POLITECNICO DI TORINO

MASTER's Degree in ELECTRONIC ENGINEERING

MASTER's Degree Thesis

Software development and validation for imaging RADAR adopting innovative mmWave FMCW sensor

Supervisor

Prof. RICCARDO MAGGIORA

Candidate

LUCA LOMBARDINI

APRIL 2023

Summary

RADAR systems are a class of embedded devices able to carry out complex tasks and measurements while providing multidimensional performance figures. Their history dates back to the early Twenties of the 20*th* century and since then RADARS have been improved constantly and they also have catalyzed the interests of many actors. Thus their study and development are a hot topic nowadays because these systems have gradually permeated various areas of key importance like automotive, industrial, scientific, defense and safety just to name a few.
Like many other systems, RADARS have seen and still see technological improvements like large-scale integration, better mixed signal and RF techniques, which have made integrated system-on-chip solutions a quite cheap and attractive solution while providing good performance in terms of absolute range, velocity, spatial location and their resolutions.
Together with hardware improvements, the software also must scale in terms of complexity because it has to manage more hardware units and software packages, while offering more features. Usually these software features are extracted by processing the raw information gathered by the RADAR and then sent to higher level where final decisions are taken, whether it is human or another system. Some typical software feature that all RADARS implement in some way are target tracking and classification, but many other exist.
Regarding the target tracking feature, its performances are strongly dependent on the application goals: number, type, material, shape, orientation, velocity, scenery, etc., of targets must be taken into account. All of these parameters are considered with respect to the RADAR itself.
An issue which arises is the situation when the RADAR is installed on a moving host or vehicle. This means that the RADAR reference system is moving and thus does not coincide with a fixed reference. In this case, the RADAR velocity is associated to the set of data that it collects, thus introducing a systematic effect which must be compensated for. A typical use case in which this situation leads to errors is when the RADAR is deployed in a selective-tracking application, where not all the collected points must be considered equal but grouped into proper tracks.
A way to extract this information is to analyze and process the data that the

RADAR collects: various possible solutions can be analyzed and a software implementation is explored and presented in this Thesis. The project is developed around the AWR1843 System on Chip (SoC) by Texas Instruments, which is an available Radar SoC targeted to automotive applications offering the dual band operation in either Automotive LRR (Long Range Radar) band (76-77 GHz) or Automotive SRR (Short Range Radar) band (77-81 GHz).
The project takes as starting point an already developed software infrastructure, which implements target group tracking, which is modified in order to extract the RADAR's Ego Velocity, known as the velocity at which the RADAR itself is moving inside an environment considered to be static and thus used as the spatial reference. The Ego Velocity estimation has been designed to be a self-contained software package written in the *C* language, with the possibility to be integrated into different RADAR software stacks and projects. The routine's development, testing and validation were performed in close relationship with each other. The development of a proper estimator function originated as a foundational interpretation of a research effort made by Kellner et al[1]. This implementation acted as the starting point, and its testing and validation procedures provided indications about the result quality and its adherence to the requirements. These indications were then used in the subsequent development-testing-validation iteration, until the validation confirmed the suitability of the implementation to be employed for the intended use case. The test campaign was designed in order to emulate a real world scenario in which the software routine behavior is checked for bugs and misunderstood requirements. In order for the tests to provide the highest amount of useful information, the same set of acquired data have been used across the various implementations. The data which has been reused for the test and validation processes has been collected by installing the RADAR on a host car and then saving the acquisition on a personal computer. This choice allows for a real case which can be reproduced "in silico", without the need to set up again the whole testing platform and environment. Furthermore, by having the same set of data allows for a comparable mean with which is possible to evaluate the results and performance of the various implementations that have been explored.
The RADAR data have been collected through the SPI serial communication provided by the AWR1843 SoC, then converted to the Ethernet protocol in order to let the PC collect it. The RADAR which has been used as testing and validation platform packs a complex firmware infrastructure in order to overcome the FMCW chirp parameters' limitations[1.4]. The frame acquisition is the result of a combination of two sub-frames. The task of the first sub-frame is to retrieve a large amount of targets with ambiguous Doppler velocity[1.1.7], while the task of the second is to use two sets of chirps with different parameters in order to disambiguate the targets' group leader (Master) velocity. The Doppler Disambiguation exploits the different ambiguous velocity obtained with the different chirp parameters by finding

