

Fig. 23

From a geometrical point of view, nearly all curvilinear models are assuming that the vehicle is moving on a circular trajectory (either with a constant velocity or acceleration).

3.5.1.1 CV (Constant Velocity)

State space of CV model:

$$\vec{x}(t) = (x \quad y \quad v_x \quad v_y)^T$$

It is a linear motion model:

$$\vec{x}(t+T) = A(t+T)\vec{x}(t)$$

It can be rewritten as the state transition function vector:

$$\vec{x}(t+T) = \begin{pmatrix} x(t) + Tv_x \\ y(t) + Tv_y \\ v_x \\ v_y \end{pmatrix}$$

3.5.1.2 CA (Constant Acceleration)

The state space is:

$$\vec{x}(t) = \begin{pmatrix} x & y & v_x & v_y & a_x & a_y \end{pmatrix}^T$$

It is a linear motion model; the linear state transition is as before, taking into considerations also the acceleration.

$$\vec{x}(t+T) = \begin{pmatrix} x(t) + Tv_x(t) + \frac{1}{2}T^2a_x \\ y(t) + Tv_y(t) + \frac{1}{2}T^2a_y \\ v_x(t) + Ta_x \\ v_y(t) + Ta_y \\ a_x \\ a_y \end{pmatrix}$$

3.5.1.3 CTRV (Constant Turn Rate and Velocity)

The state space:

$$\vec{x}(t) = \begin{pmatrix} x & y & \theta & v & w \end{pmatrix}^T$$

can be transformed by the non-linear state transition:

$$\vec{x}(t+T) = \begin{pmatrix} \frac{v}{\omega}\sin(\omega T + \theta) - \frac{v}{\omega}\sin(\theta) + x(t) \\ -\frac{v}{\omega}\cos(\omega T + \theta) + \frac{v}{\omega}\sin(\theta) + y(t) \\ \omega T + \theta \\ v \\ \omega \end{pmatrix}$$

3.5.1.4 CTRA (Constant Turn Rate and Acceleration)

The state space of this models expands the last one by a:

$$\vec{x}(t) = \begin{pmatrix} x & y & \theta & v & a & w \end{pmatrix}^T$$

The state transition equation for this model is:

$$\vec{x}(t+T) = \begin{pmatrix} x(t+T) \\ y(t+T) \\ \theta(t+T) \\ v(t+T) \\ a \\ \omega \end{pmatrix} = \vec{x}(t) + \begin{pmatrix} \Delta x(T) \\ \Delta y(T) \\ \omega T \\ aT \\ 0 \\ 0 \end{pmatrix}$$

with $\Delta x(T)$ and $\Delta y(T)$ that are functions of all the other states.

3.5.1.5 Comparison between the models

As it is, applications of these models in literature show that the sophisticated models CTRV and CTRA provide a better performance than the simple CV model in almost every case. Furthermore, the incorporation of the acceleration additionally improves the overall tracking performance.

The CV and CTRV models have been compared using scenes with a high curvature only. The CV model generates large position errors due to high curvature, whereas the CTRV model is able to provide a better estimation.

In high acceleration situations, the CTRA model performs much better than the CTRV model [16].

3.5.2 Filtering algorithms

These models apply to specific *filtering algorithms*.

3.5.2.1 Kalman filter

The algorithm is composed of two steps:

- the prediction step: the Kalman filter estimates the current state variables with some uncertainty; it makes use of the system's dynamic model, known control inputs and measurements from sensors
- the stabilization phase: once the outcome of the next measurement is obtained (result that has to be corrupted with some error, including random noise) the estimates are updated using a weighted average, with more weight being given to estimates with higher certainty; higher certainty means smaller uncertainty, and these are the values that are considered "trusted" more. Weights are evaluated from the *covariance*, a measure of the estimated uncertainty of the prediction of the system's state.

This process is repeated at every time step, with the new estimate and its covariance informing the prediction used in the following iteration. This means that the Kalman filter works recursively and requires only the last "best guess", rather than the entire history, of a system's state to calculate a new state.

Using a Kalman filter does not assume that the errors are Gaussian. However, if all noise is Gaussian, the Kalman filter minimizes the mean square error of the estimated parameters.

The Kalman filter maintains and updates at every cycle the estimates of the state:
$\hat{x}(k|k)$- estimate of x(k) given measurements at time k, k-1, …
$\hat{x}(k+1|k)$- estimate of x(k+1) given measurements at time k, k-1,...

and the error covariance matrix of the state estimate:
$P(k|k)$- covariance matrix of x(k) given measurements at time k, k-1, …
$P(k+1|k)$- covariance matrix of x(k+1) given measurements at time k,k-1,...

3.5.2.2 Extended Kalman Filter

The **extended Kalman filter (EKF)** is the nonlinear version of the Kalman filter which linearizes about an estimate of the current mean and covariance. The state transition and observation models need not be linear functions of the state but may instead be nonlinear functions.

Unlike its linear counterpart, the extended Kalman filter in general is *not* an optimal estimator (of course it is optimal if the measurement and the state transition model are both linear, as in that case the extended Kalman filter is identical to the regular one). In addition, if the initial

estimate of the state is wrong, or if the process is modelled incorrectly, the filter may quickly diverge.
Another problem with the extended Kalman filter is that the estimated covariance matrix tends to underestimate the true covariance matrix and therefore risks becoming inconsistent in the statistical sense without the addition of "stabilizing noise".

Having stated this, the extended Kalman filter can give reasonable performance, and is arguably the *de facto* standard in navigation systems and GPS.

4. Thesis contribution

4.1 Costmap

The costmap_2d package of ROS has been used and integrated.

The implementation of this thesis shows the vehicle moving through the map using two types of navigation: global and local.

- The global navigation is used to create paths for a goal in the map or a far-off distance. At this purpose, a **global costmap** is used, this is an entire map of the environment created during the movement of the car
- The local navigation is used to create paths in the nearby distances and avoid obstacles. At this purpose, a **local costmap** is used for the local navigation, it considers only a little window with the car at its center which draws only the obstacles that pass through that.

In both cases, global or local, the costmap is divided in layers.
Each layer is instantiated in the Costmap2DROS using pluginlib and is added to the LayeredCostmap [5]. In general there are:

- **Static Map Layer:** represents a largely unchanging portion of the costmap, like those generated by SLAM.
- **Obstacle Map Layer:** tracks the obstacles as read by the sensor data.
- **Inflation Layer:** is an optimization that adds new values around lethal obstacles (i.e. inflates the obstacles) in order to make the costmap represent the configuration space of the robot.
- **Other Layers:** can be implemented and used in the costmap via pluginlib.

The structure of the layers is described in Fig. 24.

The first approach to *safe* motion planning is to extend the obstacle layer.

The actual implementation of this layer only takes into account the actual position of the obstacles: everything is seen as static, also moving objects are drawn in the costmap in the actual position they have in the moment data is acquired by the sensors.

This behaviour can cause the ego-vehicle to recognize the presence of another one only when they are approaching each other, leaving not so much time to react.

The solution to this is to plot on the costmap also the trajectory of the obstacles, so that the motion planner can take into account also future positions of objects while building the path to the goal and in this way predict a crash and have time to find an alternative route solution.

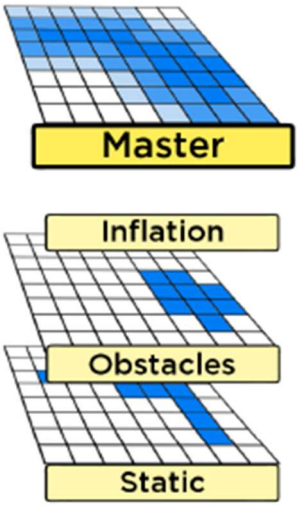

Fig. 24: all the layers that compose the costmap; the master will be a composition of the underlying layers

4.1.1 First step: costmap only with actual position of objects

The first step was to make the already existing layer work, so to plot the actual position of the cars that the Ibeo sensors have discovered.

This was possible because the object detection from Ibeo sensors publishes in a topic the information about the position of the center of gravity of the PointClouds it recognizes.

In Fig. 25, that is an rviz capture (rviz is the tool that is used to plot the costmap), it can be seen:

- colored dots, that are the points that reflect the laser ray sent by the sensor
- colored rectangles, that are the result of the object detection the platform does to classify the objects and compute their width and length
- black rectangles, that are the result of marking that cells to LETHAL in the costmap, so the result of the obstacle layer
- at the center, the footprint of the ego-vehicle.

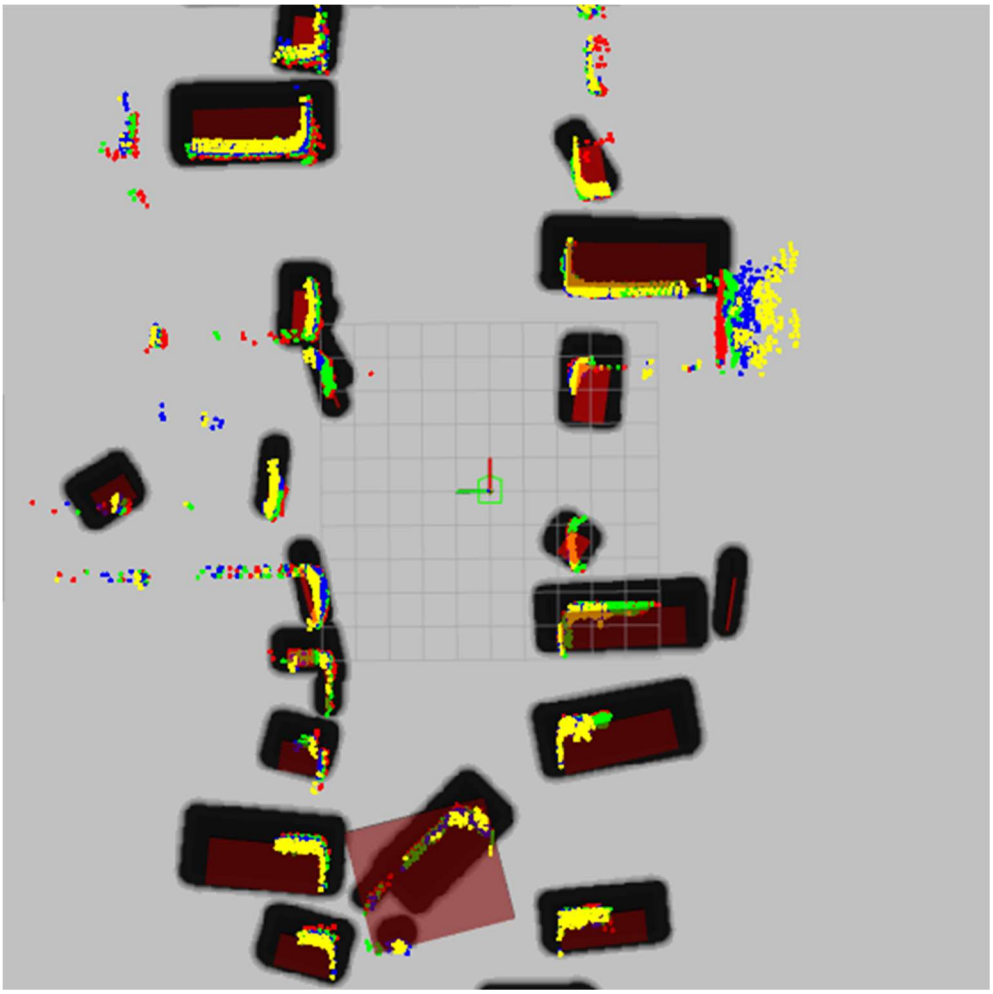

Fig. 25

4.1.2 How to recognize *not moving* objects?

The next step was to evaluate the trajectory and to plot it.

To begin in an easy way, it is decided to start from plotting a **segment** in the direction of motion of the obstacle, following an initial assumption that the vehicles are all going straight forward. This is not so easy as could seem because Ibeo only had the information about the **relative velocity of the obstacles** with respect to the ego-vehicle.

This means that if a car is not moving, it can be seen as moving in the opposite direction with respect to the one of the vehicle, and this is a problem for the next steps.

In this situation, geometry and the tf package of ROS come to help.

The goal is to understand if a vehicle is not moving, so if its position in the reference frame of the Earth is always the same.

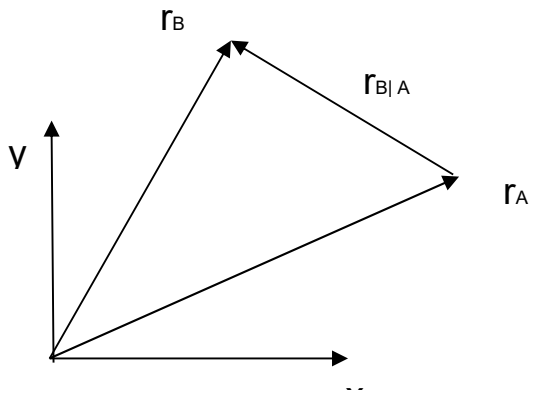

Fig. 26

In Fig. 26, if A is the ego-vehicle and B is the obstacle, let's say that $r_B$is the position of the obstacle and $r_A$is the position of the ego-vehicle with respect to a fixed reference frame, the Earth. The ibeo sensors only gives the relative position $r_{B\,|\,A}$of the obstacle B with respect to A. If the goal is the position of the obstacle with respect to the fixed frame the steps are :

$$r_{B\,|\,A} = r_B - r_A$$

so

$$r_B = r_{B\,|\,A} + r_A \ .$$

If the resulting position r_B results to be always the same (not exactly the same because of the sensor errors, but very near to the same position) it can be recognized that the object is standing still and so that it will not be a danger for the movement of the ego-vehicle.

Evaluating r_B is what tf can help to do. There are two reference frames, the one that runs with the vehicle, and the Earth-one. For what regards the first one, the origin of the frame is simply the center of gravity of the ego-vehicle. For what regards the second one, the origin is the pose the vehicle has at the beginning, when the simulation starts. Then $r_A$is computed depending on how far the ego-vehicle travels from that point on.
The position $r_{B\,|\,A}$ is given by the ibeo sensors and it is referred to the car reference frame. Then tf is capable to convert it in the fixed-one (r_B).

But the car has also an orientation with respect to the z-axis. This orientation is different if considered in the car reference frame or in the fixed reference frame so it is needed also to convert this one. It has been necessary to express the orientation angle, that basically is a yaw angle, as a quaternion[7] so to use the set of functions provided by the package tf2/LinearMath/Quaternion.
This package includes a series of functions to do the change of representation from Roll-Pitch-Yaw angles to quaternions and vice versa.

[7] In mathematics, the **quaternions** are a number system that extends the complex numbers. q = a +b ***i*** + c ***j*** + d ***k***, where *a*, *b*, *c*, *d*, are real numbers, and ***i, j, k***, are symbols that can be interpreted as unit-vectors pointing along the three spatial axes.

It is considered only a yaw angle because in this simple implementation only movements around the z-axis are considered, so the ego-vehicle is moving in a flat environment. A future work could be to introduce movements around axis x and y.

The result is that it is possible to evaluate if an obstacle is moving or not but still considering that all obstacles are moving straight is very weak.
So it is time to improve the algorithm.

4.1.3 Second step: costmap with trajectory prediction of objects

The next chapter will discuss about all the trajectory prediction algorithms that have been studied and furthermore the results of the best one with the motion planning algorithm.
What can be said here are the steps that has been done to mark the trajectory on the costmap, so it can be supposed that the trajectory has been already evaluated at this point.

4.1.3.1 Only the points of the predicted trajectory

The first approach is to take all the points that are predicted by the algorithm and then set on the costmap a higher cost for these ones.
Basically what can be seen is that some points in the costmap are highlighted but there is no connection between them, see Fig. 27.

This is acceptable when the moving obstacle has an almost constant speed, in fact all the points are aligned properly but, when the moving obstacle has a little bit of acceleration, the dots start to be very far from each other so much that they seem like stand-alone fixed obstacles. This is not acceptable because the ego-vehicle thinks that it has the right to pass in the middle of the points, thus leading to crashes.

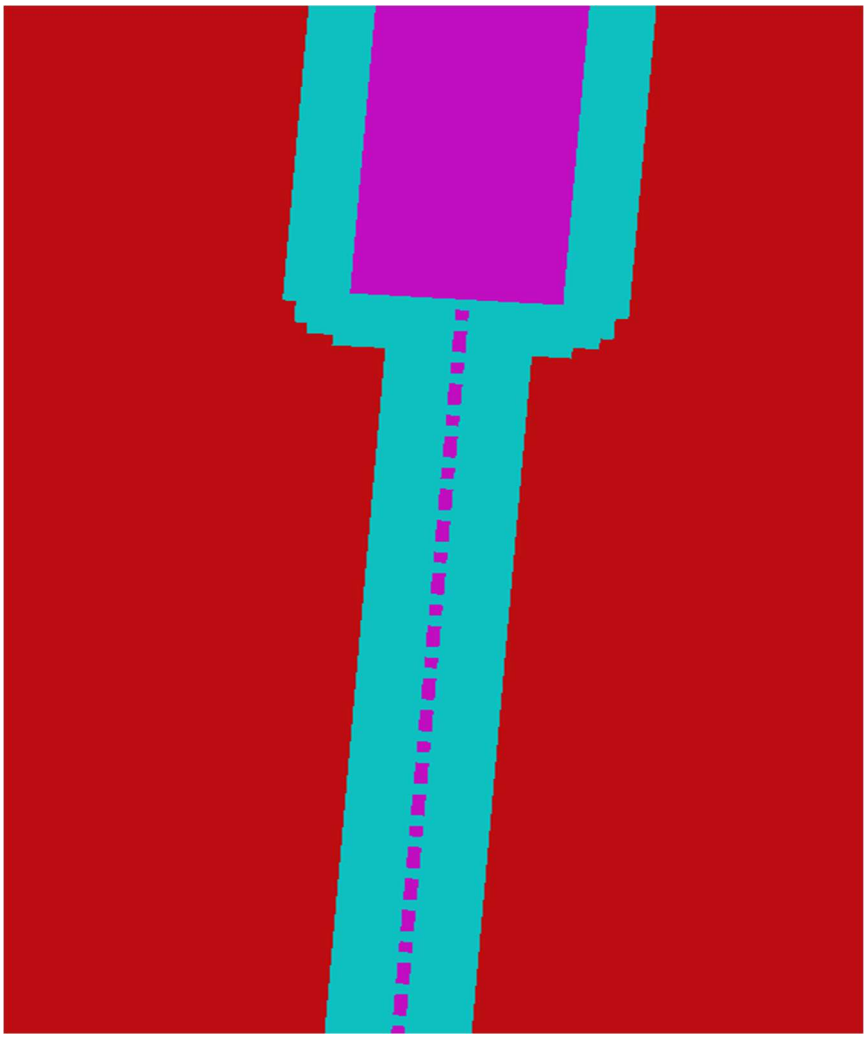

Fig. 27

4.1.3.2 Connecting points

Another solution that is preferable is to draw lines between points so as to obtain a **polygonal chain**[8] at the end.

To do so, a tracing line algorithm is used taking as model the one implemented in the package costmap_2d.

4.1.3.2.1 Breshenham's algorithm

Consider a line with initial point (x_1, y_1) and terminal point (x_2, y_2) in device space.

If $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$, the *driving axis* (DA) is the x-axis if $|\Delta x| \geq |\Delta y|$, and the y-axis if $|\Delta y| \geq |\Delta x|$.

[8] In geometry, a **polygonal chain** is a connected series of line segments. It is a curve consisting in a sequence of points and the line segments between them.

The DA is used as the “axis of control” for the algorithm and is the axis of maximum movement. While the algorithm goes on, the coordinate corresponding to the DA is incremented more than the other one, usually denoted the *passive axis* or PA, that is only incremented as needed.

As an example, it can be considered the case in which the objective is to draw a line from (0,0) to (5,3) in device space, Fig. 28.

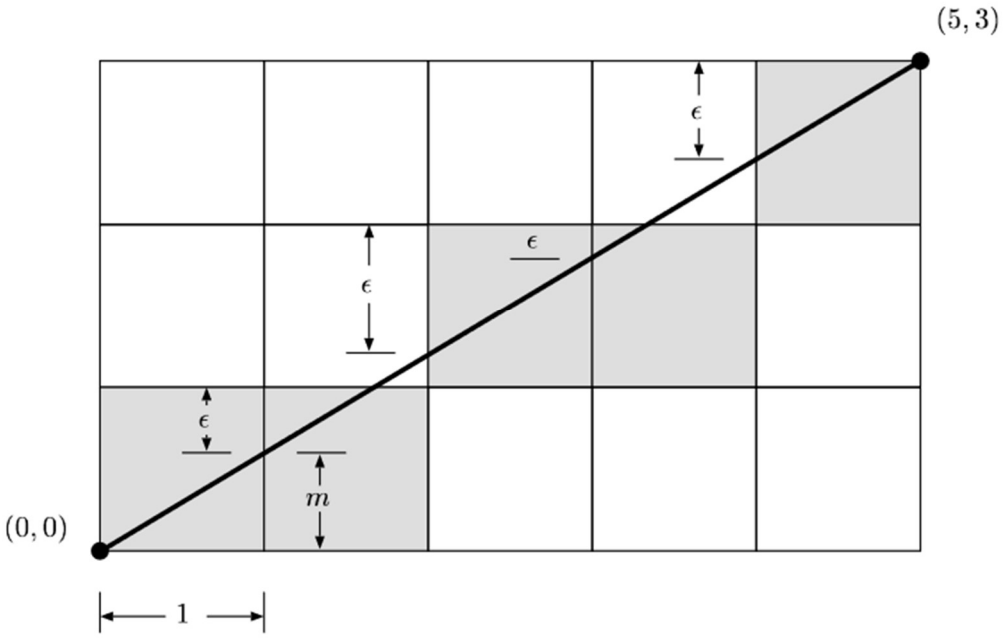

Fig. 28

Breshenham’s algorithm begins with the point (0,0) and “illuminates” that pixel.
The x coordinate is incremented by 1.
Rather than keeping track of the y coordinate (which increases by $m = \frac{\Delta y}{\Delta x}$, each time the x increases by one), the algorithm keeps an error bound ε at each stage, which represents the distance between where the line has exited the pixel and the top edge of that pixel (see Fig. 28). ε is first set to $m - 1$, and is incremented by$m$ each time the x coordinate is incremented by one.
If ε becomes greater than zero, it means that the line has moved upwards one pixel, and that the y coordinate must be incremented and the error must be readjusted to represent the distance from the top of the new pixel – which is done by subtracting 1 from ε.
All the steps are described in Fig. 29.

Till this moment, it has been considered that Δx and Δy are positive. If this is not the case, the algorithm is essentially the same except for the following:

- ε is calculated using $\left| \frac{\Delta y}{\Delta x} \right|$
- x and y are decremented (instead of incremented) by one if the sign of Δx or $\Delta y$is less than zero, respectively.

(x,y)	ε	description
(0,0) (1,0)	-0.4 0.2	pixel(0,0) increment ε by 0.6 increment x by 1
(1,0) (1,1) (2,1)	0.2 -0.8 -0.2	pixel(1,0) since ε is positive: increment y by 1 decrement ε by 1 increment ε by 0.6 increment x by 1
(2,1) (3,1)	-0.2 0.4	pixel(2,1) increment ε by 0.6 increment x by 1
(3,1) (3,2) (4,2)	0.4 -0.6 0.0	pixel(3,1) since ε is positive: increment y by 1 decrement ε by 1 increment ε by 0.6 increment x by 1
(4,2)	0.0	pixel(4,2)

Fig. 29

Breshenham's Algorithm, as discussed above, works with a floating point arithmetic. ε is initialized to:

$$\varepsilon = \frac{\Delta y}{\Delta x} - 1$$

and is incremented by $\frac{\Delta y}{\Delta x}$ at each step.

Since both Δy and Δx are integer quantities, it can be converted to an all integer format by multiplying the operations through by Δx.

The following integer quantity will be considered:

$$\underline{\varepsilon} = \Delta x \cdot \varepsilon = \Delta y - \Delta x$$

and it will be incremented by Δy at each step, and decremented by Δx when it becomes positive.

The steps for the integer version of the algorithm, considering the same example than before, are described in Fig. 30 and the pseudo-code is the following [17]:

Let $\Delta x = x_2 - x_1$
Let $\Delta y = y_2 - y_1$
Let $j = y_1$
Let $\underline{\varepsilon} = \Delta y - \Delta x$

for $i = x_1$ **to** $x_2 - 1$
illuminate (i, j)
if ($\underline{\varepsilon} \geq 0$)
j+= 1
$\underline{\varepsilon}$ −= Δx
end if
i+= 1
$\underline{\varepsilon}$ += Δy
next i
finish

(x,y)	$\underline{\varepsilon}$	description
(0,0) (1,0)	-2 1	pixel(0,0) increment $\underline{\varepsilon}$ by □y increment x by 1
(1,0) (1,1) (2,1)	1 -4 -1	pixel(1,0) since $\underline{\varepsilon}$ is positive: increment y by 1 decrement $\underline{\varepsilon}$ by □x increment $\underline{\varepsilon}$ by □y increment x by 1
(2,1) (3,1)	-1 2	pixel(2,1) increment $\underline{\varepsilon}$ by □y increment x by 1
(3,1) (3,2) (4,2)	2 -3 0	pixel(3,1) since $\underline{\varepsilon}$ is positive: increment y by 1 decrement $\underline{\varepsilon}$ by □x increment $\underline{\varepsilon}$ by □y increment x by 1
(4,2)	0	pixel(4,2)

Fig. 30

This last one is the algorithm implemented on the costmap as can be seen in Fig. 31. It produces a rasterization of the line going from one point to another. The result of all the lines is the **polygonal chain.**
In this way the ego-vehicle can not think to have the right to pass in the middle of the trajectory as it considers it as an actual obstacle marked on the costmap, thus not traversable.