the least common velocity which satisfies the system of congruence constituted by the Doppler processing of the second sub-frame. Once the Masters' velocities are disambiguated, their velocity is associated to the same Master to which it corresponds in the first sub-frame. In order to optimally test and validate the Ego Velocity routine, it must be integrated in the RADAR Software Stack. Thus the acquisition schema must be taken into account. These considerations pertains to the point in which the routine is integrated (when to execute it) and which data set is used to compute it (what does it access). In the light of these considerations, the obtained Ego Velocity is computed after the Doppler Disambiguation and before the raw measurement point (spherical coordinates) conversion to tracking-suitable point (Cartesian coordinates). For what regards the accessed data, it uses the disambiguated velocity of the second sub-frame's targets. The conversion routine, executed before the Tracking algorithm, has been adapted to get the Ego Velocity as input in order to apply corrections and compensation. Once a proper Ego Velocity estimation has been accomplished, the overall effect on the tracking performance is compared to that of the previous software version, which assumed zero Ego Velocity. This check is performed on the Tracks collected on a PC, processed and graphically displayed using MATLAB.
With respect to the set of tests, the RADAR installed on the host car has captured different scenarios obtained as a combination of the host car and single target car movements. The static and dynamic host tests were conducted for the background scenery, as well as the target opening and closing to the RADAR. Furthermore an accelerating host car test (with just the background scenery) was acquired, along with a challenging test case that involved a large number of synthetically added target cars with equivalent properties. Ultimately the performance between the implementations are discussed, a solution is suggested as the most appropriate one and some more approaches are presented. A possible evolution for this routine is the integration of the host steering direction implementation using the acquired data, thus implementing also the Direction Estimation together with the Ego Estimation.

Figure 1: *RADAR* configuration.

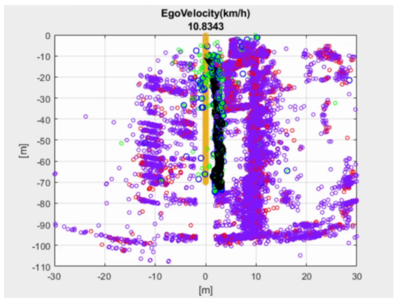


Figure 2: Example of sampled data.

In figure 2, the red and purple markers are static points, blue and green in movement, while the black ones are the tracks.

Table of Contents

List of Tables

List of Figures

Acronyms

ADC

Analog-to-Digital Converter

AoA

Angle of Arrival

ARM

Advanced Risc Machine

BPM

Binary Phase Modulation

CFAR

Constant False Alarm Rate

CFM

Carrier Frequency Multiplexing

CPU

Central Processing Unit

DSP

Digital Signal Processor

EDMA

Enhanced Direct Memory Access

EM

Electro-Magnetic

FFT
Fast Fourier Transform

FMCW
Frequency-Modulated Continuous Wave

FOV
Field of View

IEEE
Institute of Electrical and Electronics Engineers

IF
Intermediate Frequency

LNA
Low Noise Amplifier

LO
Local Oscillator

MIMO
Multiple Input Multiple Output

MMWAVE
millimiter wave

PA
power amplifier

PC
Personal Computer

PLL
Phase Locked Loop

PRF
Pulse Repetition Frequency

PRI	Pulse Repetition Interval
RANSAC	Random Sample Consensus
RCS	Radar Cross Section
RF	Radio-Frequency
RX	Receiver
SoC	System on Chip
SPI	Serial Peripheral Interface
TDM	Time Division Multiplexing
TI	Texas Instruments
TX	Transmitter

Chapter 1

Introduction to RADAR

This chapter presents the *RADAR* as a technology, thus exploring the theoretical background and details on which these devices are based in order to perform their duties.
The *RADAR* word is the acronym used as an abbreviation for the sentence "RAdio Detection and Ranging". This acronym refers to systems which employ electromagnetic waves in order to detect the return echo of objects which are displaced in an environment. Some initial attempts to implement a similar technology dates back to the early *20th* century, while its most notable adoption of these systems arose during the World War II period. Back then *RADAR*s where in charge of the detection of enemy airplanes and warships. Since then *RADAR*s have evolved, leading to improved accuracy and performance. The technological improvements allowed for this technology to also permeate other fields, being adopted also for civil applications other than the initial military ones.
Besides their evolution, the same basic concepts still hold true nowadays, while more advanced techniques have been gradually adopted.

1.1 Pulsed RADARs

For a classic *Pulsed RADAR*[2], an electromagnetic pulse of duration τ [s] is generated by a driver electronic circuit and transmitted through a transmitting antenna. This antenna is orientated in the direction on which the detection must be performed. A receiver electronic circuit is employed for the detection. This receiver circuit can use the same transmitting antenna or a different one. After some signal conditioning and processing, it is able to detect the return echoes produced by the transmitted pulse which bounced on any reflecting target if present. If the pulse is transmitted periodically with a period of T [s], the latter time interval is called *Pulse Repetition Interval* (PRI). In figure 1.1 this timing schema is presented.

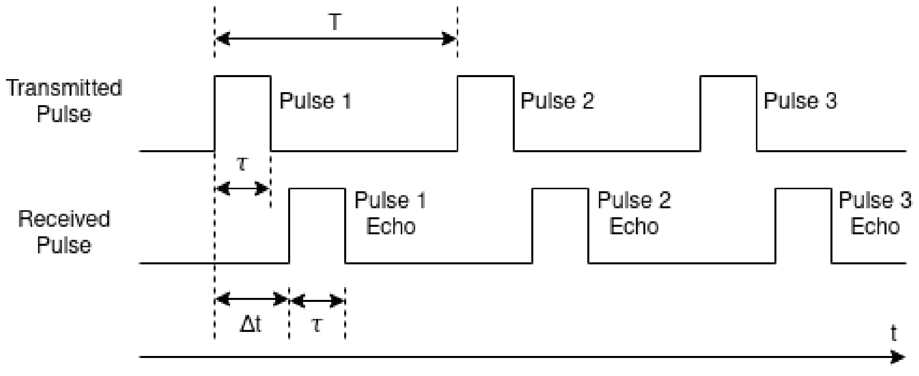


Figure 1.1: A Pulsed *RADAR*'s transmitted and received pulses, in time domain.

By analyzing the return echoes, a *RADAR* system is able to detect targets and to perform various kind of measurements on them. Possible measurement of a target may include *Range*, *Doppler Shift*, *Azimuth* angle position, *Elevation* angle position and many more.

1.1.1 Pulsed RADARs: Range Estimation

The *Range*[2] is known as the distance at which a target is located with respect to the *RADAR*. The time difference between the transmitted pulse and the received echo from a target is proportional to the target's distance. The proportion between this two quantities is the pulse's propagation velocity in the medium in which it is travelling. In a free space medium, like air, the pulse propagates as the speed of light, whose value is approximated to $c = 3 \times 10^8$ [m/s] and assumed to be constant. The time difference between the transmitted pulse and its received echo, identified as Δt, can be used to calculate the target's *Range* as:

$$R = \frac{c \times \Delta t}{2} \quad [m] \tag{1.1}$$

The *Range Resolution*[2] is defined as the minimum distance between two different targets that makes said targets be distinguished by the *RADAR* as two different ones. Thus, in a pulsed *RADAR* the minimum detectable distance is dominated by the pulse duration:

$$\Delta R = \frac{c \times \tau}{2} \quad [m] \tag{1.2}$$

The *Maximum Unambiguous Range*[2] is the maximum *Range* after which a target's return echo is ambiguous with respect to the transmitted pulses. This situation is caused by echoes of previous pulses that return during the *PRI* of another pulse. Thus the *Maximum Unambiguous Range* is tightly related to the *PRI* as:

$$R_{max} = \frac{c \times PRI}{2} \quad [m] \tag{1.3}$$

Once defined the *Maximum Unambiguous Range*, known also as *Maximum Range*, and the *Range Resolution*, it is possible to identify the *Range Bin*[2], which indicates how many different ranges a *RADAR* can detect. It can be expressed as:

$$M = \frac{R_{max} - R_{min}}{\Delta R} \tag{1.4}$$

This indicates that a *RADAR* is able to detect two different targets if those are located on different *Range Bin*s. Thus, in order to detect two different targets at the same bin, the *RADAR* must increase its *Range Resolution* or use different techniques.