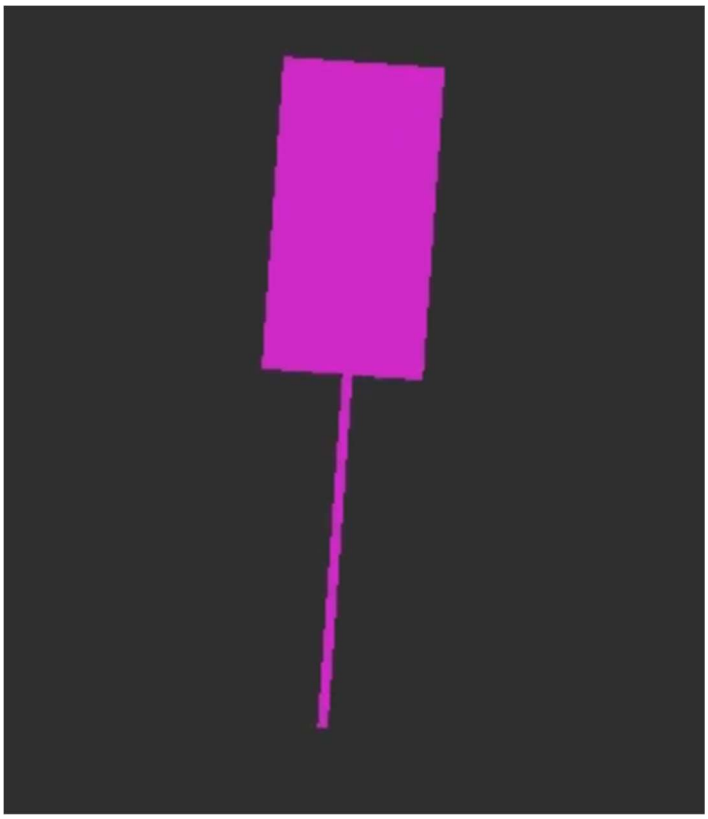

Fig. 31

4.2 Trajectory prediction

To test the Kalman filter it has been decided to build up some scenarios that should represent different profiles of motion that the obstacles can happen to follow. This has been done to evaluate the goodness of the forecast that the prediction algorithm provides.
In MATLAB it has been defined an actual path that the obstacle is supposed to follow (built using equations of motion).
This path is given to the Kalman as it was given by the sensors, so introducing a bounded error: Ibeo sensors have an accuracy of around 1.5 meters for the detection of an object so it has been decided to not go far from the reality.
At every step the filter predicts 40 steps ahead. It is possible to compare the actual position that the obstacle is going to have at the 40^{th} step (retrieved from the path that was built) and the prediction the filter does. It has been decided to set this difference as **KPI** (**Key Performance Indicator**).

Id	Name of the scenario	Comments about the creation of the scenario	Values for velocities and accelerations
1	straight with constant speed	Equations of motion with constant speed	$v_x = 30\ m/s, v_y = 0\ m/s$ $a_x = 0\ m/s^2, a_y = 0\ m/s^2$
2	straight with constant acceleration	Equations of motion with constant acceleration	$v_x = 3\ m/s, v_y = 0\ m/s$ $a_x = 1\ m/s^2, a_y = 0\ m/s^2$
3	straight with constant deceleration	Equations of motion with constant acceleration, but this time with a negative value	$v_{x0} = 30\ m/s, v_y = 0\ m/s$ $a_x = -0.5\ m/s^2, a_y = 0\ m/s^2$
4	straight with ramp acceleration	Equations of motion with a uniformly varying acceleration	$v_{x0} = 0\ m/s, v_y = 0\ m/s$ $a_{x0} = 0\ m/s^2, a_y = 0\ m/s^2$ $a_{xFinal} = 2.5\ m/s^2$
5	turning (as in road curve)	Equations of motion with constant acceleration, negative for one axis and positive for the other one	$v_{x0} = 30\ m/s, v_{y0} = 0\ m/s$ $a_x = -1\ m/s^2, a_y = 0.8\ m/s^2$
6	circumference (as in roundabout)	Sinusoidal profile both for x and y axes	$v_{xMax} = 40\ m/s, v_{yMax} = 40\ m/s$ $a_{xMax} = 3\ m/s^2, a_{yMax} = 3\ m/s^2$
7	overtaking path	Used a MATLAB function to have a smooth transition from 0 to 3 meters	$v_{x0} = 25\ m/s, v_{y0} = 1.5\ m/s$ $a_x = 0\ m/s^2, a_y = 0.8\ m/s^2$
8	sinusoidal path	Sinusoidal path only on the y axis	$v_x = 25\ m/s, v_{yMax} = 3\ m/s$ $a_x = 0\ m/s^2, a_{yMax} = 0.45\ m/s^2$

Only some motion models have been evaluated and their results have been compared. Future work could be to apply the Kalman filter to the others described in the State of the art.

4.2.1 Kalman filter: general aspects

The Kalman filter used in this project is a 40-steps ahead predictor.

Here are listed the matrices that are used:

- the matrices for the system S (A,C) change with respect of the motion model implemented.

$$x\,(t+1) = A \cdot x(t)\; + v_1(t)$$
$$y\,(t) = C \cdot x(t) + v_2(t)$$

  The A matrix is a simplification of the reality and represents the dynamics of the motion.
  The C matrix represents which kind of outputs the system is built to have.
  These matrices depend on the time between one measurement and the next one, that is $dt = 40ms$.
- the matrix P, the prediction error variance, is always initialized with relatively high values for the variance of each state; this to say that the initial values of the predicted trajectories are not well trusted
- $v_2(t)$ is the measurement noise that is characterized by the variance matrix V_2: it represents how much the measurements are accurate:

$$V_2 = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$$

- $v_1(t)$ is the process noise that is characterized by the variance matrix V_1: it represents the idea/feature that the state of the system changes over time, but it is not known the exact detail of when/how those changes occur, and thus it is needed to model them as a random process. For what regards the matrix itself, the values of the last two positions of the diagonal are a little higher due to the fact that state variables concerning the jerk values are less trusted, given that it is the third derivative of the position, so a higher variance is supposed:

$$V_1 = \begin{bmatrix} 0.00001 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.00001 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.00001 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.00001 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.00001 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.00001 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.001 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.001 \end{bmatrix}$$

In Fig. 32, a block diagram can be found. It is highlighted in red the part this thesis is interested in, the prediction.
The initial condition for the algorithm is the first measurement the sensor gives.

The error of the prediction has been evaluated in two different ways:

- the distance between the actual position of the moving obstacle and the predicted one at the 40^{th} step, both in x and y direction (previously stated as the KPI)

- the **mean squared error** (**MSE**): measures the average of the squares of the errors, that is, the average squared difference between the estimated values and the actual ones. The MSE is a measure of the quality of an estimator: it is always non-negative, and values closer to zero are better.
  The formula is:

$$MSE = \frac{1}{N}\Sigma(\varepsilon(t))^2$$

  where N is the dimension of the vector ε, that is the error in the prediction (along the x-axis or the y-axis).

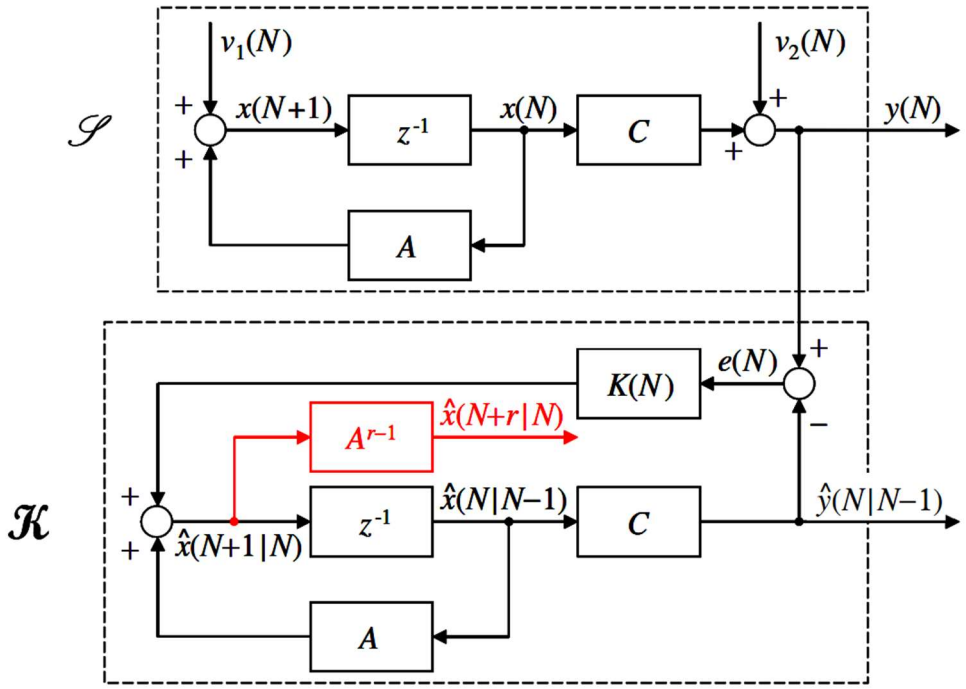

Fig. 32

4.2.2 CV (Constant Velocity)

This model considers to have constant velocity. The matrices of the system are:

$$A = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

4.2.2.1 Straight with constant speed

It can be seen in Fig. 33 that, at the beginning, the estimate of the 40^{th}step ahead is weak because the Kalman is still adjusting itself. After the transient, the prediction becomes more accurate and the error goes approximately to 0, it is bounded in a range of less than 10 cm.

Over a simulated serie of 1000 measurements, the maximum error with the used initial conditions is 30 cm on x-axis and 90 cm on y-axis.

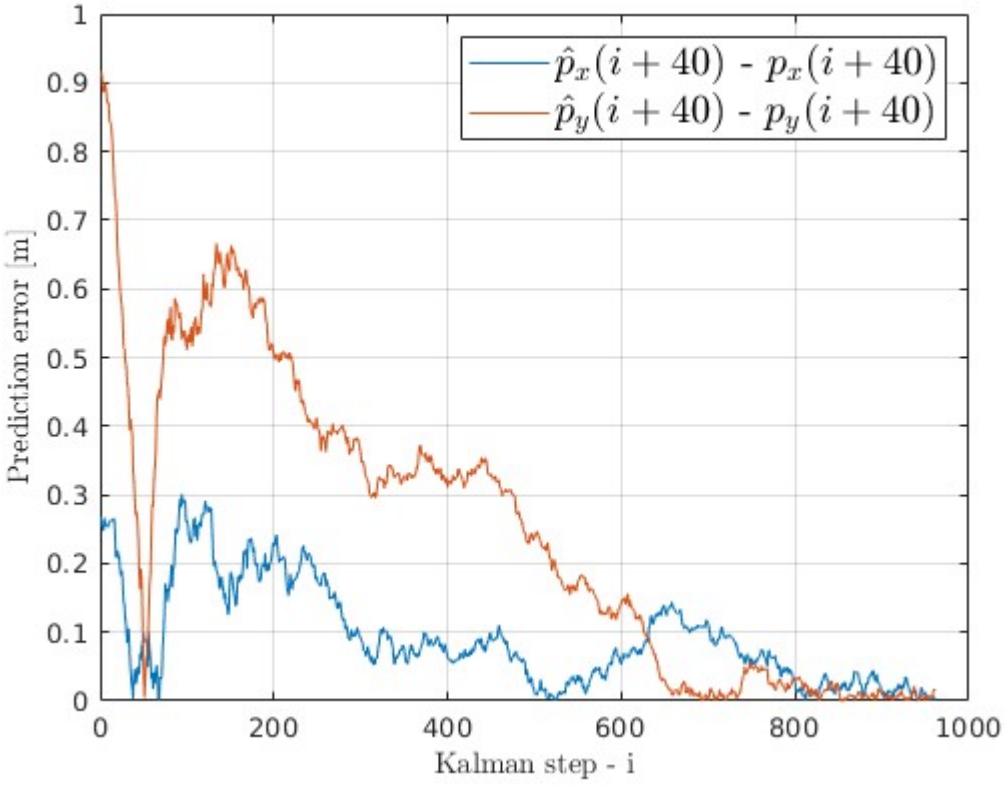

Fig. 33

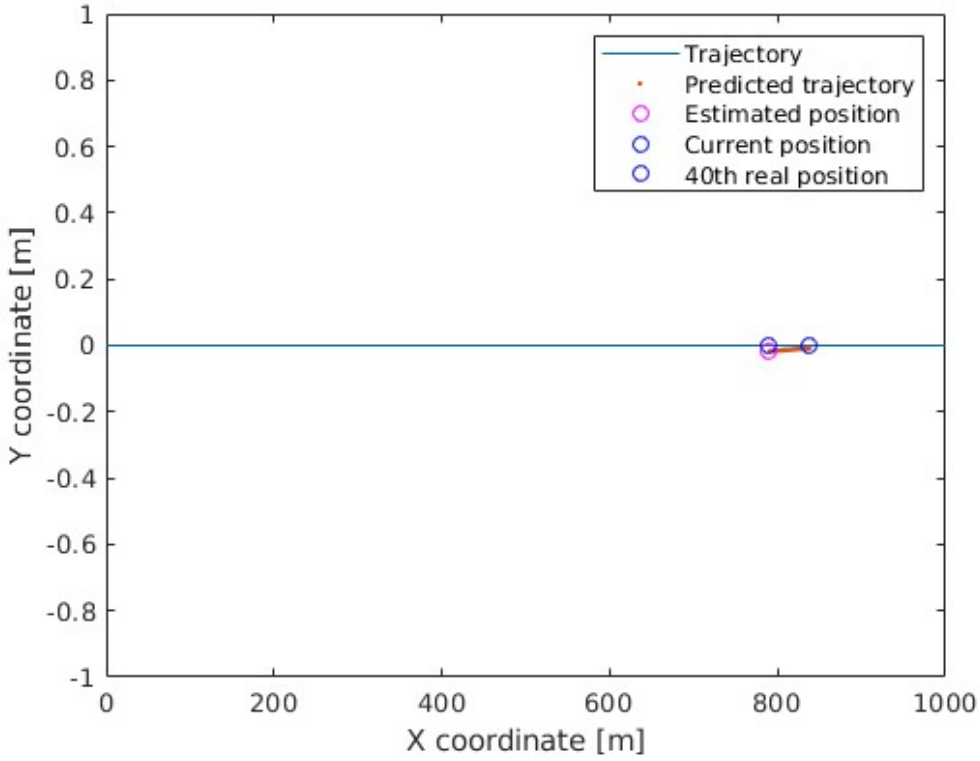

Fig. 34

This scenario is well handled because the model is at constant speed and the obstacle is supposed to travel with a constant speed. In Fig. 34, it can be shown a simulation done in MATLAB of the path the obstacle is actually following: in red, the prediction the filter does, in blue the actual path. The predicted path follows well the actual one.

4.2.2.2 Straight with constant acceleration

It can be seen that the error goes very high because the model can not handle the acceleration of the moving obstacle.
As can be seen in Fig. 35, the maximum error with the used initial conditions is approximately 60 meters on x-axis, even though it is very precise for the y-axis: this comes from the fact that the acceleration is considered only on the x-axis, on the y-axis the equations are the same as the previous case.

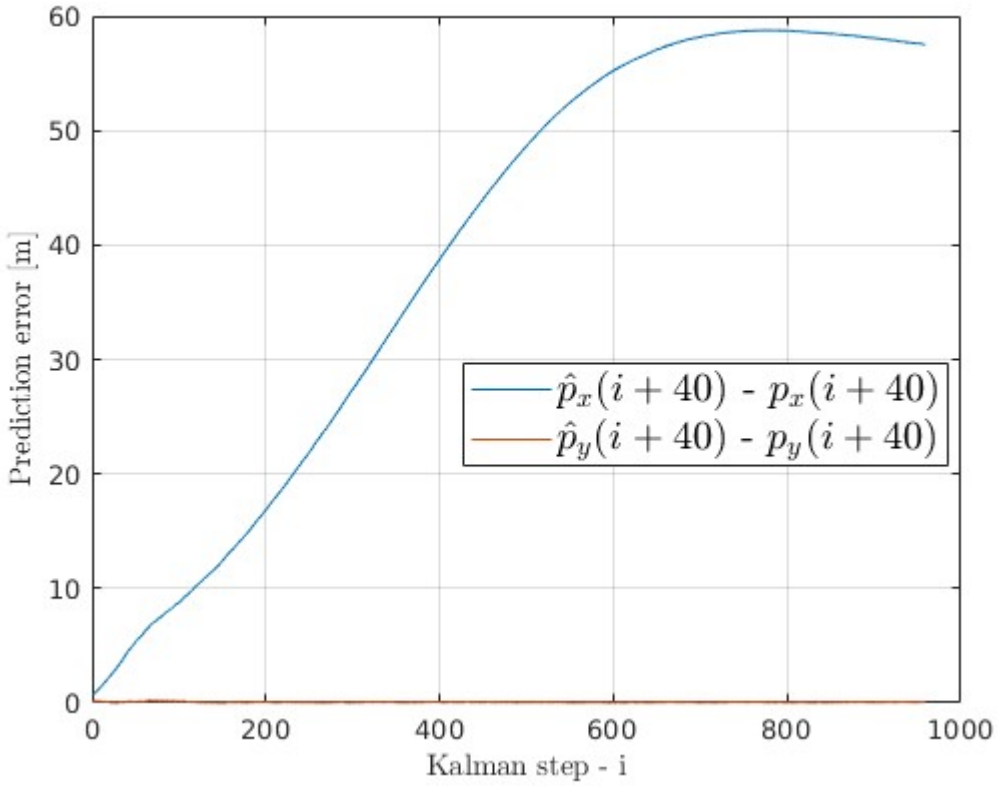

Fig. 35

In Fig. 36 there is a simulation of the path the obstacle is actually following. It can be seen that the predicted path deviates from the actual one because the filter is not capable to follow it. The actual obstacle is always in an advanced position with respect to the predicted one, this due to the fact that a constant velocity model can not handle a motion profile with constant acceleration.

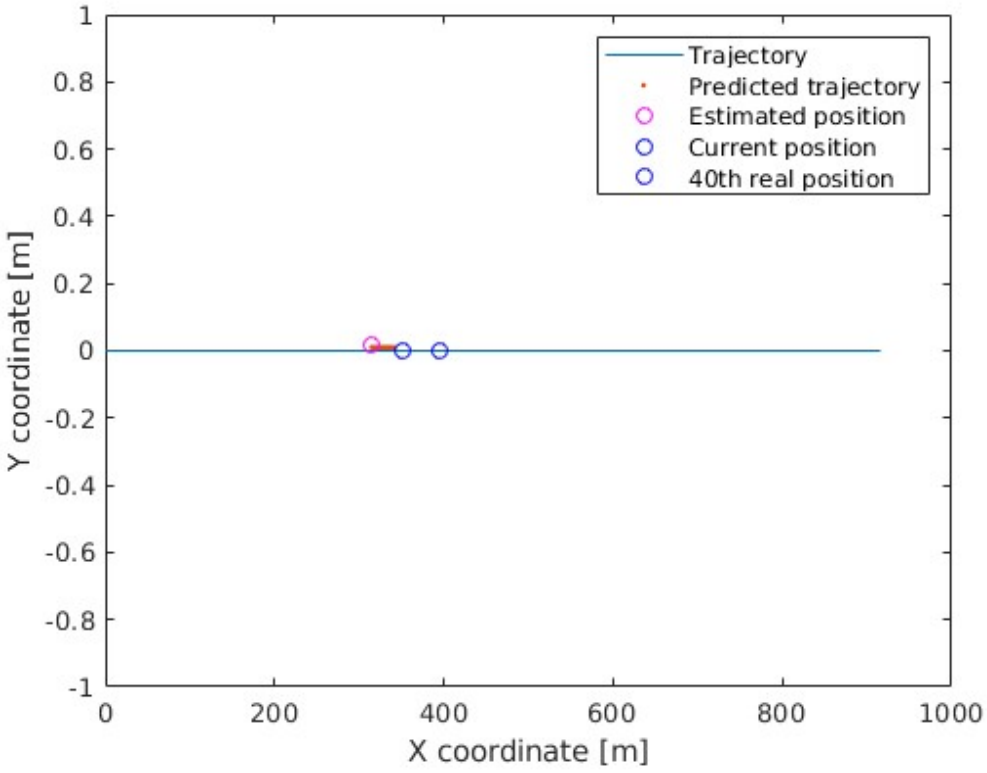

Fig. 36

4.2.2.3 Straight with constant deceleration

As expected, as this is the reciprocal of the acceleration case, the error goes very high for x-axis, while on the y-axis after the transient it follows the path.
As can be seen in Fig. 37, the maximum error with the used initial conditions is approximately 30 meters on x-axis, even though it is quite precise for the y-axis.
This is due to the fact that the deceleration is considered only on x-axis.

In Fig. 38, it can be seen that the predicted path deviates from the actual one. The actual obstacle is always in a backward position with respect to the predicted one, this due to the fact that a constant velocity model can not handle a motion profile with constant deceleration.

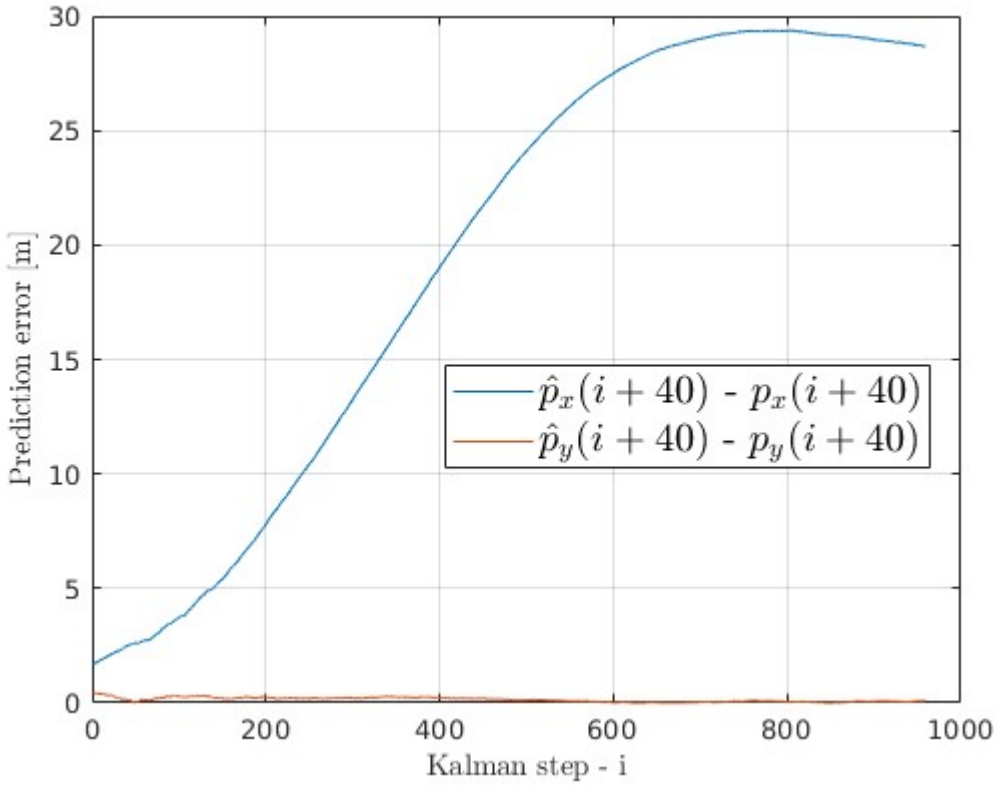
30
25
20
15
10
5
0
0
200
400
600
800
1000
Prediction error [m]
Kalman step - i
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$

Fig. 37

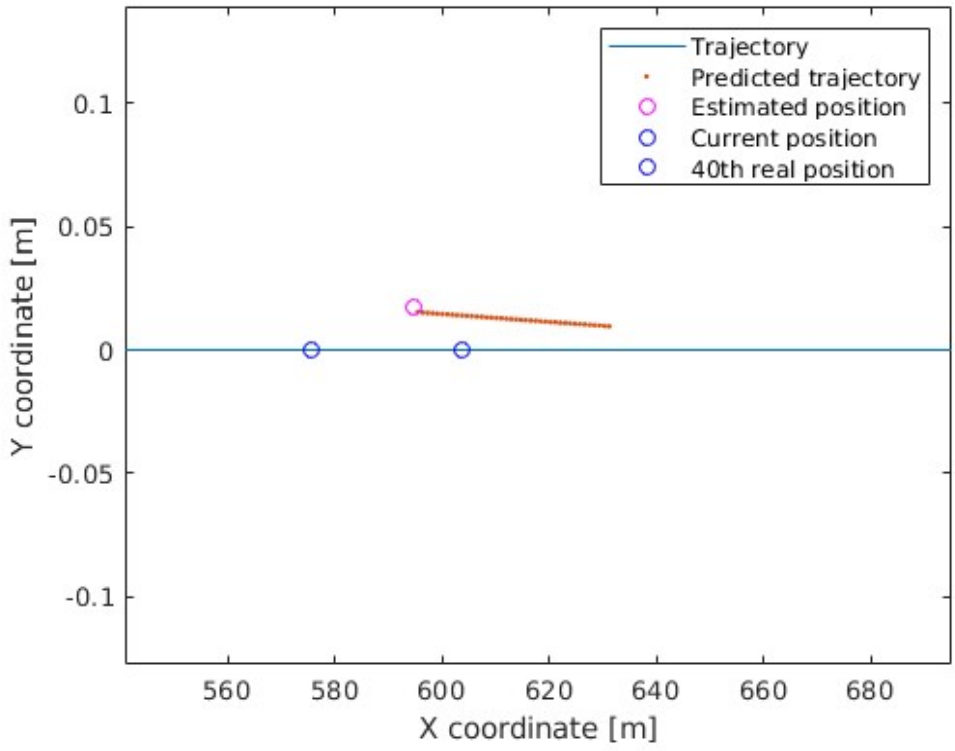
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
0.1
0.05
0
-0.05
-0.1
560
580
600
620
640
660
680
Y coordinate [m]
X coordinate [m]

Fig. 38

4.2.2.4 Straight with ramp acceleration

The error goes very high for x-axis, while on the y-axis it reaches almost 0.

As can be seen in Fig. 39, the maximum error with the used initial conditions is approximately 100 meters on x-axis, even though it is quite precise for the y-axis, this due to the fact that the acceleration is considered only on the x-axis.

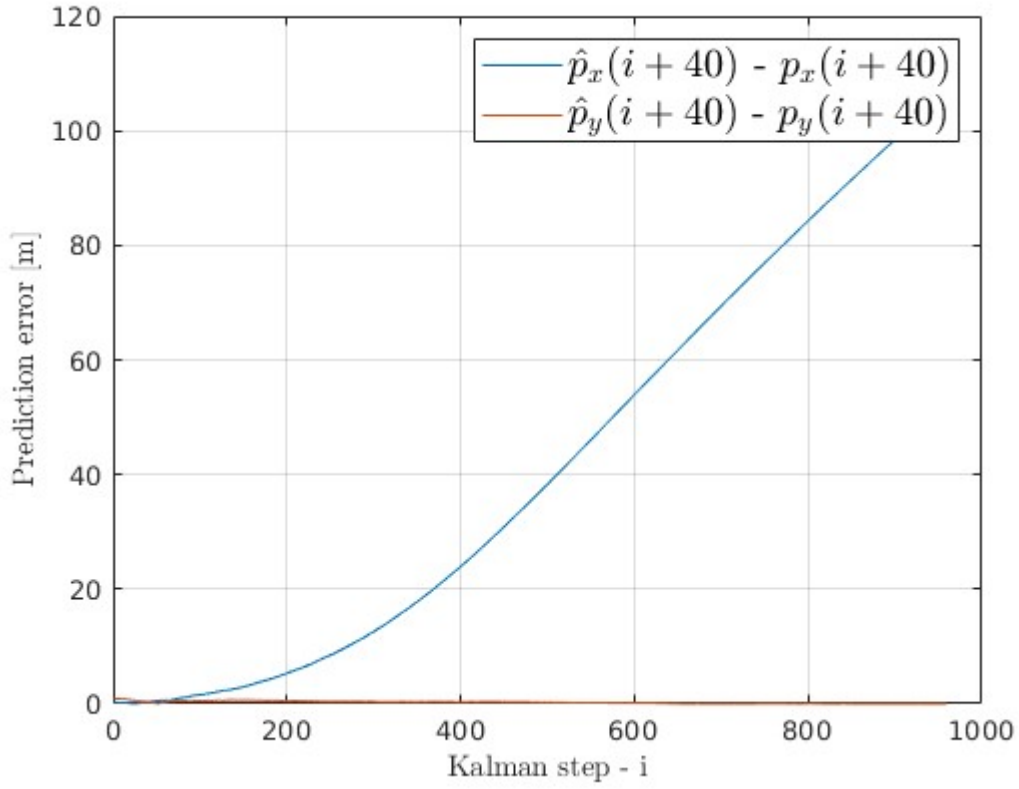

Fig. 39

In Fig. 40, the simulation shows that the predicted path deviates from the actual one. The actual obstacle is always in an advanced position with respect to the predicted one. This is almost as the case of the constant acceleration, the difference is that having a ramp acceleration makes it even more difficult for the constant velocity model to follow the actual trajectory.

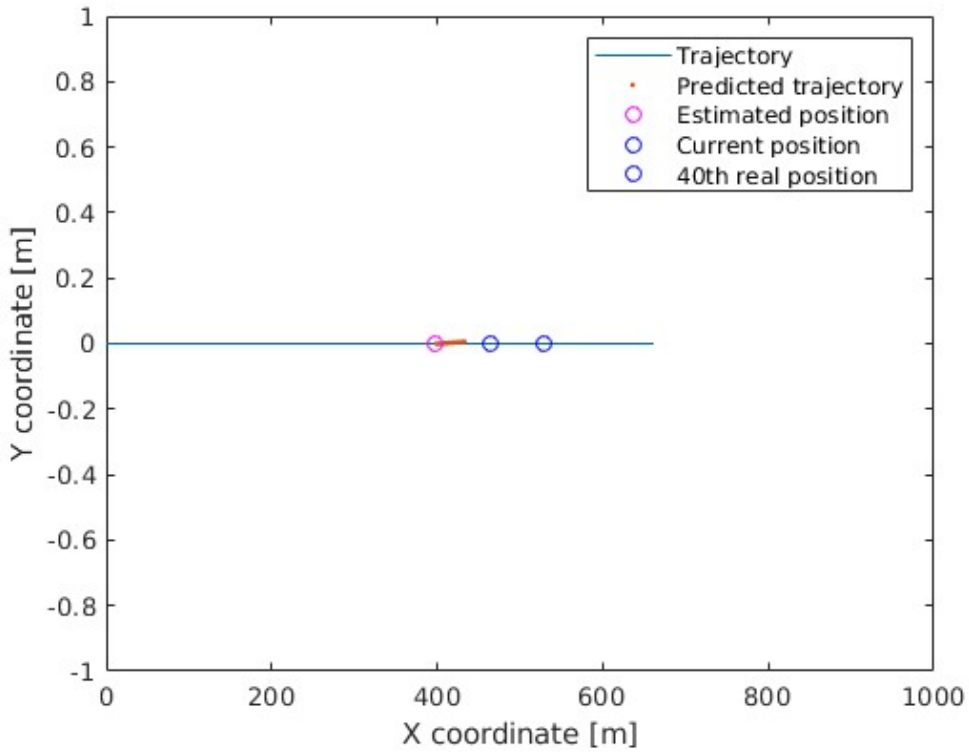

Fig. 40

4.2.2.5 Turning (as in road curve)

The error goes very high both for x-axis and y-axis.
As can be seen in Fig. 41, the maximum error with the used initial conditions is approximately 60 meters on x-axis and 50 meters for the y-axis.

In Fig. 42, it can be seen that the predicted path deviates from the actual one effectively. The actual car is far away from the predicted path, this due to the fact that in the turning case both x and y axes have information about acceleration.

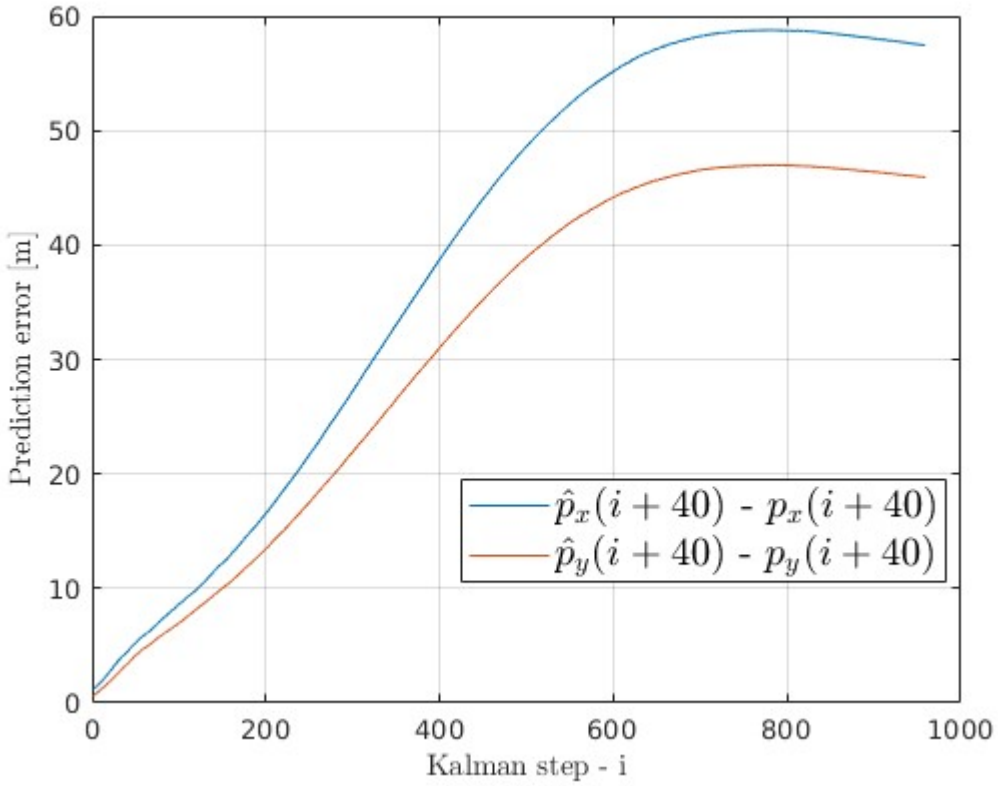
Prediction error [m]
Kalman step - i
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$

Fig. 41

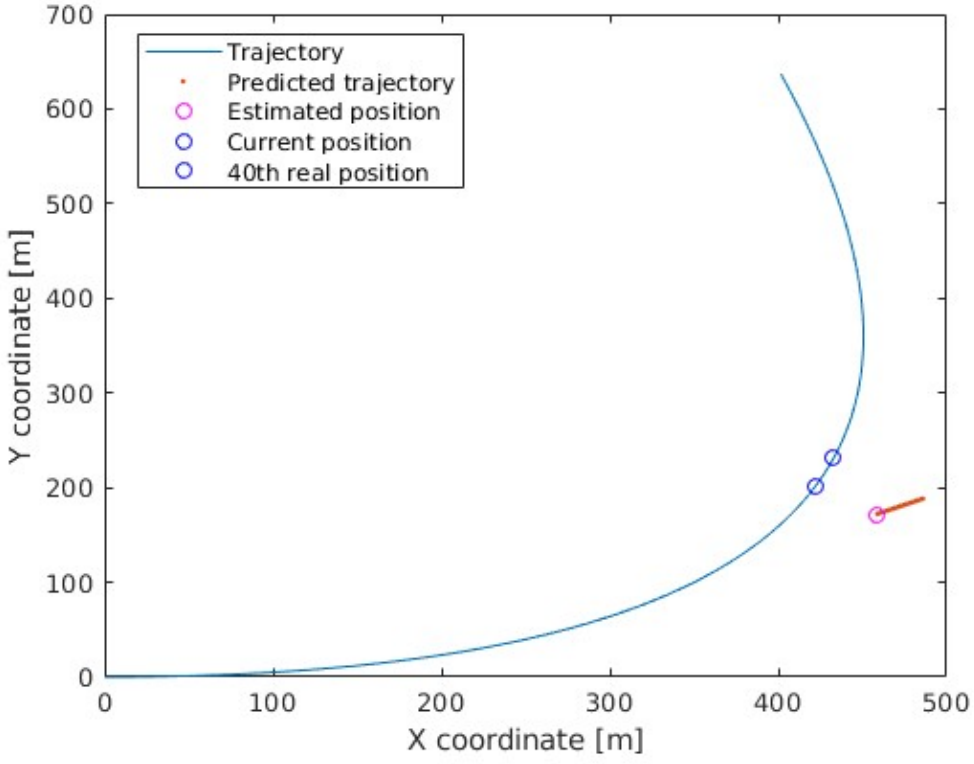
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]

Fig. 42

4.2.2.6 Circumference (as in roundabout)

With the circumference scenario it's possible to see that the Constant Velocity model is really inappropriate to predict the trajectory of an obstacle. This case demonstrates that this particular modellization of the dynamics can't follow an x and y acceleration which changes in time with a sinusoidal profile.
Fig. 43 shows the error in x and y position of the 40^{th} predicted points. It follows a sinusoidal path and reaches its maximum at around 90 meters far from the real position that the obstacle has occupied.
This bad behavior is clearer in Fig. 44 where there is the actual trajectory of the car and the predicted path. The object starts at position of [500, 0] and never follows in a good manner the actual trajectory.

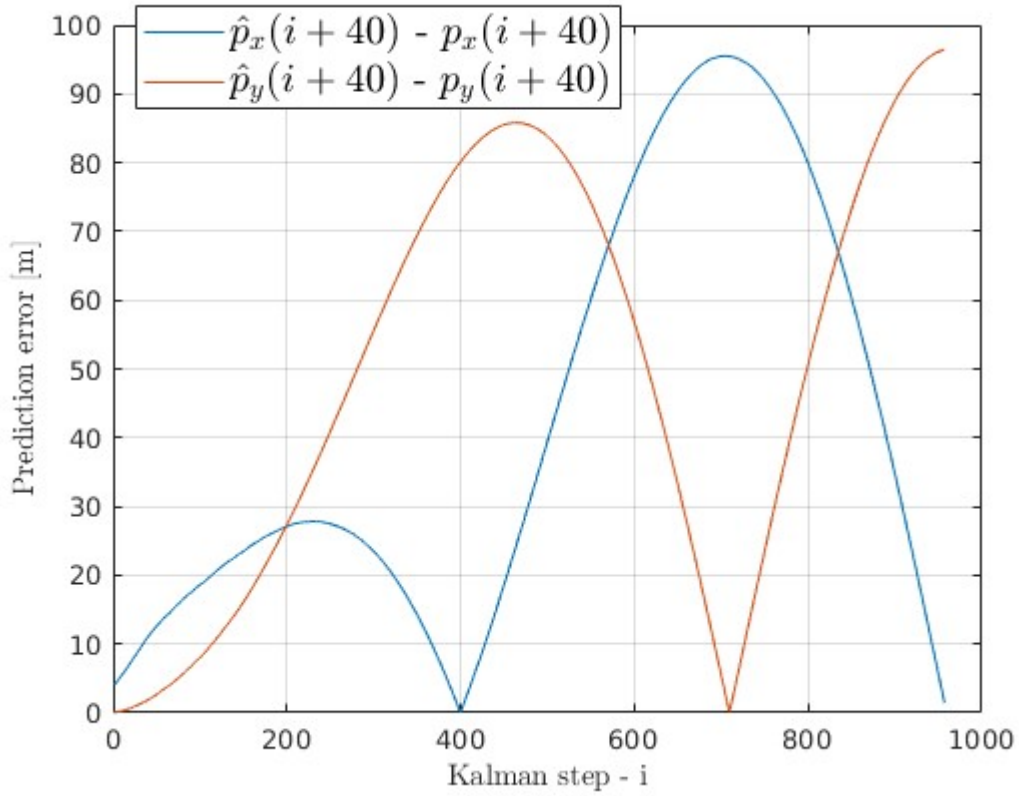

Fig. 43

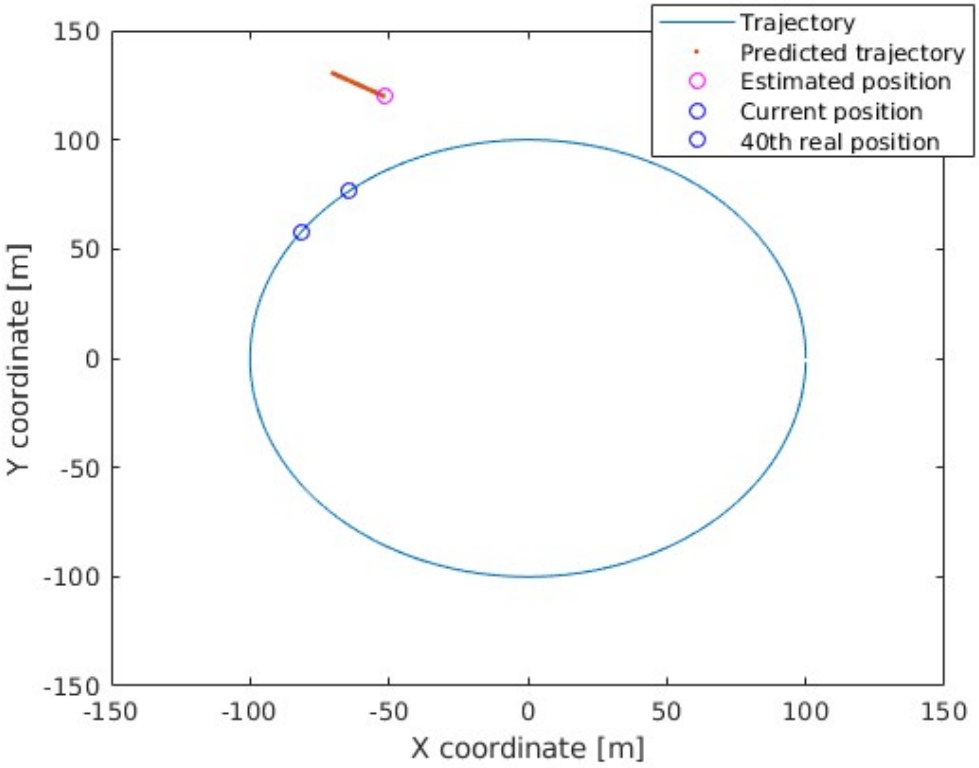

Fig. 44

4.2.2.7 Overtaking path

As before with the circumference scenario, here also it's possible to see that the model is not usable to have a good prediction of the trajectory of an obstacle in special scenarios, but anyway of everyday life.
This one takes into consideration a long overtaking, as often happens on the highway.

As can be seen in Fig. 45, the error reaches its maximum at about 2.5 meters far from the real position.
This bad behavior is clearer in Fig. 46 where there is the real trajectory of the car and the predicted path. As can be seen, on the x-axis the error is not so high, in fact the actual position at the 40^{th}step and the predicted one are aligned, but on the y axis it is as if the predicted path can not understand that overtaking must be undertaken.

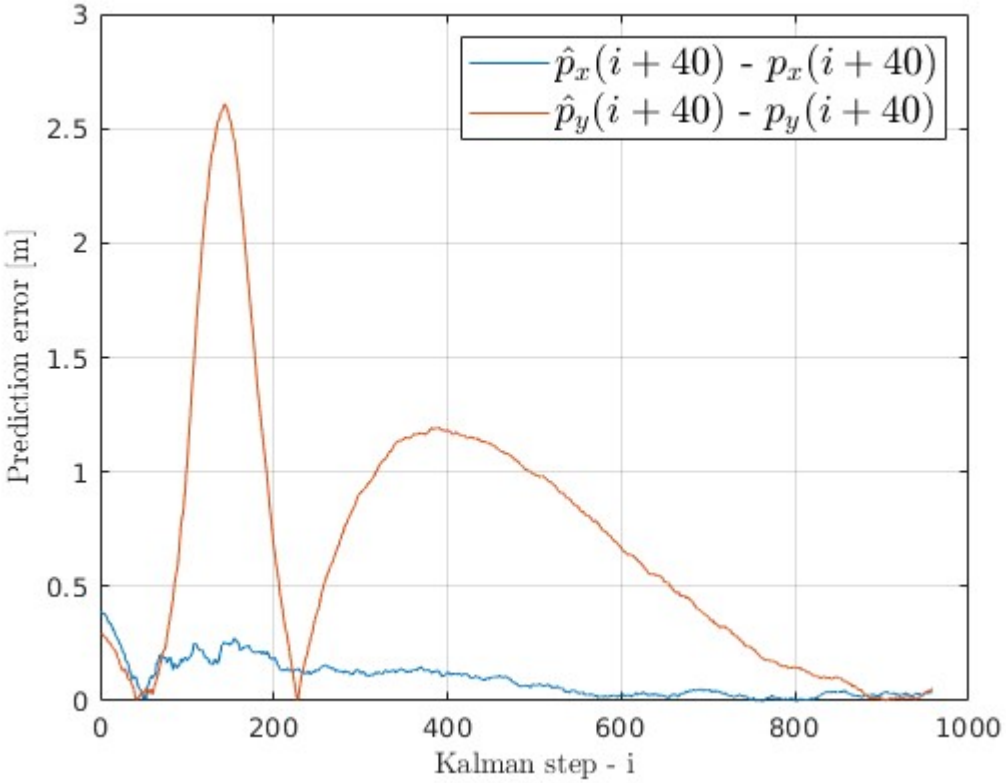
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
0.5
1
1.5
2
2.5
3
0
200
400
600
800
1000

Fig. 45

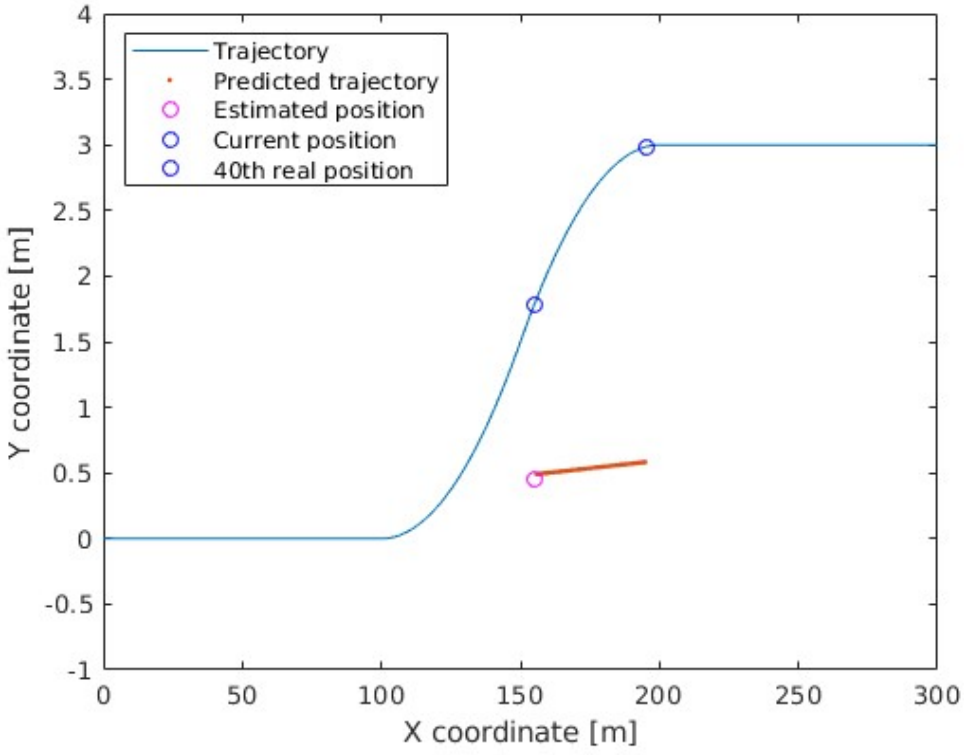
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
0
50
100
150
200
250
300

Fig. 46

4.2.2.8 Sinusoidal path

This particular path is a generalization of the previous one. It considers the scenario of a series of curves or an overtaking with a return to the carriageway. As can be seen in Fig. 47, the error reaches its maximum at about 18 meters far from the real position.

In Fig. 48 there is the result of the simulation that has been made. As before, on the x axis the error is almost 0, actual and predicted position are aligned, but on the y-axis the error is very high.

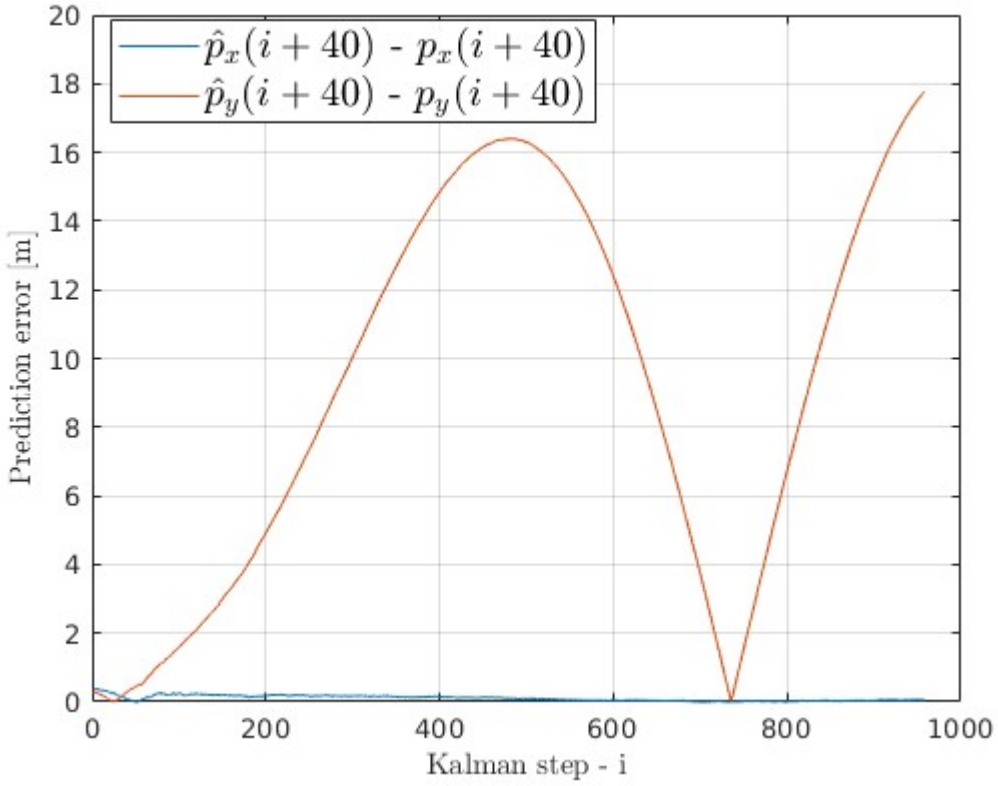

Fig. 47

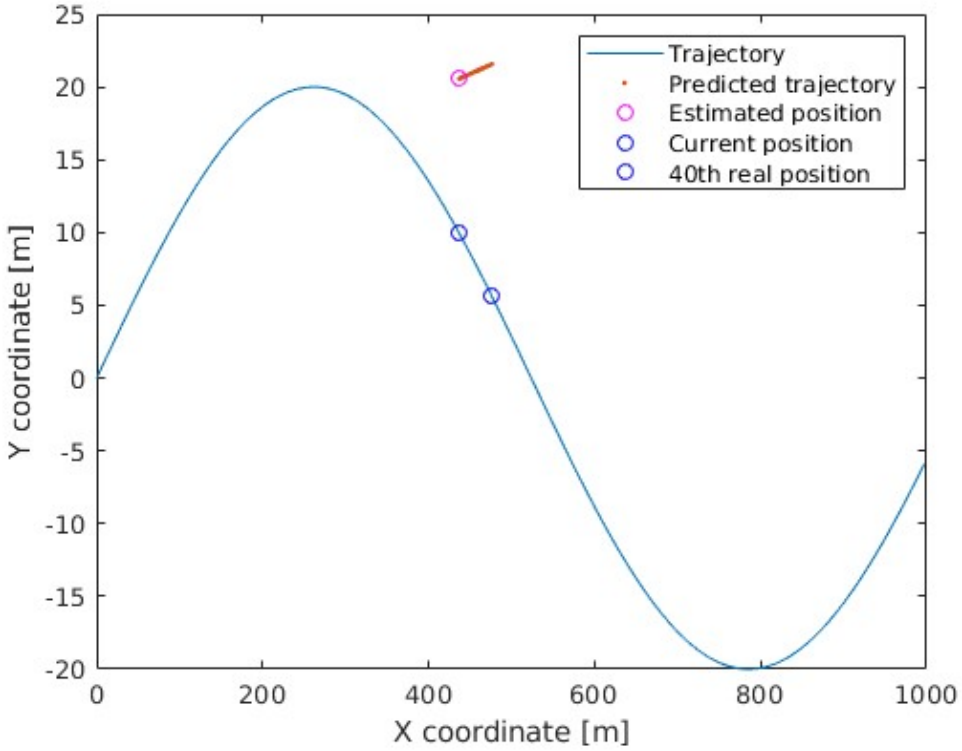

Fig. 48

4.2.2.9 MSE report

Here are reported the MSE for each axis in all scenarios:

Scenario	$\boldsymbol{MSE_X}$	MSE_Y
Straight with constant speed	$1.42 \cdot 10^{-2}$	0.11
Straight with constant acceleration	$1.93 \cdot 10^{3}$	$4.11 \cdot 10^{-3}$
Straight with constant deceleration	$4.77 \cdot 10^{2}$	$2.83 \cdot 10^{-2}$
Straight with ramp acceleration	$2.92 \cdot 10^{3}$	0.11
Turning (as in road curve)	$1.93 \cdot 10^{3}$	$1.24 \cdot 10^{3}$
Circumference (as in roundabout)	$2.71 \cdot 10^{3}$	$3.33 \cdot 10^{3}$
Overtaking path	$1.45 \cdot 10^{-2}$	0.81
Sinusoidal path	$1.74 \cdot 10^{-2}$	$1.13 \cdot 10^{2}$

It is highlighted also by the MSE how the CV model only predicts well the path in the “Straight with constant speed” scenario. In all other cases the error reaches too high values, so the model has to be discarded. The most important problem is that the other scenarios are real life ones, it happens really frequently that an obstacle is undertaking a curve like in a “Turning” scenario, so another motion model has to be evaluated.

4.2.3 CA (Constant Acceleration)

This model considers to have a constant acceleration. The matrices of the system are:

$$A = \begin{bmatrix} 1 & 0 & dt & 0 & \frac{1}{2}dt^2 & 0 \\ 0 & 1 & 0 & dt & 0 & \frac{1}{2}dt^2 \\ 0 & 0 & 1 & 0 & dt & 0 \\ 0 & 0 & 0 & 1 & 0 & dt \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.2.3.1 Straight with constant speed

It can be seen from Fig. 49 that, at the beginning, the estimate of the 40^{th}step ahead is weak but after the transient, the prediction becomes more accurate and the error goes approximately to 0, it is bounded in a range of 10 cm.