The *Range Resolution* is a quantity which can be analyzed in terms of frequency content. The transmitted rectangular pulse signal has a duration τ and it can be modeled in the time domain as:

$$x(t) = \begin{cases} 0 & \text{if } |t| > \tau/2 \\ A & \text{if } |t| \leq \tau/2 \end{cases}$$

The frequency content of the rectangular pulse is the *sync* function, whose bandwidth is given as:

$$B = \frac{1}{\tau} \quad \left[\frac{1}{s}\right] \tag{1.5}$$

The result above states that in order to increase the *Range Resolution* of a pulsed *RADAR*, a shorter pulse is needed. But shorter pulses increase the bandwidth usage. Furthermore using shorter pulses leads to lesser transmitted energy during the pulse, for a given peak transmitted power (P_t). Also the average transmitted power (P_{avg}) decreases. The average power scales with the *Duty Cycle*:

$$P_{avg} = d_c \times P_t \quad [W] \tag{1.6}$$

Where the *Duty Cycle* is expressed as:

$$d_c = \frac{\tau}{PRI} \tag{1.7}$$

A way to increase the transmitted energy is to increase the peak transmitted power, but this solution this solution is only theoretically valid. In fact is impossible to safely operate a system with extreme peak powers delivered in very short times.

1.1.2 RADAR Equation

The *RADAR* equation[2] is a mathematical model which describes the relationship between the various physical quantities that are involved during the transmitted signal propagation to the target and its echo return. The returned echo results greatly attenuated with respect to the power levels of the transmitted signal.
The received power depends on:

- the transmitted power (P_t)
- the *RADAR* antenna gain (G)
- the given target's *Radar Cross Section* (σ)
- the antenna's effective aperture (A_e)
- the distance between the *RADAR* and a given target

Neglecting the *noise* contribution, this mathematical model can be described with the following formula:

$$P_r = \frac{P_t G \sigma A_e}{(4\pi R^2)^2} \quad [W] \tag{1.8}$$

The reason for which the formula 1.8 is inversely proportional to $(4\pi R^2)^2$ is that as can be seen in figure 1.2, the electromagnetic wave propagation can be modeled as a spherical wavefront. Thus the power density is spread across the spherical surface both during the signal transmission and during the echo reception.

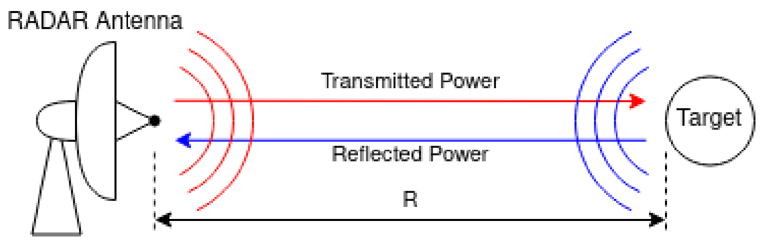


Figure 1.2: Graphical representation of the physical quantities effects on the transmitted and received powers.

In case the *noise* has also to be taken into account, the formula 1.8 can be adapted to a *Signal-to-Noise Ratio* expression by modeling the noise as an equivalent thermal noise (kB_wT). Furthermore also the losses (L) can be taken into account. These losses can group together many terms which originate from various sources such as *Signal Processing* losses, microwave losses and waveguide losses, but many more can be considered. Some losses have also a dependency on the signal frequency, thus making the mathematical model even more complex. Thus the *Signal-to-Noise Ratio* of the returned echo can be expressed as:

$$\frac{S}{N} = \frac{P_t G \sigma A_e}{(4\pi R^2)^2 k T B_w L} \tag{1.9}$$

1.1.3 Radar Cross Section

The *Radar Cross Section*[2] is a parameter of key importance for *RADAR* systems. The *Radar Cross Section* parameter contains the information about the target's physical characteristics and modeling all of those is a complex task. A common definition relies on the target's equivalent area computation, which can be obtained as "*the ratio of the power scattered from an object in units of power per solid angle (steradian) normalized to the plane wave illumination in units of power per unit area*"[3]. Thus this parameter can be obtained as:

$$\sigma = \frac{4\pi \frac{\text{reflected power directed to the receiver}}{\text{solid angle}}}{\text{incident power density on the object}} \quad [m^2] \tag{1.10}$$

The *RCS* models how an illuminated object reflects back the incident electromagnetic wave's energy towards the *RADAR*, thus giving information about the echo strength. This parameter ultimately depends on the object's material, its size, its orientation with respect to the incident power, the electromagnetic wave's polarization, the signal frequency (etc).