The maximum error with the used initial conditions was 50 cm on x-axis and 1.5 meter on y-axis.
These results are completely comparable with the one of the CV model because the speed of the obstacle is constant so the acceleration is 0 and the model can be described in the same way as the CV one.

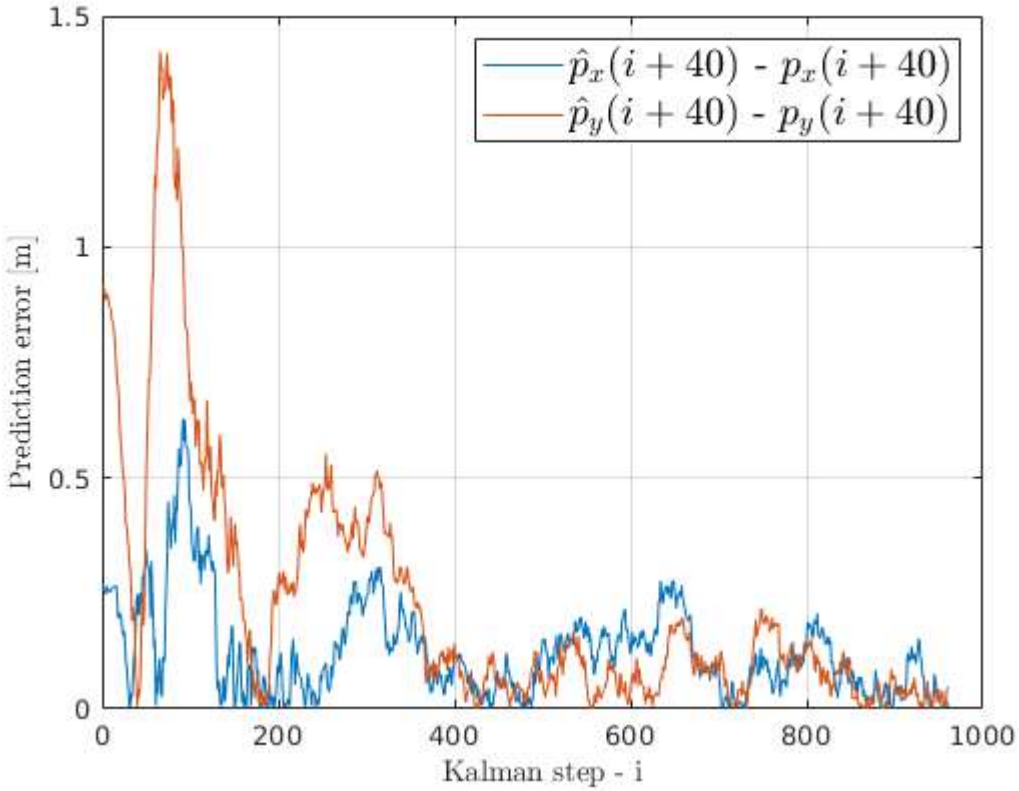

Fig. 49

In Fig. 50 there is a simulation of the path the obstacle is actually following. It can be seen that the predicted path follows well the actual one.

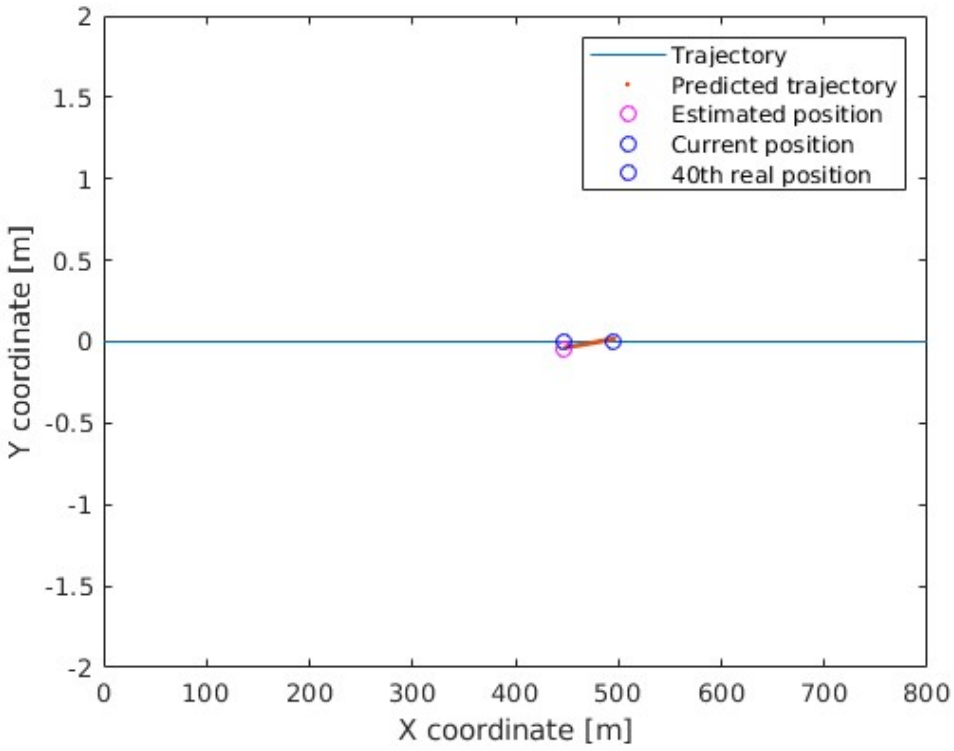

Fig. 50

4.2.3.2 Straight with constant acceleration

It can be seen in Fig. 51 that the error goes very high at the beginning but then it approaches the zero, in fact it is a constant acceleration model with a constant acceleration motion profile of the object.
The maximum error with the used initial conditions is approximately 5 meters on x-axis and 50 cm for the y-axis, but then both result to be bounded around 10 cm as before. The large error on the x axis can be explained with the fact that the acceleration is considered only on the x-axis.

As can be seen in Fig. 52, the simulation that has been done in MATLAB shows that the prediction follows the actual path.

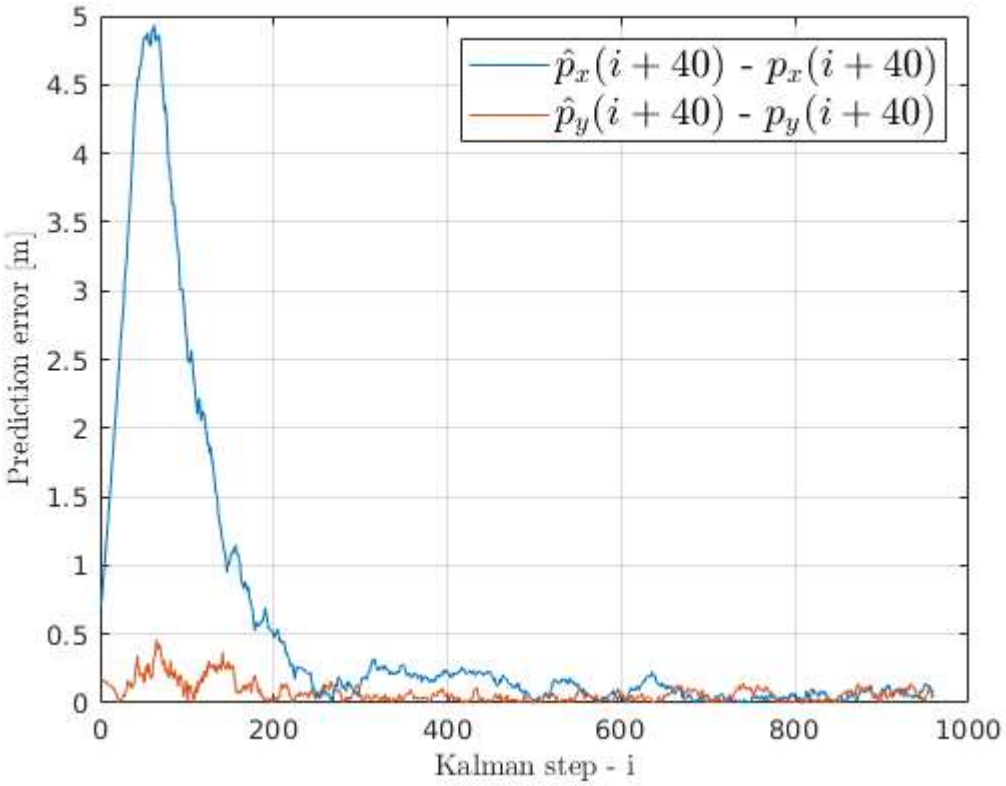
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
200
400
600
800
1000

Fig. 51

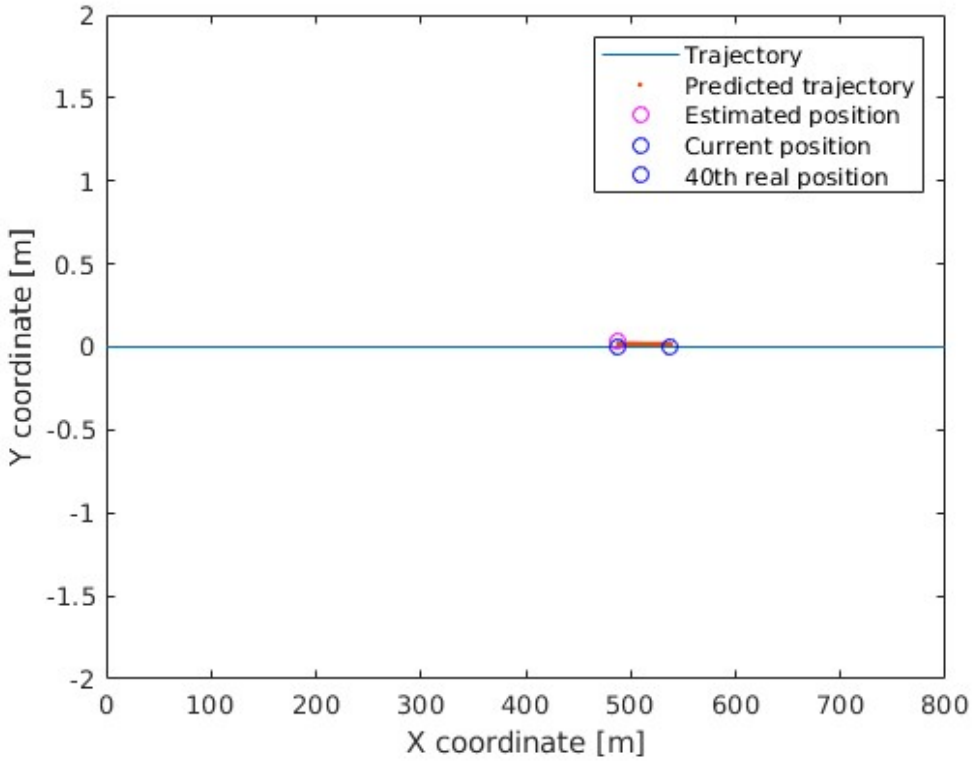
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
100
200
300
400
500
600
700
800

Fig. 52

4.2.3.3 Straight with constant deceleration

As expected, as this is the reciprocal of the acceleration case, the error goes high at the beginning but then it stabilizes around 0.

As can be seen in Fig. 53, the maximum error with the used initial conditions is approximately 2 meters on x-axis and 50 cm on y-axis.
The errors with respect to the acceleration case result to be different due to the different values of acceleration and deceleration used for the two scenarios.
The large error on the x-axis can be explained as deceleration is considered only on x-axis.

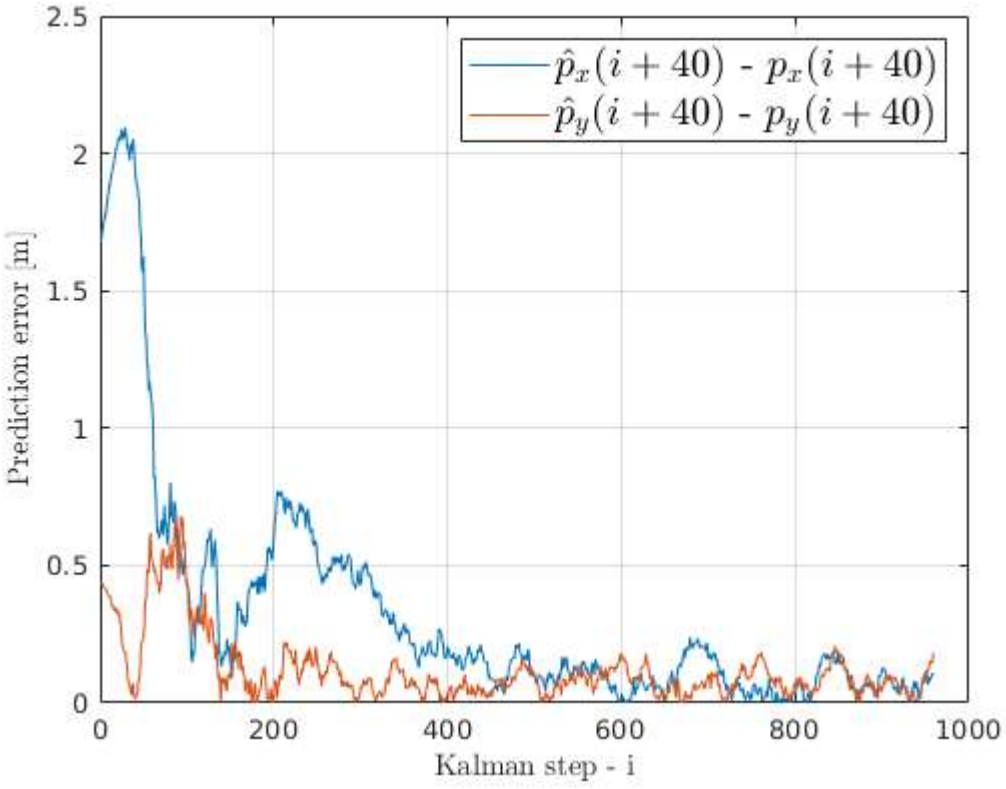

Fig. 53

As can be seen in Fig. 54, the simulation have as a result the same shown for the acceleration case, the predicted path can actually follow the actual one.

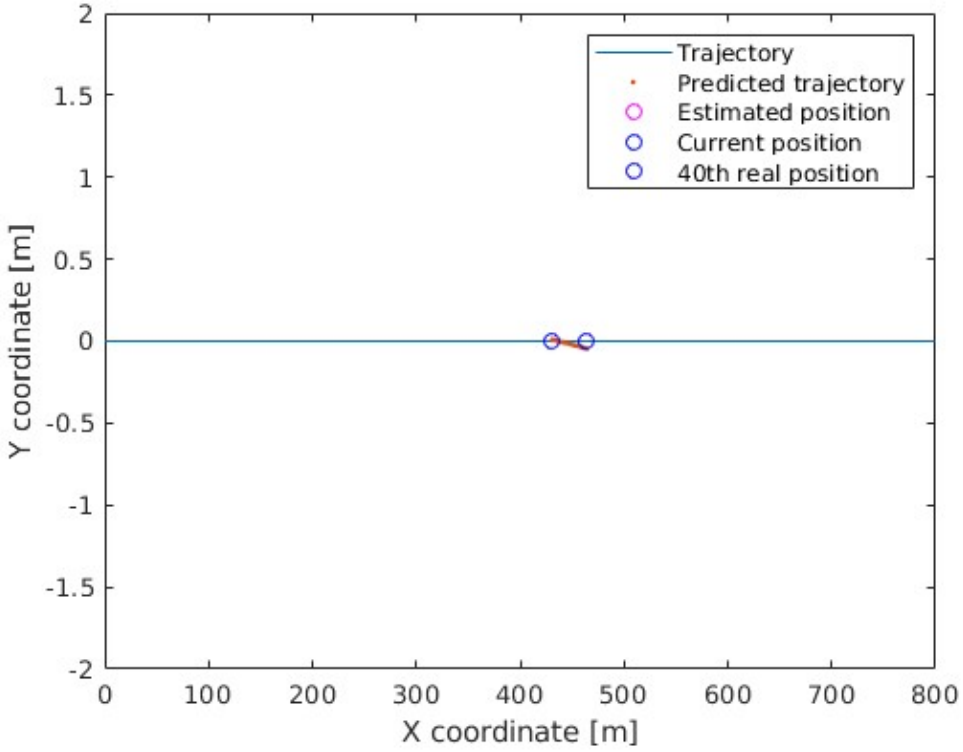

Fig. 54

4.2.3.4 Straight with ramp acceleration

In Fig. 55, it can be seen that the error goes very high for the x-axis, it reaches 6 meters of difference with respect to the actual position of the object, while on y-axis it is almost zero, this can be explained with the fact that the acceleration is only along x.
The overall behavior is explained by the fact that the acceleration is not constant so the model is not very capable to predict the exact path.

In Fig 56, it can be seen that in simulation the path is almost well followed on the y-axis but on the x-axis the actual obstacle happens to be in an advanced position with respect to the predicted one. It is not properly visible in the figure because on the x-axis there are values that change by units of 50 meters.

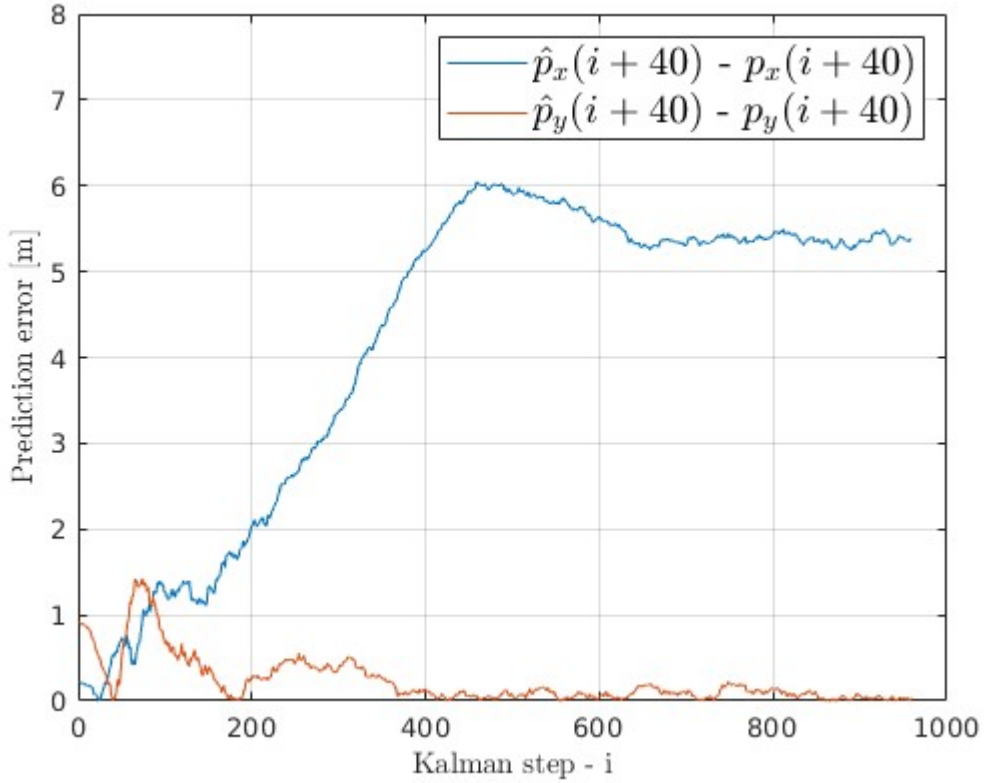
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
1
2
3
4
5
6
7
8
0
200
400
600
800
1000

Fig. 55

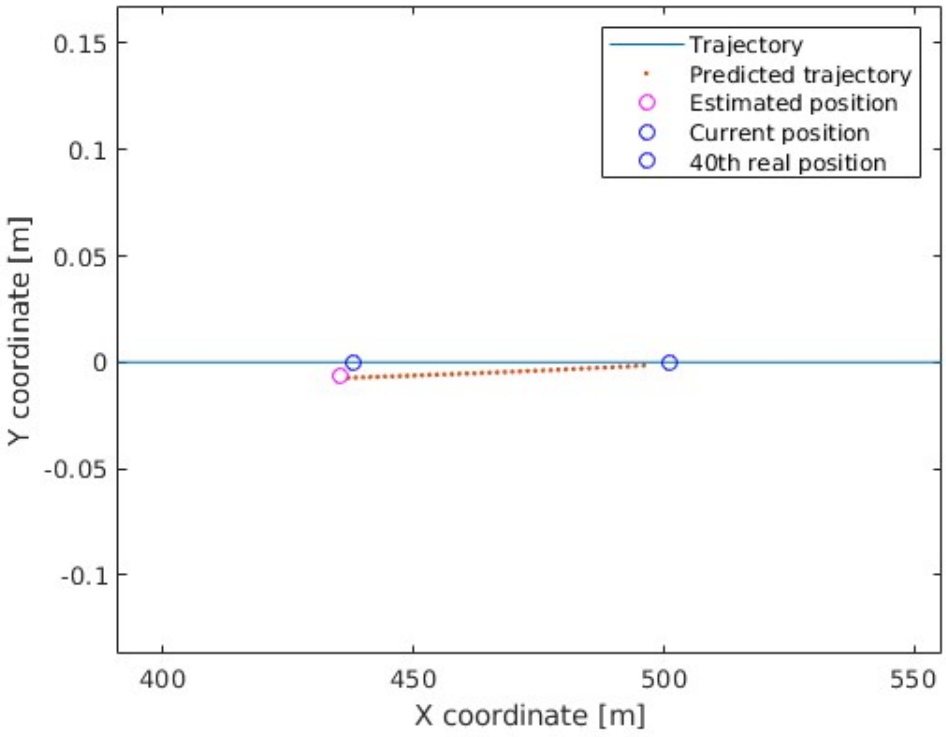
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
0.15
0.1
0.05
0
-0.05
-0.1
400
450
500
550

Fig. 56

4.2.3.5 Turning (as in road curve)

As can be seen in Fig. 57, the error goes high at the beginning but after the transient it goes to 0 almost. The maximum error on the x-axis is around 4 meters, while on y-axis is around 3.5 meters. At the end it results bounded around 10 cm. This good results comes from the fact that it has been set a constant acceleration during the turning phase.

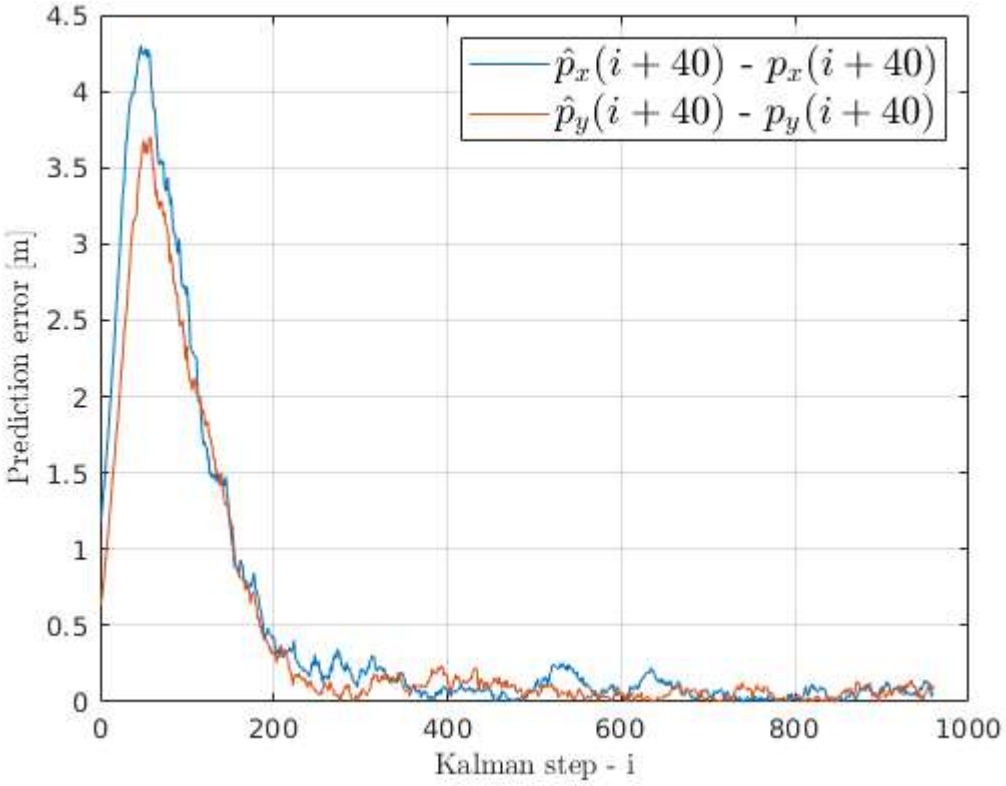

Fig. 57

The error goes to 0, this has the result that in simulation, as can be seen in Fig. 58, the predicted path follows very well the actual one.

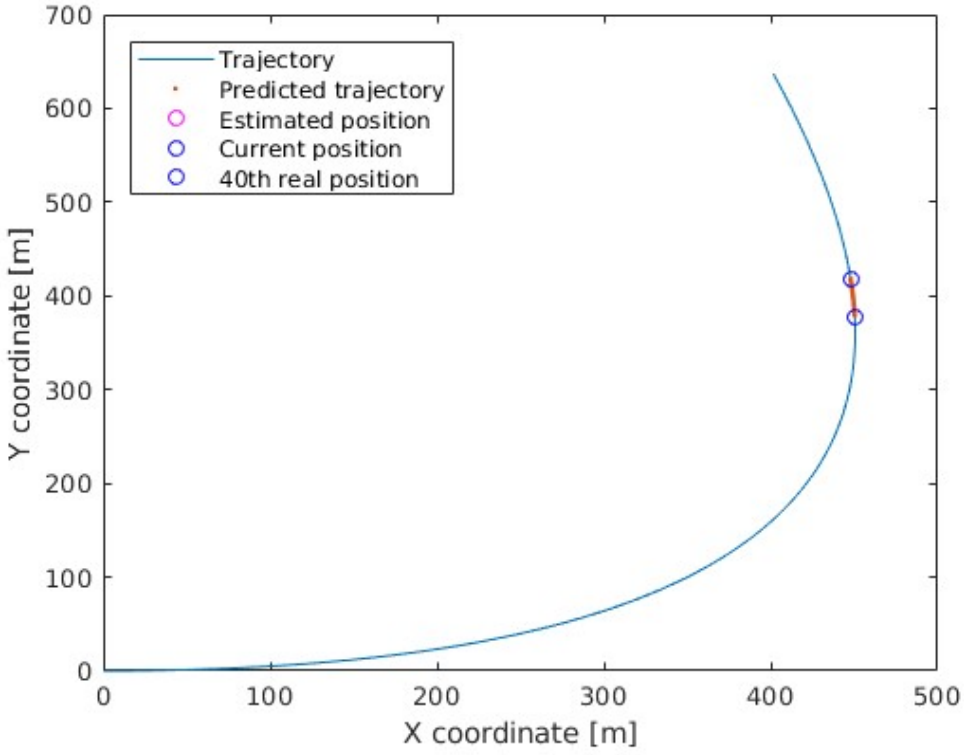

Fig. 58

4.2.3.6 Circumference (as in roundabout)

As can be seen in Fig. 59, this particular case has an error that has a sinusoidal profile that reaches its maximum at more than 30 meters of distance from the actual position, both for the x and y axes. This is due to the fact that a non-constant acceleration is present on both axes, in particular it has a sinusoidal profile.

In Fig. 60, the simulation shows that the high values in the error reflect on the fact that the predicted path is not following the actual path, it is really far from what it should be.

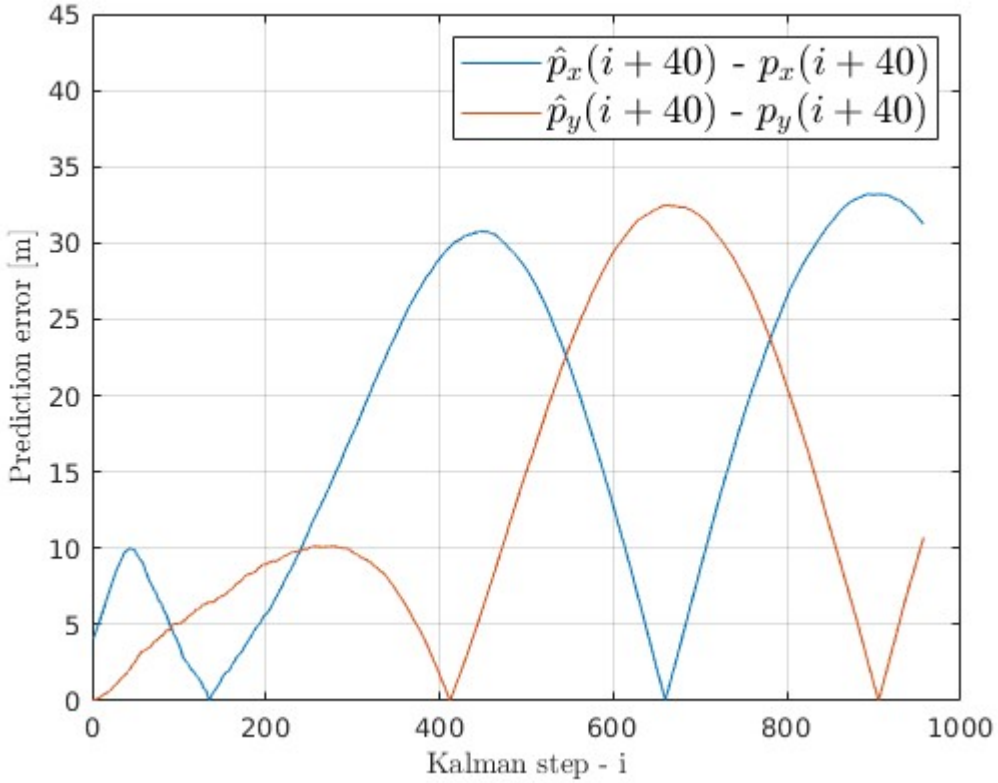
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
5
10
15
20
25
30
35
40
45
0
200
400
600
800
1000

Fig. 59

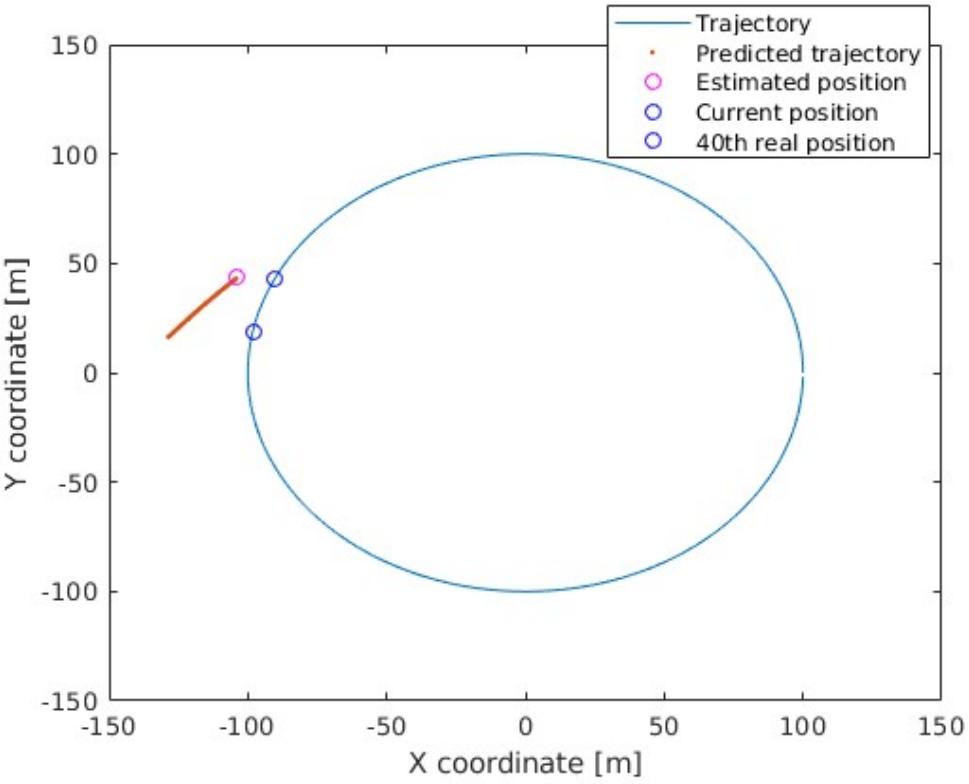
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
150
100
50
0
-50
-100
-150
-150
-100
-50
0
50
100
150

Fig. 60

4.2.3.7 Overtaking path

As can be seen in Fig. 61, the error is high at the beginning, it reaches around 50 cm for the x-axis and 2.5 meters for y-axis, but after a transient both reach values around the 0.
This behavior can be explained by the fact that during an overtaking there is a phase of changing acceleration and then the acceleration become constant again so the model is capable to follow the path.

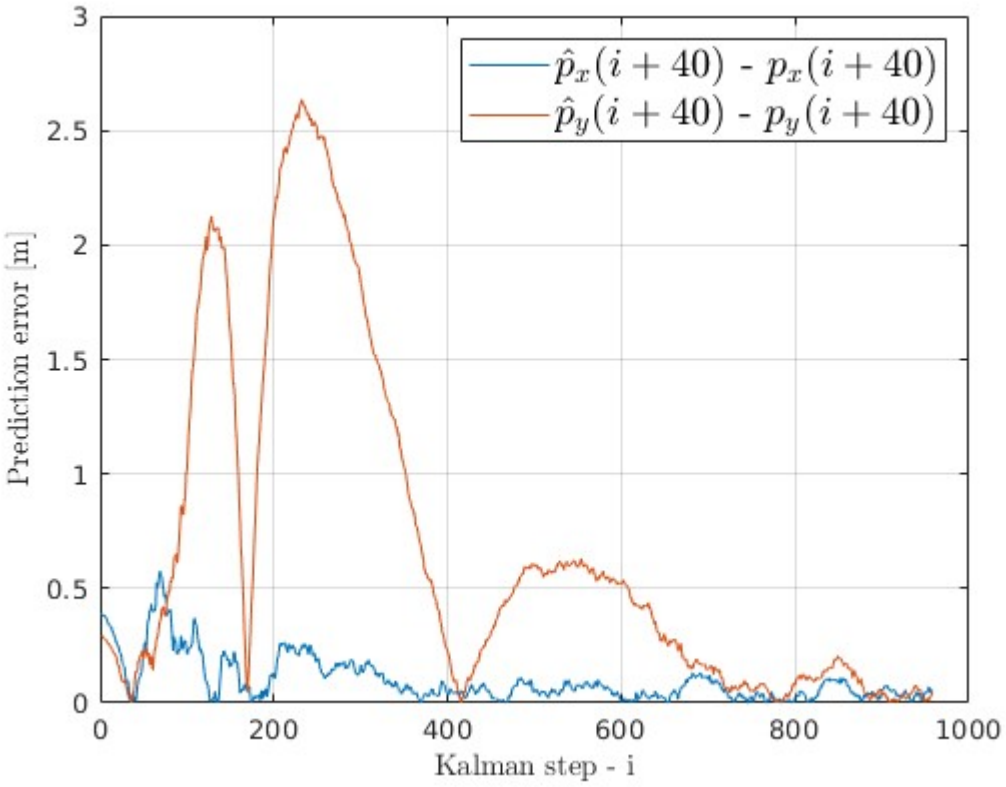

Fig. 61

In Fig. 62, it can be seen that during the changing acceleration phase, the predicted path is far away from the actual one. The x-axis coordinate is in general more precise than the y-axis one. It is anyway a better result than the one of the CV model that is not even capable of understanding if a turn has to be undertaken.

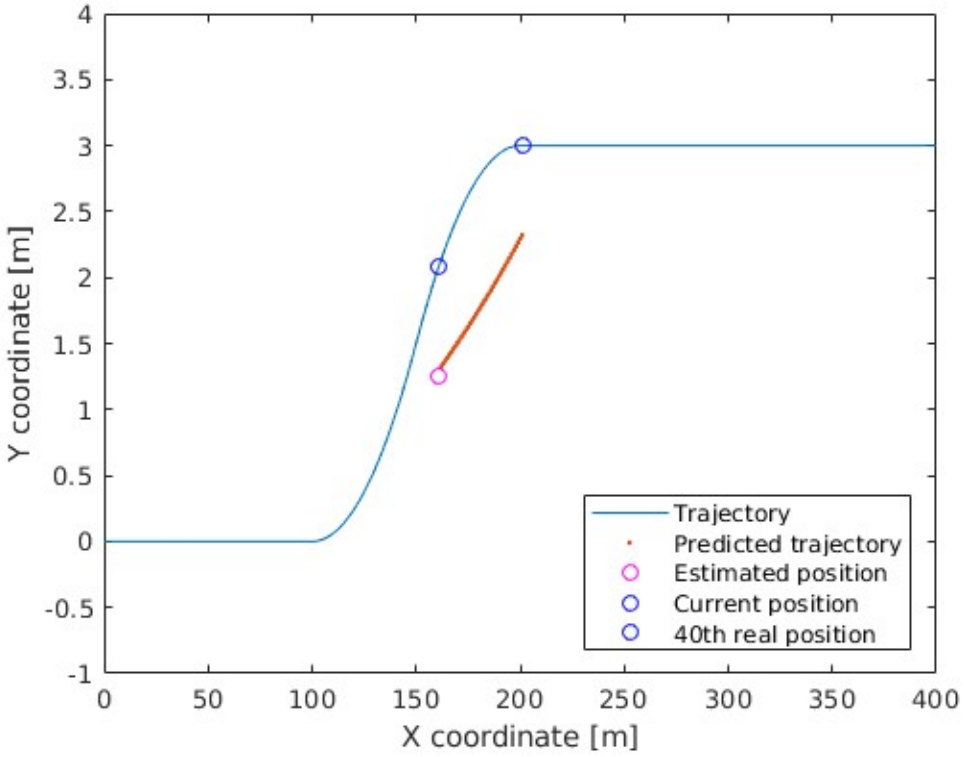

Fig. 62

4.2.3.8 Sinusoidal path

As can be seen in Fig. 63, the error happens to be very high, 50 cm for the x-axis and around 6 meters for the y-axis. This is due to the continuous change of acceleration during the path.

As can be seen from the simulation in Fig. 64, the error makes the predicted path diverge from the actual one, especially in the moments of drastic acceleration change. In the figure in particular it can be seen a moment in which the obstacle decelerates but the filter is still maintaining the same information about the acceleration it had before. This leads the actual object to be in a backward position with respect to the predicted one.

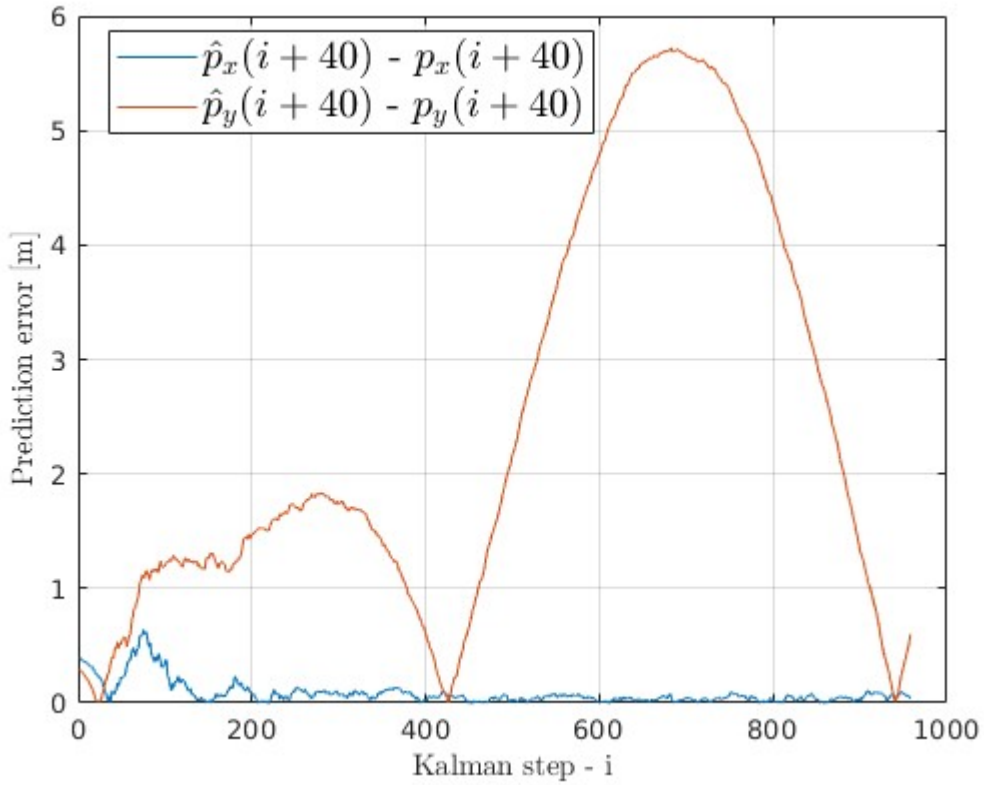
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
1
2
3
4
5
6
0
200
400
600
800
1000

Fig. 63

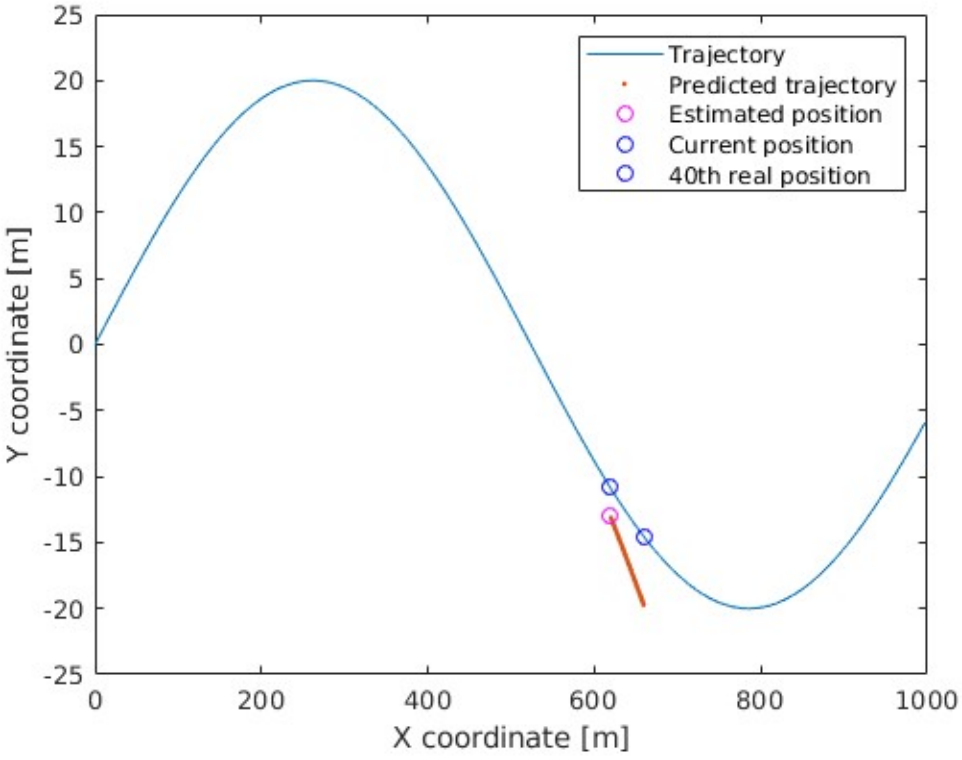
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
25
20
15
10
5
0
-5
-10
-15
-20
-25
0
200
400
600
800
1000

Fig. 64

4.2.3.9 MSE report

Here are reported the MSE for each axis in all scenarios:

Scenario	MSE_X	MSE_Y
Straight with constant speed	$2.69 \cdot 10^{-2}$	0.13
Straight with constant acceleration	1.63	$1.09 \cdot 10^{-2}$
Straight with constant deceleration	0.29	$2.88 \cdot 10^{-2}$
Straight with ramp acceleration	$2.09 \cdot 10^{1}$	0.13
Turning (as in road curve)	1.37	0.99
Circumference (as in roundabout)	$4.32 \cdot 10^{2}$	$2.87 \cdot 10^{2}$
Overtaking path	$1.96 \cdot 10^{-2}$	1.03
Sinusoidal path	$1.78 \cdot 10^{-2}$	$0.92 \cdot 10^{1}$

It is highlighted also by the MSE how the CA model is really accurate in predicting the trajectory of obstacles. The scenario in which it happens to be more imprecise is the "Circumference (as in roundabout)". That is the one that most of all results in an incorrect prediction due to the fact that non-constant acceleration is present on both axes.
What has been discovered from simulations anyway is that better performances are needed to have a good forecast, so another motion model is evaluated.

4.2.4 CJ (Constant Jerk)

This model considers to have a constant first derivative of the acceleration. It is not a standard motion model but it has been investigated and the results were really promising.

The state vector is:

$$\vec{x}(t) = \begin{pmatrix} x & y & v_x & v_y & a_x & a_y & j_x & j_y \end{pmatrix}^T$$

where $j_x = \frac{da_x}{dt}$ and $j_y = \frac{da_y}{dt}$.

The matrices of the system are:

$$A = \begin{bmatrix} 1 & 0 & dt & 0 & \frac{1}{2}dt^2 & 0 & \frac{1}{6}dt^3 & 0 \\ 0 & 1 & 0 & dt & 0 & \frac{1}{2}dt^2 & 0 & \frac{1}{6}dt^3 \\ 0 & 0 & 1 & 0 & dt & 0 & \frac{1}{2}dt^2 & 0 \\ 0 & 0 & 0 & 1 & 0 & dt & 0 & \frac{1}{2}dt^2 \\ 0 & 0 & 0 & 0 & 1 & 0 & dt & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & dt \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

4.2.4.1 Straight with constant speed

As can be seen in Fig. 65, the error goes high at the beginning but after the transient it stabilizes until it is bounded around 50 cm for both x and y axes.
The maximum difference between the predicted and the actual position reaches 4 meters for the x-axis and 2 meters for the y-axis. This result can be explained with the fact that the Kalman filter needs some time to adjust itself and to tune its parameters, but after the transient the error stabilizes around 0.

From the simulation in Fig. 66 it can be seen that the path is followed correctly.

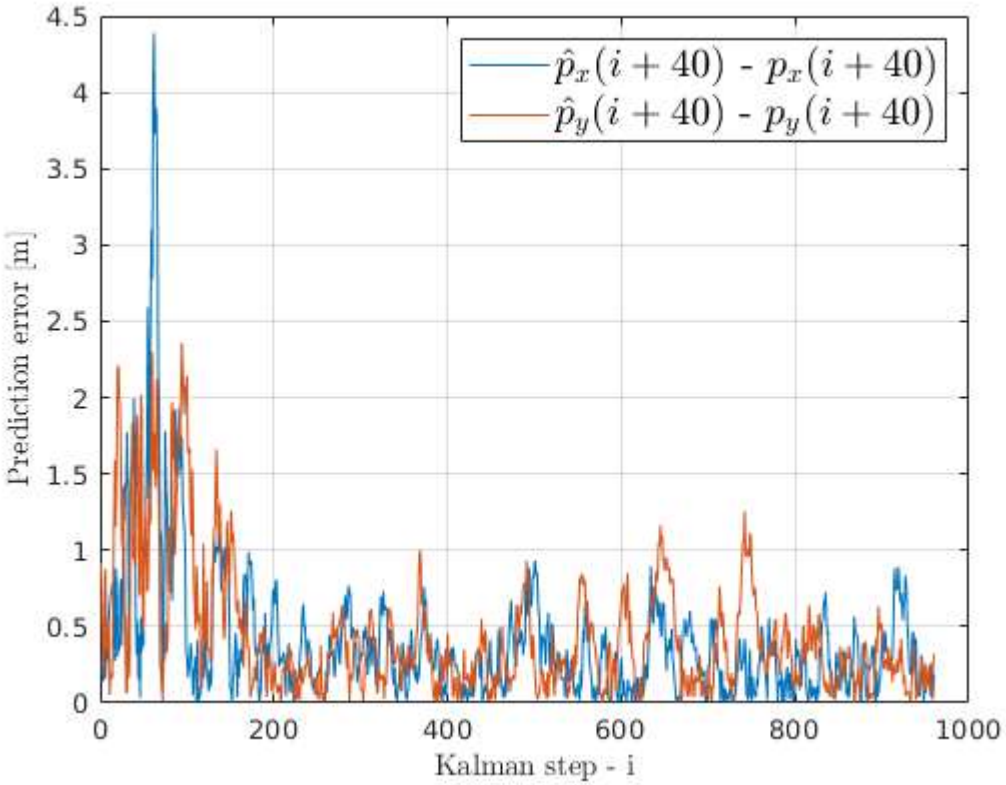
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
0
200
400
600
800
1000

Fig. 65

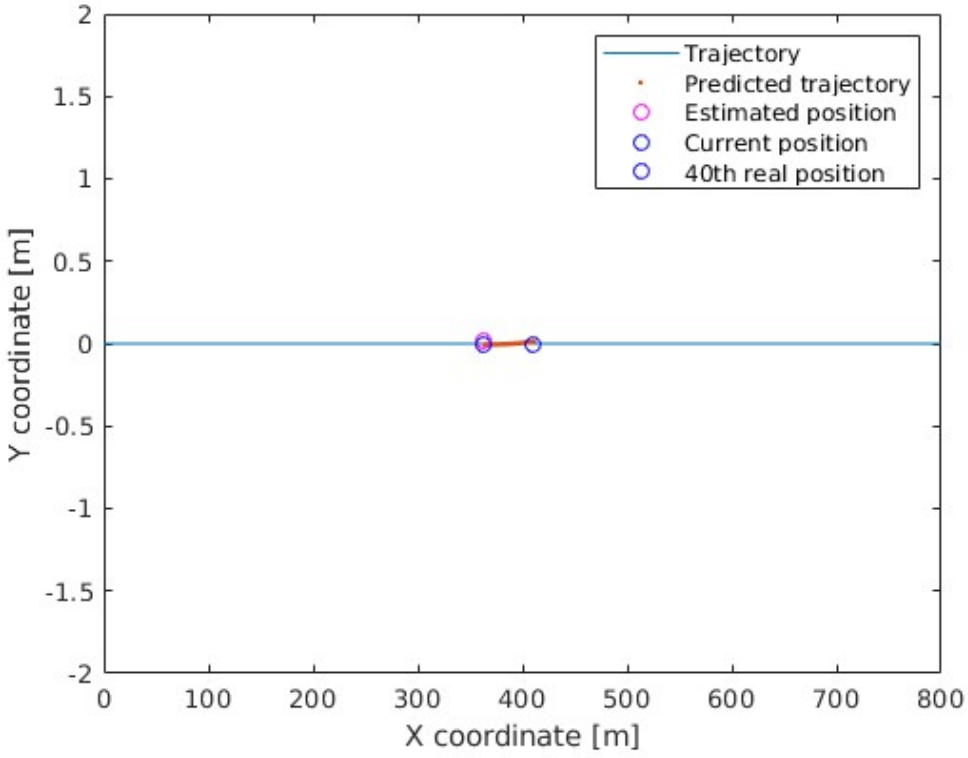
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
0
100
200
300
400
500
600
700
800

Fig. 66

4.2.4.2 Straight with constant acceleration

As can be seen in Fig. 67, the error follows the same path than in the previous case, high at the beginning and then around 0. Maximum values for the error around 4 meters.
The simulation gives the same result than before with the constant speed, meaning that a significant difference is not visually appreciable.

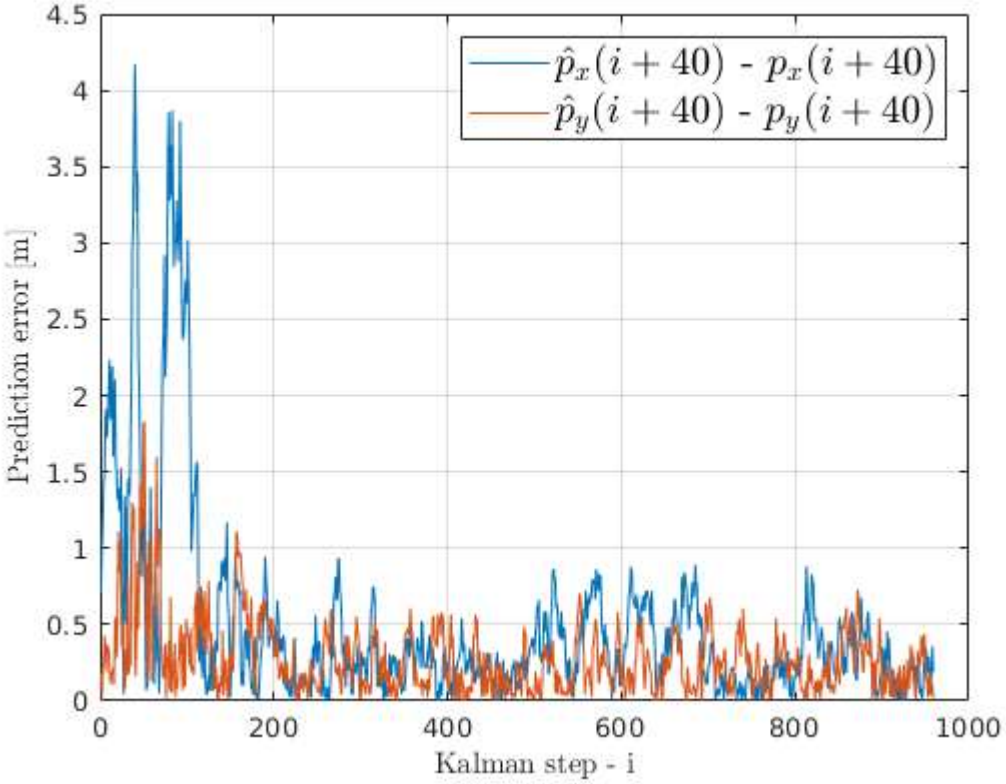

Fig. 67

4.2.4.3 Straight with constant deceleration

As expected, since this is the reciprocal of the previous case the behavior is the same.
Due to the fact that in this scenario the module of the deceleration is lower than the one of the acceleration, the values of the error change as shown in Fig. 68. The maximum value reaches 2.5 meters for the x-axis and 1.5 meters for the y-axis, but in the end both are bounded around 1 meter of difference between actual and predicted trajectory.