Corner Reflectors are artificial targets used for test and calibration processes of *RADAR*s. These particular test targets are built using highly reflective materials and adopting particular shapes in order to provide the desired directivity. A typical *Corner Reflector* can be realized with metal sheets jointed in a trihedral angle structure. This particular shape grants an almost constant response for each angle in the range $-60°$ to $60°$ as shown in figure 1.3.

$$\sigma_{corner} = \frac{4\pi {A_e}^2}{\lambda^2} \quad [m^2] \tag{1.11}$$

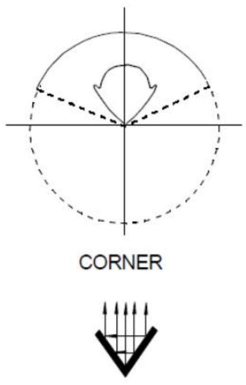


Figure 1.3: The echo *Radiation Pattern* of a Corner reflector[4]

1.1.4 Detectability & Time-Bandwidth Product

The *Detectability* of a target depends on the received echo energy with respect to the *noise* energy. This means that a target detection can be improved by maximizing the instantaneous peak-signal-to-mean-noise power (R_f).

$$R_f = \frac{|s_0(t)|^2_{\max}}{N} \tag{1.12}$$

Maximizing this quantity means to apply to the received echo the optimal filter. The optimal filter is the *matched filter*, which has a frequency response which is the complex conjugate of the transmitted signal. If this processing is applied, the *SNR* is maximized to $\frac{2E}{N_0}$.

Pulsed *RADARs* have a bandwidth usage which depends on the transmitted pulse duration, thus the *time-bandwidth* product is limited to 1.

$$B\tau = 1 \tag{1.13}$$

If the matched filter processing is adopted, the *Detectability* can be improved by two means: eiterh improving the *SNR* figure or increase the *Time-Bandwidth Product*. The former leads to largely increased system cost, while the latter increases the processing cost. This statement is supported by the result obtained in (1.17). The noise energy entering the *RADAR* system is

$$N_i = \frac{N_0}{2}B \quad [J] \tag{1.14}$$

The received echo energy is:

$$P_i = \frac{E}{\tau} \quad [J] \tag{1.15}$$

The *SNR* can be computed as:

$$(SNR)_i = \frac{P_i}{N_i} = \frac{2E}{N_0 B\tau} \tag{1.16}$$

As mentioned above, the *Detectability* is maximized when R_f is maximized. Thus the expression of the *SNR* performance against the R_f leads to:

$$\frac{R_f}{(SNR)_i} = B\tau \tag{1.17}$$

1.1.5 Azimuth and Elevation Angles

RADAR systems are able to detect the *Angle of Arrival* (AoA), but in order to compute it, more complex architectures must be adopted. A possible implementation is the adoption of *mechanically steered antenna*, thus changing the antenna beam orientation in order to scan for targets in the desired direction. This implementation requires expensive mechanical components for the waveguide joints.
The same task can be performed by adopting a different approach based on signal phase shift and the adoption of *antenna arrays*. The *beamforming* technique focuses on the phase modulation during the signals' transmission, while the *MIMO* technique focuses on the phase of the received signals. These techniques can be used both alone or concurrently.

1.1.5.1 Angular Resolution of a Single Antenna

The *Angular Resolution* of a single antenna can be approximated, in *Far Field Condition*s, as a dependency between the transmitted signal's wavelength (λ) and the antenna's dimensions (d). This means that the *RADAR* is able to detect two targets position as different only if their angular difference is larger than the angle θ_S, as modeled in 1.18.

$$\text{Angular Resolution} = \theta_S = \frac{\lambda}{d} \quad [rad] \tag{1.18}$$

The *Angular Resolution* model thus improves at higher frequency and with larger antennas. The *Angular Resolution* can be also translated into the *Cross-Range Resolution*, which is the spatial resolution orthogonal to the radial *Range*. Thus it depends on the actual *Range* at which the detection is performed, as modeled in 1.19.

$$\text{Cross-Range Resolution} = r \cdot \theta_S = r \cdot \frac{\lambda}{d} \quad [m] \tag{1.19}$$

In figure 1.4, a representation of these quantities is provided.

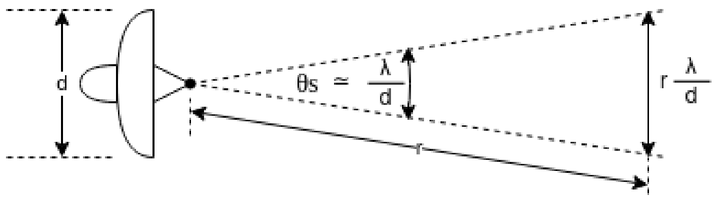


Figure 1.4: Graphical representation of the Angular Resolution of a Single Antenna.