Also from the simulation nothing evident has been seen, both the constant acceleration and the constant deceleration case behave like the scenario with constant speed.

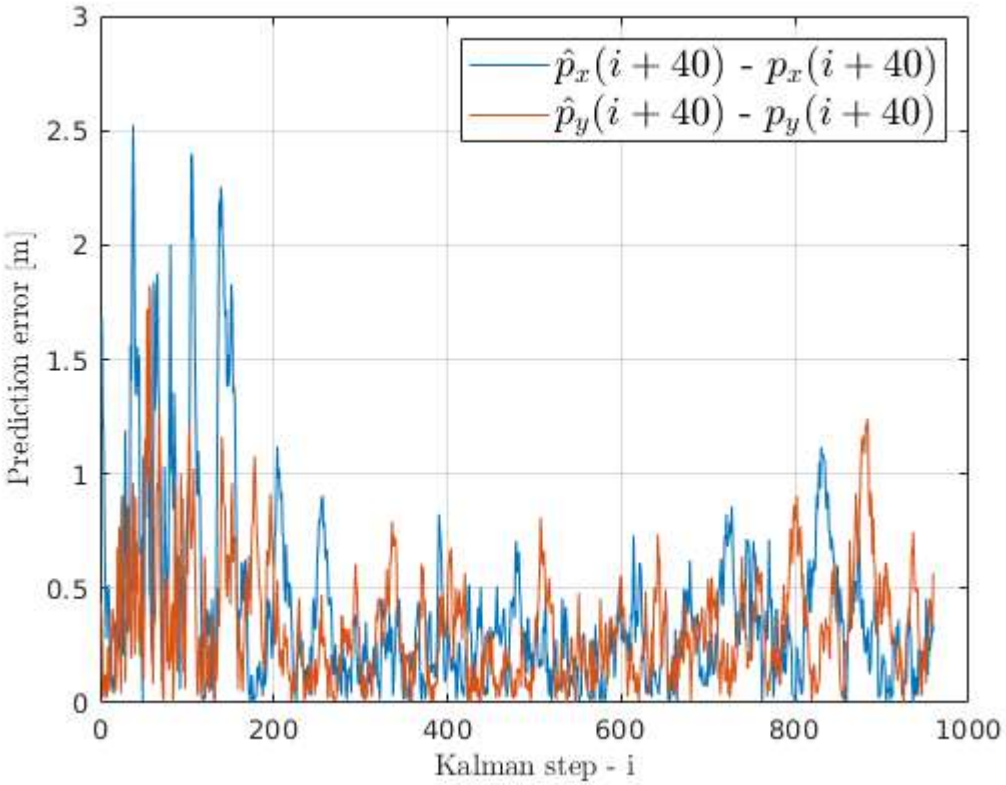

Fig. 68

4.2.4.4 Straight with ramp acceleration

This case is the first one in which the effect of the jerk can be seen. In the previous cases, having a constant speed or acceleration means that the jerk is 0 so basically not so much difference can be noticed with the CA model.
In this case it can be seen from Fig. 69 that the error is higher at the beginning but then it stabilizes around the 50 cm which is very good. This is the only model as far as of now that really predicts well the path, obviously after having the time to tune its parameters. This can be explained with the fact that a ramp acceleration has a constant jerk, so it is very well modelled with a constant jerk model.

From the simulation it can be seen that basically no difference is there between the actual and the predicted positions, just like the previous cases.

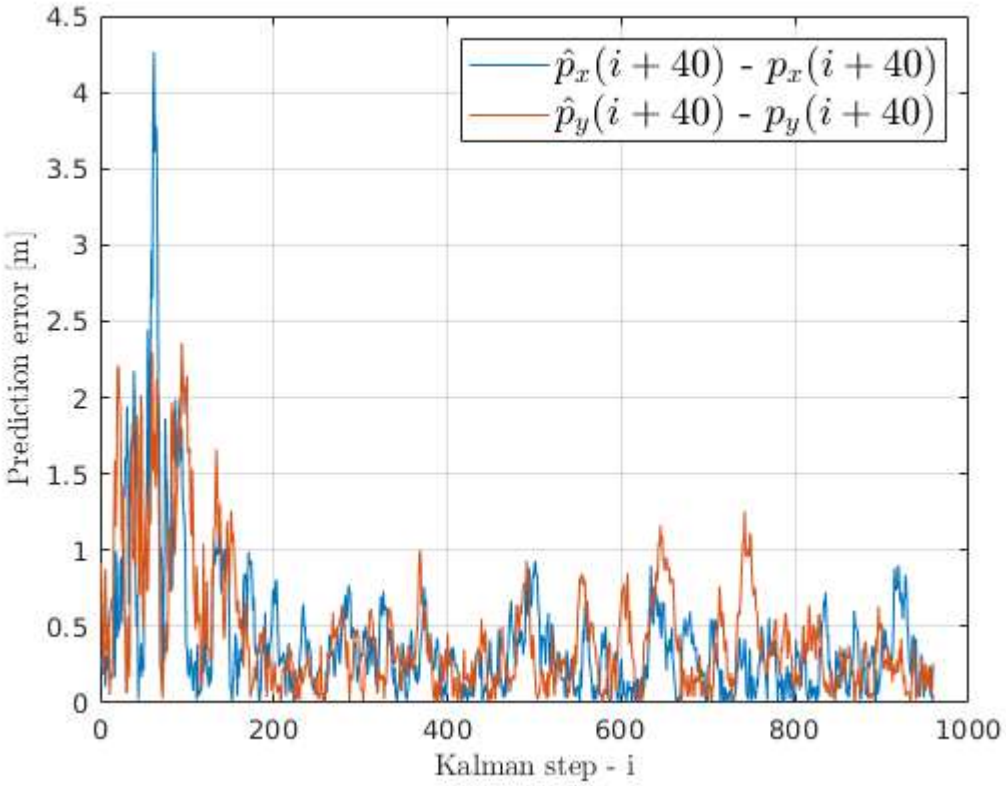

Fig. 69

4.2.4.5 Turning (as in road curve)

This scenario is very difficult to be handled. Even with this model, that so far has been showing very good results, it can be seen from Fig. 70 that the error goes very high, it reaches 3.5 meters for the x-axis and 2.5 meters for the y-axis, but after the transient it stays around 50 cm for both axes. This is really not a bad result with respect to the previously used models, in fact this model is capable to well predict the trajectory of the obstacles like the simulation shows.
In Fig. 71, it can be seen that the predicted path is following the actual one, they are coincident.

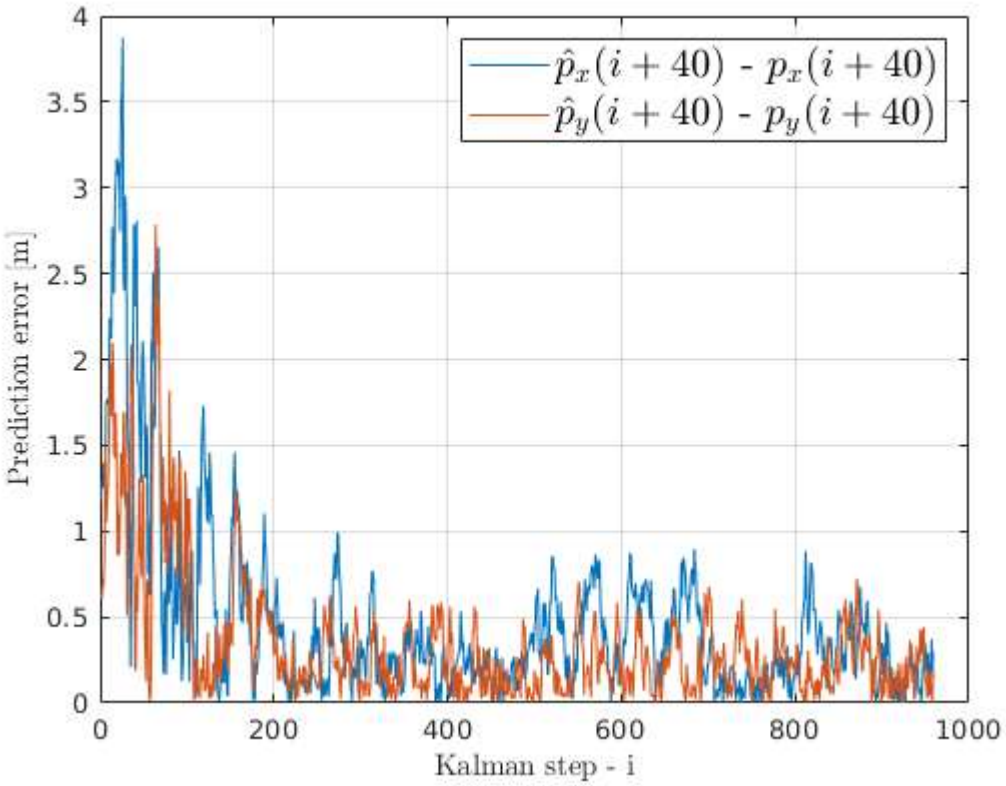
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
0.5
1
1.5
2
2.5
3
3.5
4
0
200
400
600
800
1000

Fig. 70

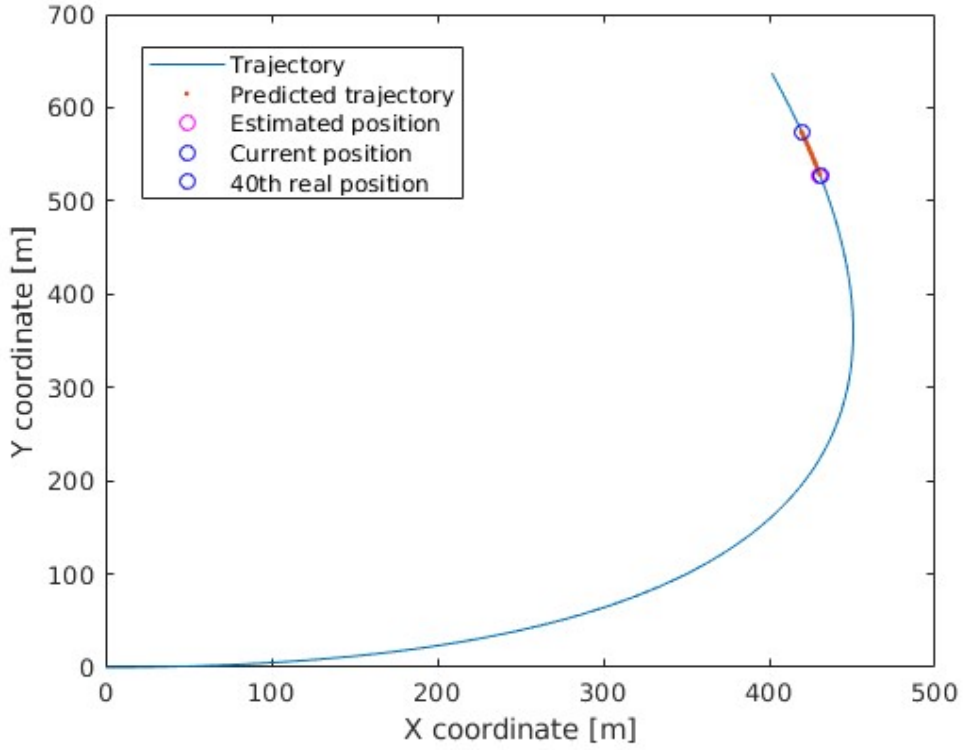
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
0
100
200
300
400
500
600
700
0
100
200
300
400
500

Fig. 71

4.2.4.6 Circumference (as in roundabout)

This is another scenario that is pretty difficult to be followed. It can be seen from Fig. 72 anyway that after a transient, in which the error reaches 5 meters of difference from the actual position in the x-axis, the error stabilizes around 2 meters. It is an optimal result with respect to the ones of the previous models that had a sinusoidal profile.

As can be seen in Fig. 73, the actual path and the predicted one, along the way, have no relevant differences.

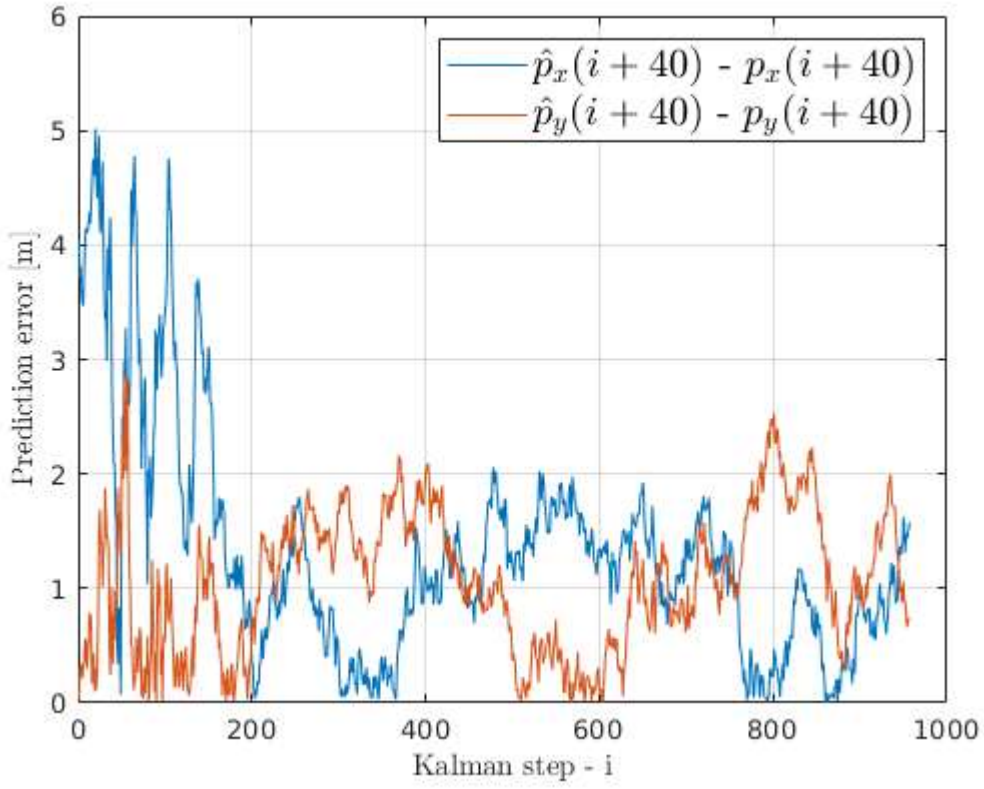

Fig. 72

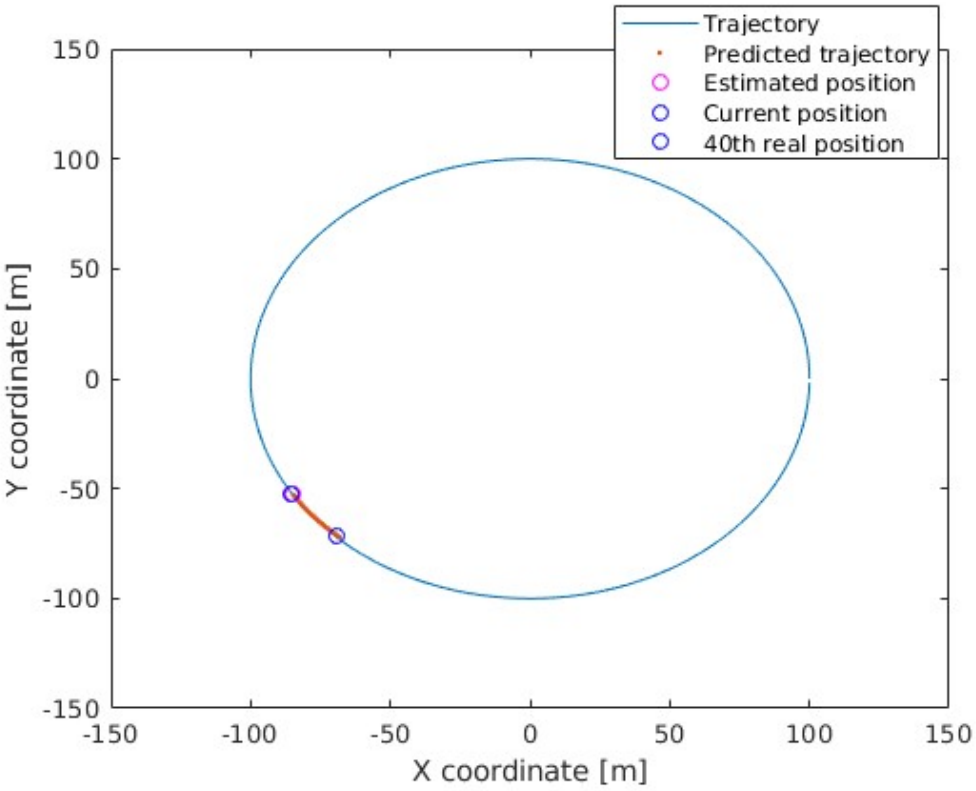

Fig. 73

4.2.4.7 Overtaking path

As can be seen in Fig. 74, after the transient, the error reaches 20-25 cm. During the transient the maximum error go to 1/1.5 meters for the x-axis and 4.5 meters for the y-axis. This is an unprecedented result, this model is capable of handling a changing acceleration: in the moment in which the obstacle is effectively doing the overtaking, instant in which the acceleration varies also not regularly, the error happens to be a little higher but then it stabilizes around 0.

In Fig. 75, it can be seen from the simulation that the actual path and the predicted one, along the way, have no relevant differences. There are situations in which the actual object is in a backward position with respect to the predicted one but the difference decreases with time.

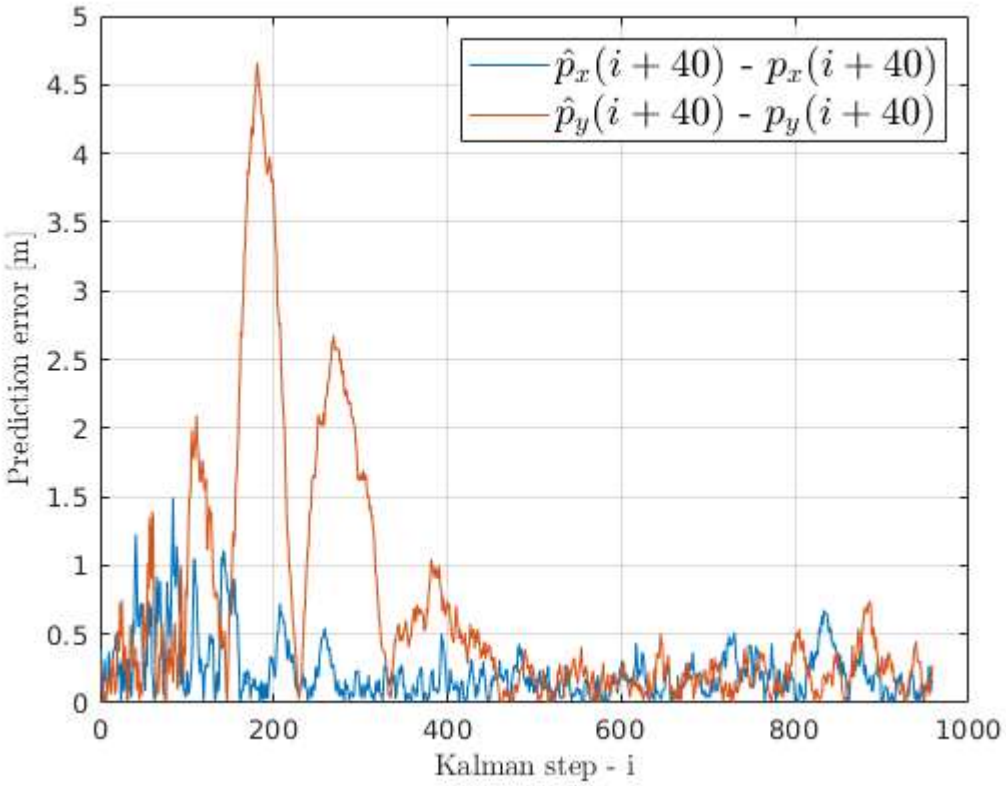
$\hat{p}_x(i+40)$ - $p_x(i+40)$
$\hat{p}_y(i+40)$ - $p_y(i+40)$
Prediction error [m]
Kalman step - i
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
200
400
600
800
1000

Fig. 74

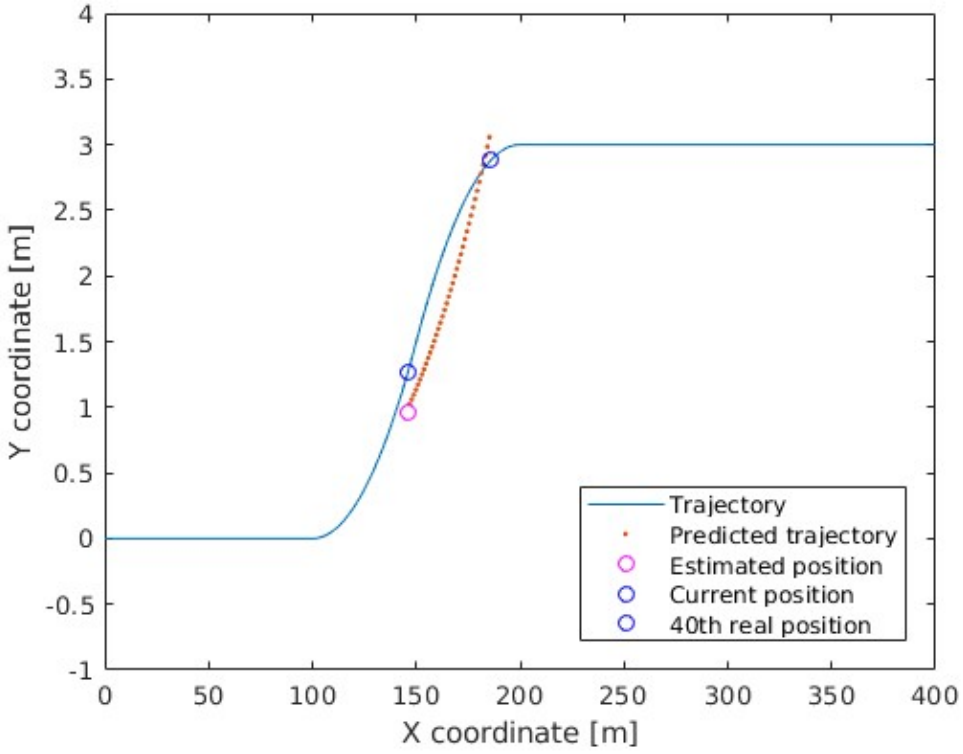
Trajectory
Predicted trajectory
Estimated position
Current position
40th real position
Y coordinate [m]
X coordinate [m]
-1
-0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
0
50
100
150
200
250
300
350
400

Fig. 75

4.2.4.8 Sinusoidal path

As can be seen in Fig. 76, the error for the x-axis reaches a maximum of 1 meter while for the y-axis it reaches a maximum of 2 meters but then they both stabilize around 20/25 cm. This means that after the transient, the jerk model is capable of predict a sinusoidal path with very high precision. None of the previously analyzed models has been capable to do so and this is visible also from the simulation.

As can be seen from Fig. 77, the predicted path never diverges from the actual path. Along the way it is always possible to follow and predict properly the actual trajectory the object is taking.

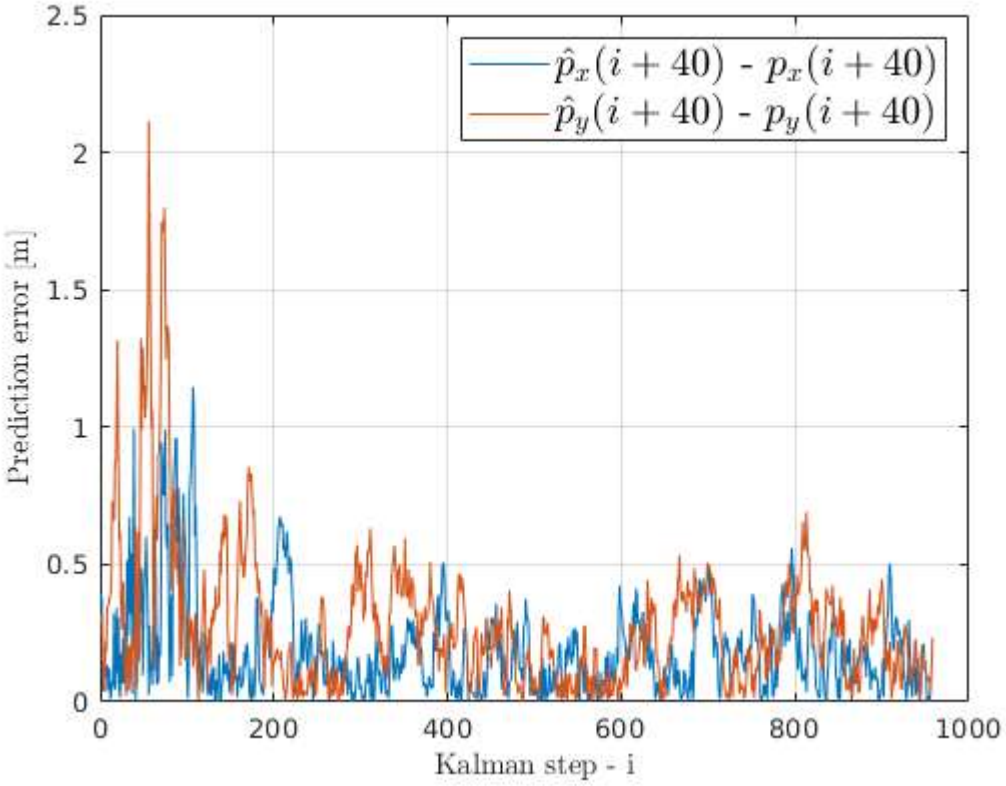

Fig. 76

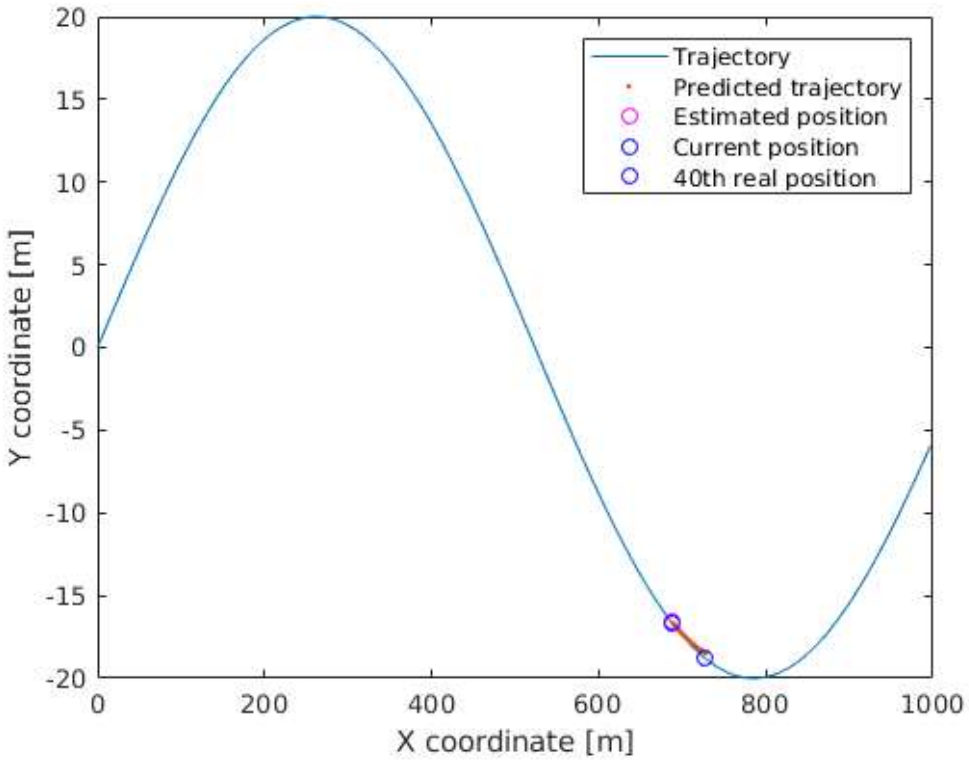

Fig. 77

4.2.4.9 MSE report

Here are reported the MSE for each axis in all scenarios:

Scenario	$\boldsymbol{MSE_X}$	MSE_Y
Straight with constant speed	0.37	0.37
Straight with constant acceleration	0.65	0.13
Straight with constant deceleration	0.32	0.18
Straight with ramp acceleration	0.37	0.37
Turning (as in road curve)	0.52	0.27
Circumference (as in roundabout)	2.71	1.47
Overtaking path	$0.94 \cdot 10^{-1}$	1.29
Sinusoidal path	$0.64 \cdot 10^{-1}$	0.14

It is highlighted also by the MSE how the constant jerk model predicts well the path in all scenarios. The MSE is bounded for all of them.

4.2.5 Chosen motion model

After all the simulations done, it has been chosen the **constant jerk** model. It has the best performances for what regards all possible scenarios that has been studied, as can be shown by the simulations and by the MSE reports.
The model has demonstrated to follow well profiles in which acceleration changes, both linearly and not and so it is well suited for everyday life scenarios.

Here a table with all the scenarios and the information about if the related model has performances high enough to say if the actual path has been followed or not:

- ✓ : means that the model can predict well the path defined in the scenario
- ✗ : means that the model is not capable to predict well the actual path of the scenario

Scenario	CV model	CA model	CJ model
Straight with constant speed	✓	✓	✓
Straight with constant acceleration	✗	✓	✓
Straight with constant deceleration	✗	✓	✓
Straight with ramp acceleration	✗	✗	✓
Turning (as in road curve)	✗	✓	✓
Circumference (as in roundabout)	✗	✗	✓
Overtaking path	✗	✗	✓
Sinusoidal path	✗	✗	✓

4.3 Pathfinding algorithm

Before considering the local motion planning algorithm, it is important to evaluate which global planning algorithm has been used. The choice is the A* algorithm but it has been studied other algorithms that could give better performances. A possible future work could be to implement another one of them.

A*, as expressed before, is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it explores the tree of possible configurations thus leading to the choice of the shortest path to reach the destination.

The A* algorithm works fine in static environment. In dynamic environment it does not because, once the car has decided for a path, it does not change its mind so, if an obstacle happens to be in the middle of the path, there would be a crash.

A solution to this has been to call A* every few milliseconds to re-calculate the path every time. This can solve the collision avoidance problem but still it is computationally heavy because every time the algorithm has to forget about what it decided before and, considering all the new information from the obstacles, evaluate another path. This basically means to erase all previous knowledge to obtain a new one.

What could be done instead could be to use a motion planner based on A* that is capable to update the path to reach the goal only when it is necessary, as when a new obstacle happens to be on the map or a previously recognized one is moving towards the ego-vehicle.

To be noticed that it is not used the A* algorithm that the platform already had. It was implemented in a way such that it considered already the non-holonomicity of the car.
In this thesis the non-holonomicity of the vehicle has been handled by the motion planner algorithm.

4.4 Motion planning

To do motion planning, that also provides collision avoidance, a few approaches could be considered. Not all of them have been used but this could be a possible future work.
The work of this thesis has been to implement the DWA algorithm. Anyway, it has been discovered in simulation that the algorithm does not work in specific scenarios, that still are very important for a real car in a real environment: it has been discovered in fact that the algorithm is suitable for vehicles with turning radius equal to 0, thus not representing the car model: in fact the car is not supposed to turn on itself. It has been decided so to evaluate also the performance of another algorithm, the TEB motion planner.

The **dwa_local_planner** package in ROS provides a controller that drives a mobile base in the plane. Using a map, the planner creates a kinematic trajectory for the robot to get from a

start to a goal location. Along the way, the planner creates, at least locally around the robot, a value function, represented as a grid map. This value function encodes the costs of traversing through the grid cells. The controller's job is to use this value function to determine dx, dy, dθ velocities to send to the robot [18].

The basic idea of the DWA algorithm implemented in this package is as follows:

1. Discretely sample in the robot's control space (dx,dy,dθ)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

On the other side, the **teb_local_planner** package implements an online optimal local trajectory planner for navigation and control of mobile robots as a plugin for the ROS navigation package. The initial trajectory generated by a global planner is optimized during runtime minimizing the trajectory execution time (time-optimal objective), separation from obstacles and compliance with kinodynamic constraints such as satisfying maximum velocities and accelerations.
The optimal trajectory is efficiently obtained by solving a multi-objective optimization problem and letting the user decide the weights of the different constraints the car should follow [19].

4.4.1 Simulation with DWA: no obstacles

Dwa algorithm has been applied in simulation, using an environment built by the developers of the platform this work starts from. Better details in section 4.4.4.
The first implementation sees the car going on its own, without both static and dynamic obstacles. In Fig. 78, it can be seen a snapshot of the ego-vehicle inside the simulator.

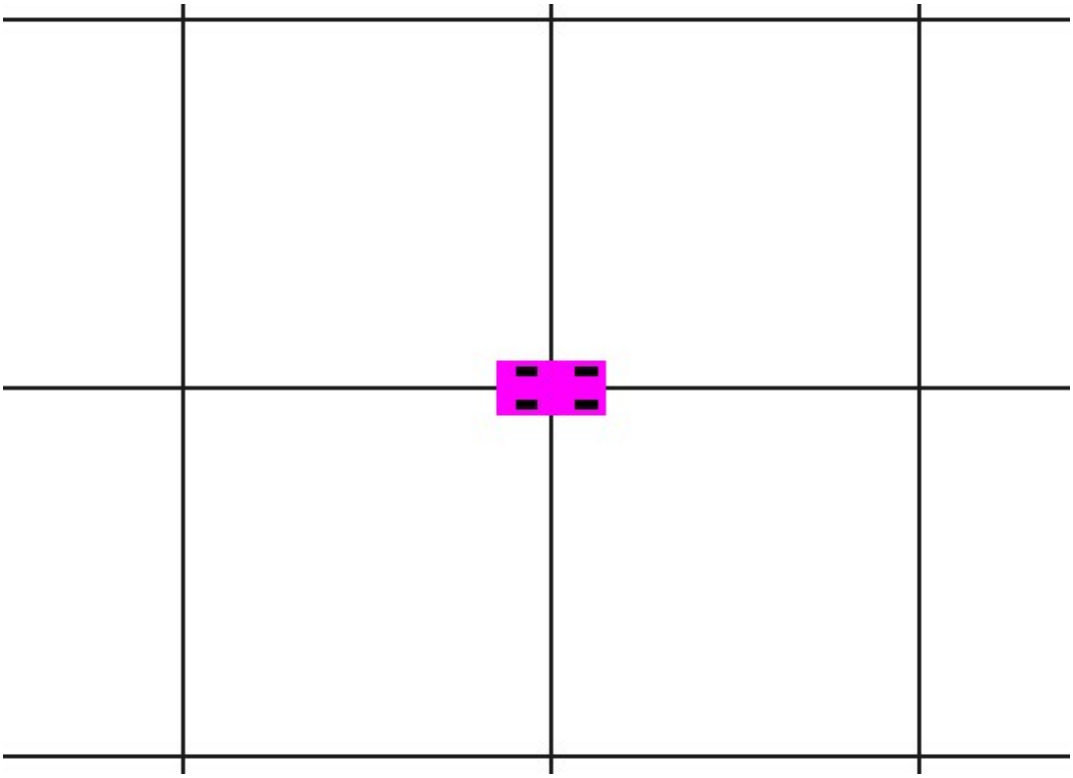

Fig. 78: this is what can be seen in the simulator: the colored rectangle is the ego-vehicle; it moves inside a grid that represents the free space that surrounds the car.

What has been shown by the simulation is a car that evaluates the best path, the shortest to reach the destination, and then follows it.

For what regards the global path, it follows the one that has been decided from the beginning due to the fact that no obstacles are present so there is nothing that can hinder the path. It evaluates the path for a moving dot so no maneuvers are taken into account.

The local path tends to adjust itself, due also to the fact that the car is non-holonomic so no velocity along the y-axis is considerable. In this situations the local path follows a different trajectory with respect to the global path as can be shown in Fig. 79. The local path is composed of a bundle of lines because at each step a new local path is computed depending on the actual position of the ego-vehicle.

Fig. 79: The ego-vehicle is represented by the blue polygon visible on the right. The purple line is the global path and the green lines the local path. As the goal is on the opposite direction on motion of the ego-vehicle, it has to find a way to reach it. Since it is not a simple moving dot but it has dynamic constraints, the local path consists in a curve that tries to put the ego-vehicle on the global path.

One thing that can be noticed is that sometimes the car is moving around the goal trying to reach exactly the position, sometimes turning unnecessarily. This is easily solved by softening the constraints on the destination, ie by saying that even being in the proximity of the goal could be considered good.

4.4.2 Simulation with DWA: static obstacles

Avoiding static obstacles is what the developers of the platform already achieved **but** they used only a path planning algorithm; in this thesis it has been introduced also a motion planning so the goal here is to understand if everything works fine as before.

In general, the dwa algorithm has shown to well evaluate a path to reach the destination.
For what regards the global path no relevant difference can be found with the work previously developed on the platform: this is a consequence of the fact that the global path algorithm is founded on A*. If nothing dynamically happens to be in the middle of the path of the ego-vehicle, the global path follows the one that has been decided from the beginning.

The local path tends to adjust itself, due also to the fact that dynamic constraints have been introduced, just like the no obstacle scenario.

When the dynamic constraints are no longer a problem for the motion, what can be seen is that local and global path overlaps, as it is shown in Fig. 80. Also the local path is composed no more of a bundle of lines but of a single green line because the direction of motion has become stable.

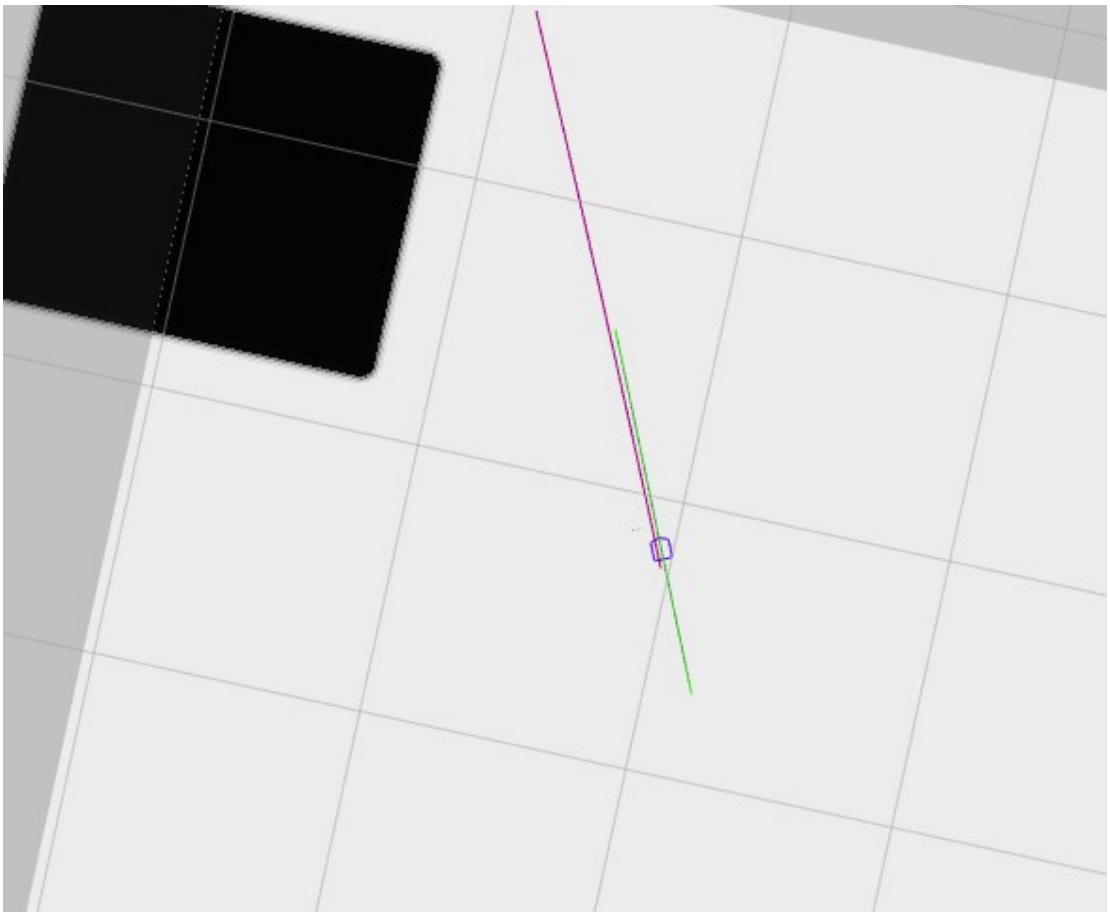

Fig. 80: the blue polygon represents the ego-vehicle. Local path (green line) and global path (purple line) almost overlap.

To test the effective capabilities of the ego-vehicle to avoid static obstacles, a scenario has been built consisting in a set of rectangles, representing for example buildings or parked cars, put in a way that the car is forced to follow a sinusoidal profile of motion between them.
As can be seen from Fig. 81, the global path happens to be very next to the obstacles, this is due to the fact that the A* algorithm has to consider the shortest possible path to reach the goal and it has no information about the dimensions of the car.
The local path also is very interesting because it can be noticed that is composed of a tree of choices that represent all the local trajectories. As the car is following a curve, the local direction of motion changes moment by moment.

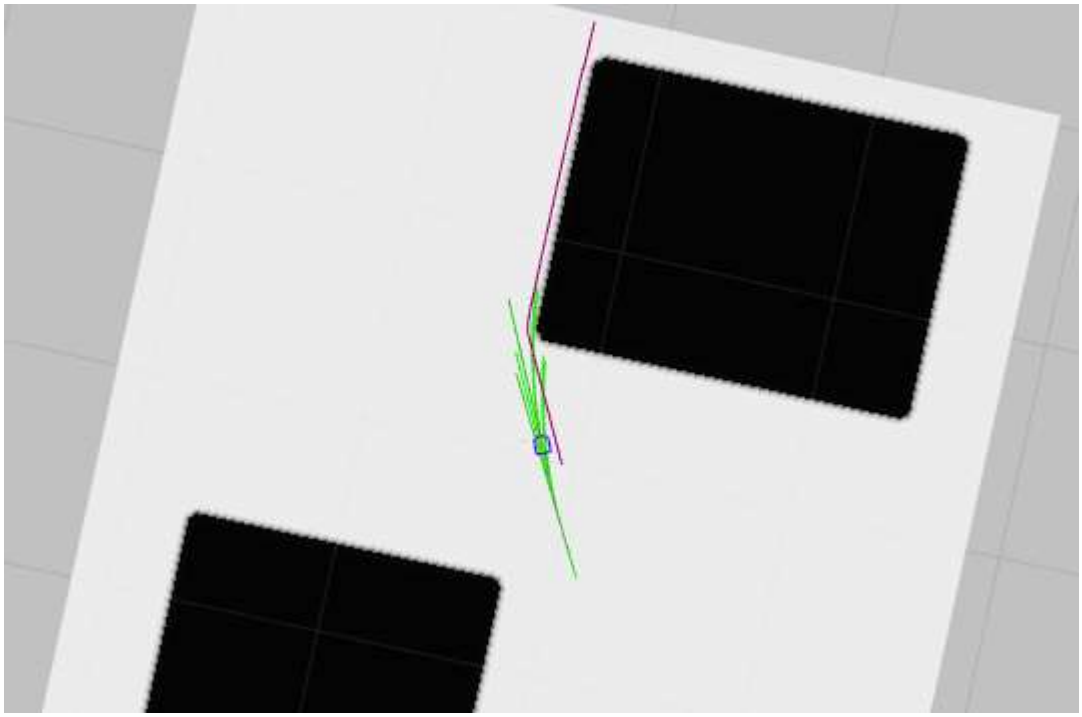

Fig. 81

Anyway a particular scenario has got the highest attention: if a goal is chosen such that it has a yaw angle perfectly perpendicular to the orientation of the footprint of the ego-vehicle, the car results to be stuck:

- if the maximum value for the velocity on y-axis is different from 0, the dwa continues to give commands for a speed along that axis but the car is subject to non-holonomicity constraints so a movement with only a component on y-axis is not acceptable. To solve this, it has been chosen to set the maximum acceptable velocity along the y-axis to 0.
- If it is 0, the result is that anyway the car is stuck because the dwa evaluates as best path the one that considers only a turn around the z-axis: that also for car-like robots is not possible because the turning radius is not 0.

With this particular scenario it has been discovered that the dwa algorithm is capable to handle non-holonomic robots **but only** the ones with turning radius equal to 0, for example two-wheeled robots. Car-like robots have information for turning that are not handled.
In Fig. 82 it is shown the stuck car.

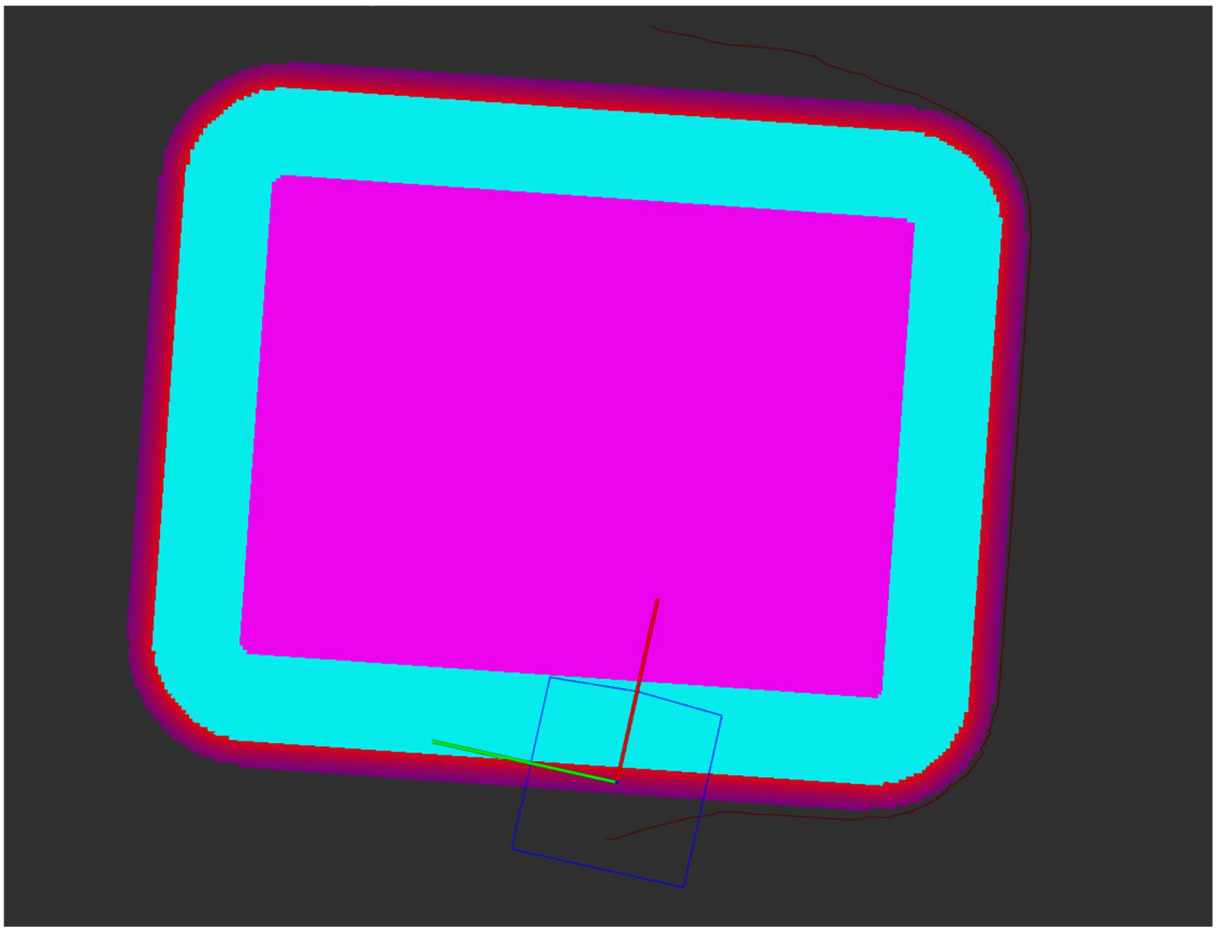

Fig. 82: the pink square is the obstacle. The light blue and red areas that are around the obstacle are the result of the inflation. The blue polygon is the ego-vehicle.

It can be noticed that the car could reach any other configuration if only it was capable to understand how to do the necessary maneuvers.
This has led to the implementation of another motion planner, the TEB local planner.

4.4.3 Simulation with TEB: static obstacles

The first problem that has been faced in the implementation of this algorithm is that if a goal is chosen a little bit far away from the actual position of the ego-vehicle, it starts to follow the path towards it but at a certain point it fails and no valid path and speed commands are produced. This has been explained with the fact that the algorithm is computationally heavy and the high resolution of the costmap does not help. At a certain point the algorithm results to be infeasible and no solution is provided.
The solution has been to reduce the resolution, thus increasing the size of a cell in the costmap, to lighten up the algorithm: the resulting behavior has been as expected, the ego-vehicle is capable to take long paths to the destinations.

The algorithm for the global planner is not changed, it is always based on A*: in effect, the teb is a local planner and so its performances can only be seen there.

For the local plan, the general behavior is the same as seen with the dwa, with the difference that, with the teb, the car is doing maneuvers: after setting the same situation that caused the stuck car problem with the dwa, the result is that the car is now capable to do all the movements needed (that is a little movement backward and then forward to the destination).

Having understood that in a static situation the vehicle is behaving well, it is time to take into account also dynamic obstacles.

4.4.4 Rust environment upgrade for simulation

The environment built for the simulation has been developed in Rust: Rust is a multi-paradigm systems programming language focused on safety, especially safe concurrency. It is syntactically similar to C++, but is designed to provide better memory safety while maintaining high performance. Rust's rich type system and ownership model guarantee also thread-safety — and enable developers to eliminate many classes of bugs at compile-time.

The environment already consisted in a grid map on which the ego-vehicle and randomly generated obstacles can run, as can be seen in Fig. 83.
What has been done is that a scenario has been built inside this environment. A crossroad has been sketched and put as base of the grid map, then cars are automatically generated as traversing the crossroad, as shown in Fig. 84.

When the simulation starts, so do cars too: this is however problematic because after a while the vehicles will no longer be in the field of action the ego-vehicle is interested in.
So an upgrade has been implemented: whenever a specific button of the keyboard is pressed a car is generated: this car goes back and forth passing every time for the crossroad. Then to simulate a busy road, it has been established that, by pressing another button, another machine is generated that moves anyway back and forth but in a manner opposite to the previously generated one as can be seen in Fig. 85.

Within this scenario the TEB motion planning algorithm has been studied with and without the trajectory prediction of road users to evaluate performances.

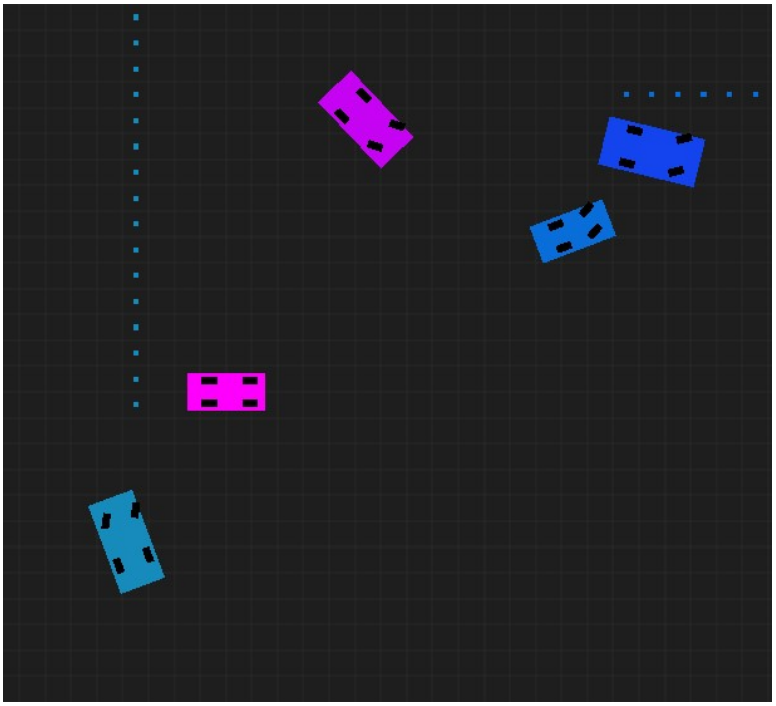

Fig. 83: the pink car is the ego-vehicle. The others are randomly generated.

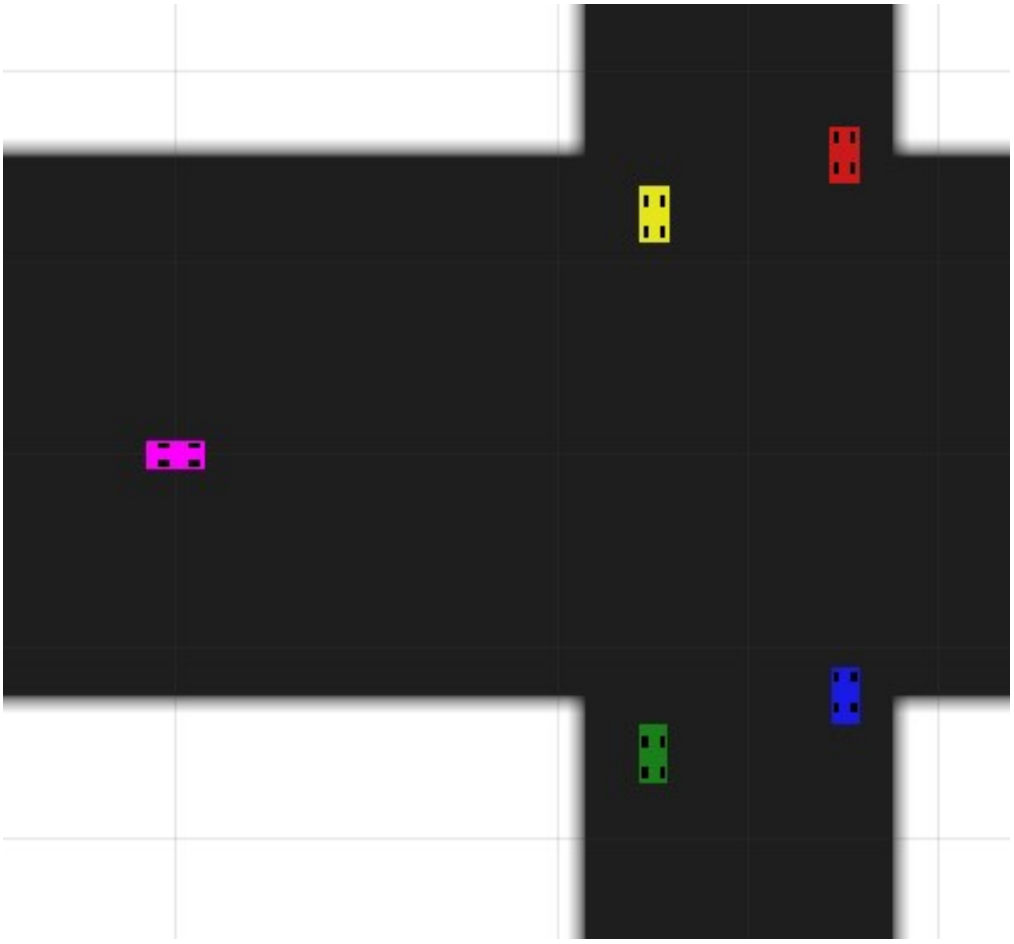

Fig. 84: Scenario built for the simulation of dynamic obstacles

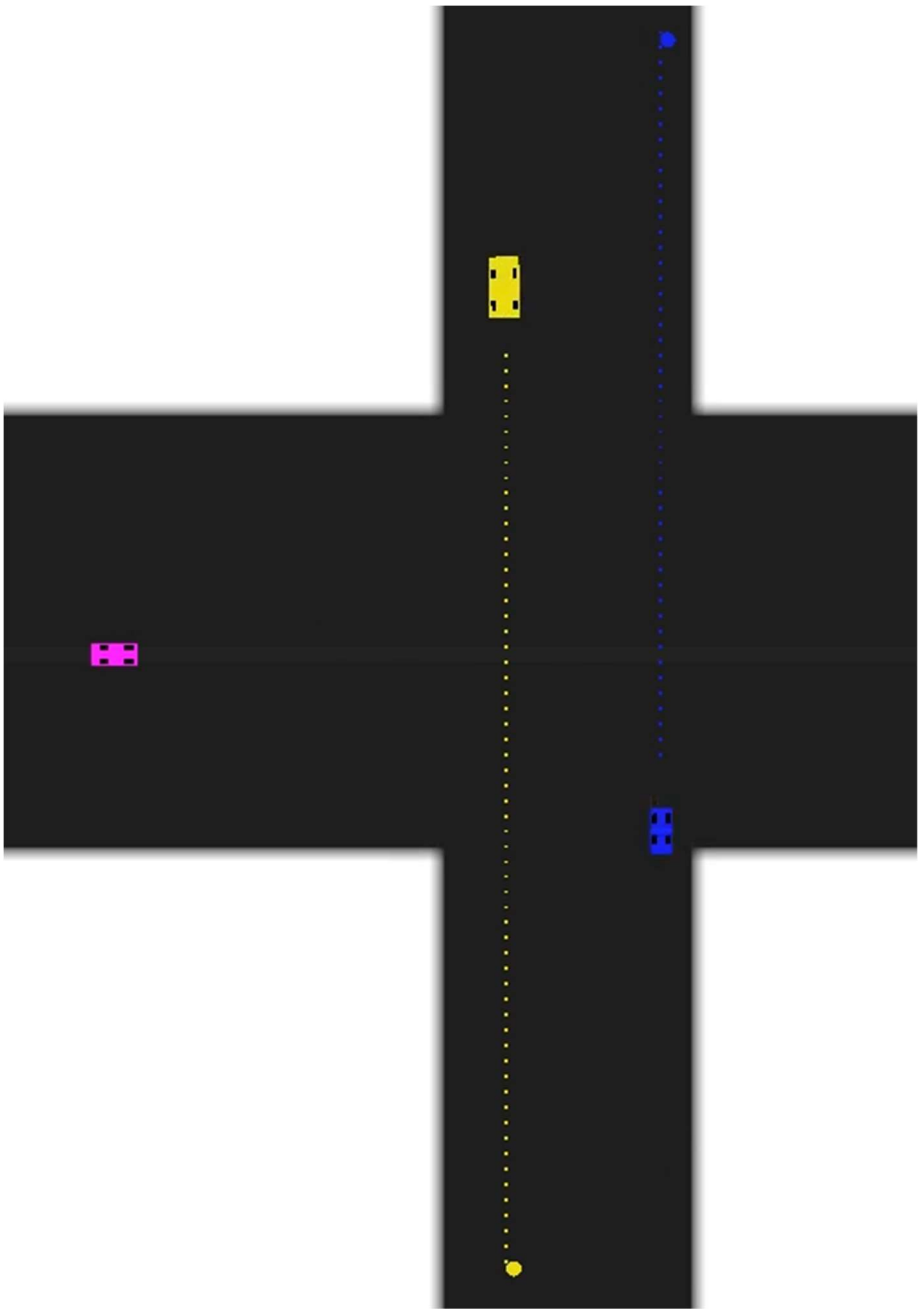

Fig. 85: Scenario built to simulate a continuously busy road: the yellow and the blue car go back and forth. The small dots represent the path the car have to follow, the bigger dot is the temporary destination. This last one will change any time the vehicles reach it.

4.4.5 Simulation with TEB: dynamic obstacles, no trajectory prediction

The local planner establishes a trajectory and speed commands within a square of the costmap defined effectively “local costmap”.
The ego-vehicle has information about other road users very far away from it, due to the fact that it is equipped with Ibeo sensors, so it could be decided to set this local costmap to cover a great area that surrounds the vehicle: in this way every car that appears is mapped. However the main drawbacks are the following:

- the algorithm becomes computationally heavy and at some point the motion planner becomes infeasible and do not produce valid paths and speed commands
- a lot of information regarding cars far away, that will not interfere with the movement of the ego-vehicle, have to be discarded, have not to be considered for the movement, thus leading to the collection of a lot of data that are not used
- sometimes the ego vehicle recognizes that another car is going to pass over the path that it has to follow, so the effect is that it produces a change of the planned path; the problem is that it happens that the actually recognized car is really far away and it will transit on the planned path but then it will go away, thus not being a problem for the ego-vehicle. In the meanwhile anyway the ego-vehicle has already changed the path, then it will have to change again when the obstacle will move away. With multiple obstacles also, what can be observed is the ego-vehicle that continuously changes the path, thus producing a very slow movement with frequent stops and re-evaluations of the path.

On the other hand having a very small local map has shown to be very dangerous: the ego vehicle adjusts its speed commands only in the local area, as said before, so if nothing appears on the local map it evaluates that the maximum speed can be reached; whenever an obstacle is mapped in the local area is already too late and the speed command passes directly to 0: the result is that the car suddenly stops, thus there is not so much consideration for safety and comfort. To be noticed that this reasoning applies both for applications without trajectory prediction both for applications with it.

It has been evaluated a middle way that can be considered good with the limited speed of the ego-vehicle and that can work in basic scenarios. Anyway a possible future work could be to evaluate a changing costmap, with dimensions that dynamically change with the speed of the ego-vehicle: in fact, it the car is going slowly, it is not necessary to have information so much far away because a sudden stop will not cause problems. On the contrary, if the car is going fast, a sudden stop will cause a lot of issues also for the possible passengers, so a greater area would be preferable.
Once taken that into consideration, in this particular test suite, the goal is to have a look at the performances of the algorithm without the trajectory prediction of the other road users.
What can be noticed is that the ego-vehicle recognizes new obstacles when it is really in the proximity of these ones, if not right in front of them.

What can be seen in Fig. 86 is effectively a situation in which a crash verifies:

- the red line is the global path that leads to the target destination
- the blue square represents the ego-vehicle
- the black square is an obstacle, another car, passing in the environment

The ego-vehicle does not care about the presence of the other car because it is not yet upon the global path nor upon the local path. It decides then to continue its movement and to traverse the path of its obstacle.
The result is that the obstacle sees its way interrupted but it is too late to react. Then, the crash.

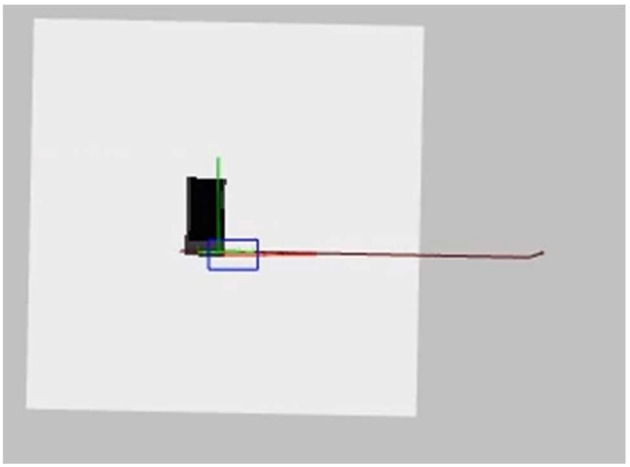

Fig. 86

4.4.6 Simulation with TEB: dynamic obstacles, trajectory prediction

In this scenario the trajectory prediction comes in the game.
What can be seen is that if the ego-vehicle comes in the proximity of another vehicle it is capable of re-arrange another trajectory without crashing.
The trajectory passes upon the global path so the ego-vehicle has the time to foresee the future possible crash and to evaluate another way in order to avoid it.

The decision process of the reaction is the following:

- Fig.87: The first thing the vehicle does is to stop or in any way to reduce the speed; the re-evaluation of the path begins;
- Fig. 88: If in the meanwhile the obstacle has gone away, the ego-vehicle chooses the old computed path if it is still the best one; if not, another one is selected.

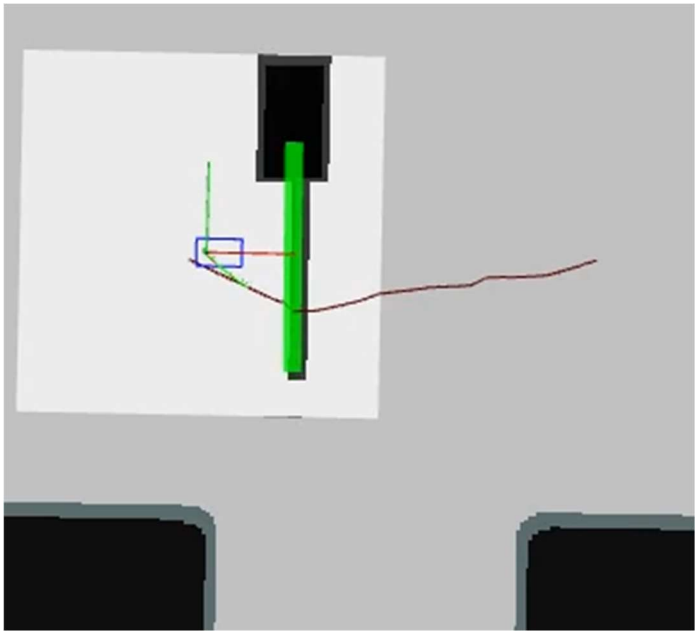

Fig. 87: the green line is the predicted path of the obstacle

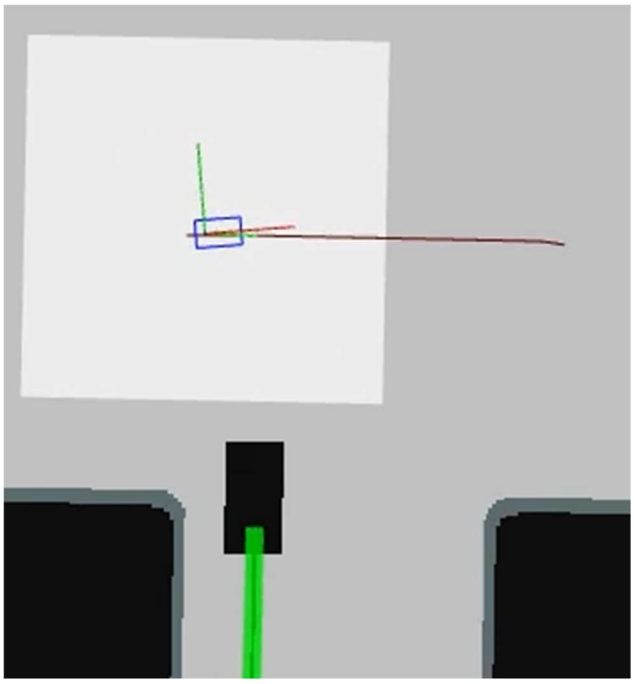

Fig. 88

Something has been noticed anyway: it happens that the ego-vehicle re-computes too many times the path to reach the destination: it happens, for example, that very fast cars pass upon the path but in a moment they are already far away. The algorithm recalculates the path, even if it is not necessary anymore. This causes frequent re-calculations and, due to the fact that a single evaluation is heavy on its own, big problems for the feasibility of the algorithm. In effect what has been seen is that it fails to produce a path and the vehicle is seen following random behaviors.
To solve this, it has been decided to decrease the controller frequency that is the rate in Hz at which to run the control loop of the global plan and send velocity commands to the base.

5. Conclusions

The focus of this thesis has been to implement a dynamic motion planning algorithm for an automated driving system in order to control continuously a vehicle from a start to a goal position avoiding collisions with known obstacles, both static and dynamic (people on the streets, other cars, bikes and so on).

The key points of this thesis have been the following:

- The inputs of the process come from the sensors; their task is to collect as much information as possible on the surrounding environment. Then the information from sensors is mapped inside a costmap. Cells corresponding to obstacles, both static and dynamic, are marked as *not-traversable.*

  Due to the fact that the costmap is a map of the environment divided in cells which have a cost information, the not-traversability can be done setting a high cost for a group of cells.
  - The software can handle three levels of **cost**: each cell can be either free, occupied, or unknown. When a cell is recognized as occupied, a “lethal obstacle” cost is assigned to it.
  - The **group of cells** represents the actual obstacle that is in the surroundings of the ego-vehicle. The sensors are capable to evaluate width and length of the obstacles, then the costmap is capable to compute the cells that have to be taken into consideration for those specific dimensions.

- A trajectory prediction algorithm forecasts the future positions that the moving objects are about to take. This information is also mapped inside the costmap. The prediction is done using a Kalman filter and applying a specific motion model.

  - The motion model effectively models the system of the actual moving obstacle. *Linear motion models* have been chosen that assume a constant velocity (CV) or a constant acceleration (CA). Their major advantage is the linearity of the state transition equation which allows an optimal propagation of the state probability distribution.
    Despite of these properties, however, these models have proven to not handle all the possible profiles of motion, in some cases also having a significant error when used within the Kalman filter to predict future positions of objects.
    Another model then has been identified: the constant jerk (CJ) model (the jerk is the derivative of the acceleration) has shown to have the best performances among the considered models, and, once applied within the Kalman filter, it has proven to well predict the trajectories of the obstacles.

  - The result of these predictions is that, in addition to information from the sensors, every obstacle in the surroundings of the ego-vehicle has also its predicted path. For each object this information is displayed on the costmap marking as occupied also the cells corresponding to the future positions of obstacle.
    To have then a better visualization of the actual path that has been predicted, the cells are joined to form a polygonal chain.

- The motion planning algorithm can safely evaluate a route from the start to the goal and compute speed commands for the vehicle depending on the chosen path.

    - A few algorithms have been implemented: the first one, the dynamic window approach, has shown not to handle all the constraints and limitations to which a car is subjected; the solution to this has been to evaluate another motion planner, the timed elastic band, designed for differential-drive robots with turning radius different from 0. Basically it is suited for robots that are not capable to rotate on themselves, just like cars.
    - Since dynamic obstacles have to be handled, the implemented motion planner is a dynamic one: this means that it is capable to re-evaluate the best path to reach the goal depending if new information happens to be on the map. In fact, if a moving object appears in the surroundings of the ego-vehicle, or a previously recognized one moves away from the previous position, it is evaluated if this can have an impact on the actual path the ego-vehicle is taking. If so, a new one is found.

The result of this work is thus that:

- the vehicle is capable to evaluate a safe path to be taken
- the vehicle is capable to react if a new information about an obstacle arrives: what happens is that it is actually capable to slow down or to stop the motion and re-evaluate the best path if it recognizes that the previously chosen one has a safety critical issue. Besides, if the moving object moves away and it is no more an obstacle for the ego-vehicle, it is capable to take the previously evaluated best path, if it is still the preferred one.

This last point constitutes the major achievement: the union of a trajectory prediction algorithm and a dynamic motion planner gives the possibility to start to talk about "autonomous vehicle".

As a matter of fact this implementation has some limitations that will be discussed later in paragraph 5.1.
They could be anyway topics from which thesis of other students can start or topics that can be better explored also by the authors of this thesis to upgrade knowledge and implementation.

5.1 Future work

The work for this thesis has gone through some choices that determined the achieved performances. However, improvements could be made that can constitute subject of future works.
The ego-vehicle is considered moving **only around the z-axis**, so only a yaw angle is defined. This is due to the fact that, in general implementations of motion planning algorithms, the vehicles are considered to move simply in a flat environment.
This reflects in the fact that the state vectors of the Kalman filter have less parameters to take into account and the equations of the system, represented by the moving obstacle, are

easier to be solved. A future work could be to introduce movements around axes x and y, so to take into account also roll and pitch angles.

Only some **motion models** have been considered for the Kalman filter and their results have been compared. They have been chosen for the low levels of complexity, due to the fact that they are *linear motion models*. The main drawback is that these models assume straight motions and are thus not able to take rotations (especially the yaw rate) into account. Future work could be to apply the Kalman filter to *curvilinear models* which have a second level of complexity that can be defined by taking rotations around the z-axis into account.

The global planner that has been implemented is based on the **A*** path planning algorithm.
A* is formulated in terms of weighted graphs and it is an exploring algorithm, in the sense that it searches in the space of configurations for the best path to the destination.
The A* algorithm works fine in static environment. In dynamic environment it does not because, once the car has decided for a path, it does not change its mind so, if an obstacle happens to be in the middle of the path, there would be a crash.
In the State of the art section, other algorithms have been described that could give better performances as they take into account dynamic obstacles. A possible future work could be to implement another one of them.

In robotic motion planning, the **dynamic window approach** and the **timed elastic band** are two very important collision avoidance algorithms for mobile robots. The dynamic window approach is derived directly from the dynamics of the robot, and is especially designed to deal with the constraints imposed by limited velocities and accelerations of the robot. The timed elastic band has as goal to minimize the time to reach the goal taking into account the dynamic constraints of a car. Anyway there are other motion planning algorithms that have been described in the State of the art that can also have good performances. It could be interesting to implement one of them in the future.
Also another possible upgrade could be to have them work in an environment with a local costmap that has dynamically changing dimensions: width and length can change with the speed of the ego-vehicle thus producing a greater area to be aware of whenever the speed is high, a small area whenever the speed is low.

6. Bibliography

[1] ROS Wiki, (Last modification, 2014-06-21 11:53:27). ROS/Concepts. Retrieved from http://wiki.ros.org/ROS/Concepts

[2] Ben-Ari, Mordechai, and Francesco Mondada. *Elements of Robotics*. Springer International Publishing, 2018

[3] LIDAR Technology ► Ibeo Automotive Systems GmbH, Hamburg. (n.d.). Retrieved from https://www.ibeo-as.com/aboutibeo/lidar/

[4] Barrera S., Comito V.*, A ROS-based Platform for Autonomous Vehicles: Foundations and Perception,* Master's Degree Thesis, *2018*

[5] ROS Wiki, (Last modification, 2018-01-10 15:43:59). costmap_2d. Retrieved from http://wiki.ros.org/costmap_2d

[6] ROS Wiki, (Last modification, 2017-10-02 13:40:32). tf. Retrieved from http://wiki.ros.org/tf

[7] Bmw. (2019, January 25). Autonomous Driving – five steps to the self-driving car. Retrieved from https://www.bmw.com/en/automotive-life/autonomous-driving.html

[8] Fox, D., Burgard, W., & Thrun, S. (1997, 03). The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine, 4*(1), 23-33. doi:10.1109/100.580977

[9] Burgard, W., Stachniss, C., Bennewitz, M., Arras, K., Introduction to Mobile Robotics *Robot Motion Planning*, Slides by Kai Arras*,* Uni Freiburg

[10] Minguez, J., & Montano, L. (2004, 02). Nearness Diagram (ND) Navigation: Collision Avoidance in Troublesome Scenarios. *IEEE Transactions on Robotics and Automation, 20*(1), 45-59. doi:10.1109/tra.2003.820849

[11] Borenstein, J., & Koren, Y. (1991, 06). The vector field histogram-fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation, 7*(3), 278-288. doi:10.1109/70.88137

[12] Gehrig, Stefan K., and Fridtjof J. Stein. "Collision Avoidance for Vehicle-Following Systems." *IEEE Transactions on Intelligent Transportation Systems* 8, no. 2 (06 2007): 233-44. doi:10.1109/tits.2006.888594.

[13] C. Rösmann, F. Hoffmann and T. Bertram: Kinodynamic Trajectory Optimization and

Control for Car-Like Robots, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Vancouver, BC, Canada, Sept. 2017.

[14] Cannon, J., Rose, K., Ruml, W. 2012. Real-Time Motion Planning with Dynamic Obstacles. In Symposium on Combinatorial Search.

[15] ROS Wiki, (Last modification, 2013-04-25 00:17:00). Inflation Costmap Plugin. Retrieved from http://wiki.ros.org/costmap_2d/hydro/inflation

[16] Schubert, R., Richter, E., Wanielik, G.: Comparison and evaluation of advanced motion models for vehicle tracking. In: International Conference on Information Fusion, pp. 1-6 (2008)

[17] On-Line Computer Graphics Notes. http://www.idav.ucdavis.edu/education/GraphicsNotes/Bresenhams-Algorithm/Bresenhams-Algorithm.html.

[18] ROS Wiki, (Last modification, 2018-06-14 15:58:10). dwa_local_planner. Retrieved from http://wiki.ros.org/dwa_local_planner

[19] ROS Wiki, (Last modification, 2018-07-05 13:24:48). teb_local_planner. Retrieved from http://wiki.ros.org/teb_local_planner

7. Acknowledgements

Vorremmo ringraziare il relatore Massimo Violante per tutti i consigli dati e per la disponibilità.

Ringrazio i miei genitori per avermi sempre sostenuto e per avermi dato la possibilità di intraprendere questo percorso; sono sempre stati il mio punto di riferimento.
Ringrazio i miei fratelli che con il loro affetto mi sono sempre stati vicini e la mia ragazza Alessia che, nonostante gli alti e bassi, ha sempre creduto in me.
Un ringraziamento particolare va alla mia collega e amica Assunta con la quale è stato un piacere realizzare questo progetto.
Ringrazio tutto il team di Objective software che è sempre stato come una grande famiglia per me. In particolare, colgo l'occasione per ringraziare Marco per avermi dato la possibilità sin dall'inizio di unirmi al gruppo, credendo nelle mie capacità; Simone e Stefano che hanno saputo ritagliare un pò del loro tempo per seguirci in questo percorso, dandoci sempre dei preziosi consigli; Vincenzo e Sebastiano che sono stati dei veri e propri fratelli maggiori in tutto il lavoro da noi svolto; Michele che mi è stato vicino durante il lavoro e durante lo sviluppo della tesi, come collega ma soprattutto come grande amico.

Vorrei ringraziare il mio collega e amico Luigi: abbiamo tanto sudato per arrivare fino a qui, ci siamo anche scontrati, ma ce l'abbiamo fatta e alla fine direi che possiamo andare solo fieri di tutto quello che abbiamo ottenuto.
Vorrei dire grazie a tutti i colleghi di lavoro (in particolar modo al mio Boss) per il loro sostegno, per l'intesa che si è instaurata, tale da poterci definire una piccola famiglia.
Ringrazio tutti gli amici, Filomena, Angela, Federico, che hanno saputo arricchire gli anni di questo percorso così da renderlo speciale e indimenticabile.
Doverosa menzione per i mitici Francesco, Valeria, Luca e Vito, i primi conosciuti in una città sconosciuta! Non avrei potuto incontrare persone migliori, persone che mi hanno aiutata a crescere e con cui ho condiviso dolori ma anche tante tante gioie!
Quanti ricordi, quante risate e soprattutto che "modalità"..
Ringrazio Matteo, perchè è semplicemente quella parte di cuore che mi mancava, perchè mi ha insegnato, mi ha consigliato, mi ha sorretto quando situazioni nuove mi
spaventavano. E' stato la fonte di sorrisi e risate anche quando l'animo era cupo e pieno di pensieri, riuscendo a dare un senso anche ai giorni più bui.
Dulcis in fundo, ringrazio la mia famiglia senza la quale arrivare fin qui sarebbe stato impossibile! Sono stati sempre sostegno e forza, senza di loro non sarei la persona che sono. Ringrazio mia madre per la sua perseveranza e tenacia e pazienza nel saper gestire anche i tratti peggiori del mio carattere; ringrazio mio padre per aver sempre spinto affinchè facessi ciò che era giusto fare, sostenitore quanto giudice severo e giusto della mia carriera; ringrazio mio fratello per la sua grande complicità e per i consigli dati e saputi ricevere.

Grazie a tutti.. e ad maiora!