A possible way to increase the *Angular Resolution* is to adopt an *Antenna Array* schema, which provides also the possibility to implement more complex techniques like *Beamforming*.

1.1.6 Beamforming

The *Beamforming* is a technique used to change the equivalent radiation pattern of an antenna array by changing the phase on each antenna element. This operation can be performed both on transmission and reception of signals and also adopting *Analog* or *Digital* techniques. The array driven in this way takes the name of *Phased Array Antenna*. This technique is based on each signal's interference, which result in an equivalent wavefront orientated towards the direction proportional to the applied phase, as modeled with the formula 1.20.

$$\Delta\phi(\psi_i) = sin(\psi_i)2\pi\frac{d}{\lambda}[rad] \tag{1.20}$$

When the phase shift $\Delta\phi(\psi_i)$ is applied with a proper integer multiple to each antenna, the *Phased Array Antenna* produces a wavefront which is orientated towards the ψ_i angle direction, as can be seen in figure 1.5.

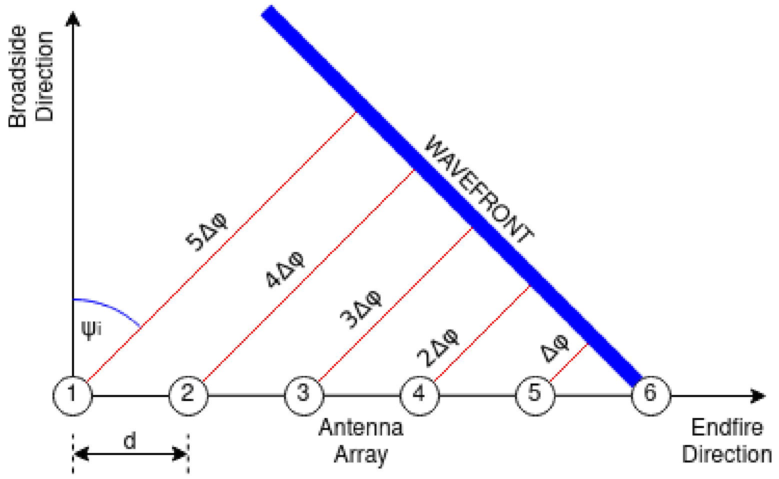


Figure 1.5: Beamforming shaping effect on the wavefront orientation in a *Phased Array Antenna*. This example refers to a six elements antenna array.

1.1.6.1 Analog Beamforming

The *Analog Beamforming* consist in applying an analog phase shift on the signals whether on the transmitter, the receiver or both. This technique allows to shape the *antenna array*'s *Main Lobe* to be oriented in the desired angular orientation. In figure 1.6, a possible block schema is presented.

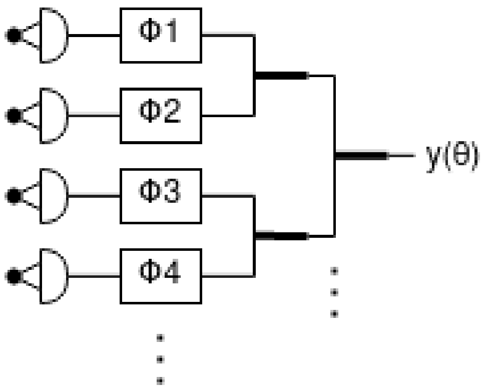


Figure 1.6: Analog Beamforming structure.

In order to adopt this technique, a *Phase Shifter* is required on each channel. Thus the *Analog Beamforming* technique has the inconvenience of having an higher cost and susceptibility to noise and non-idealities, which are introduced by the additional *Phase Shifters*.

1.1.6.2 Digital Beamforming

The *Digital Beamforming* consist in applying a phase shift during the *Signal Processing* of the *RADAR*. This processing technique does not require the adoption of analog phase shifters, thus leading also to better *SNR* figures and reduced hardware cost. The signal echoes are received by all the receiving channels at the same time, sampled and then store in memory. Then a phase shift can be introduced by *numerical* means, thus providing the *RADAR* the capability of scanning in each *Angle Bin* by applying a different phase shift, without the need to perform a different data acquisition on the receiving antennas. This procedure is also referred as "*Electronic Beam Steering*".

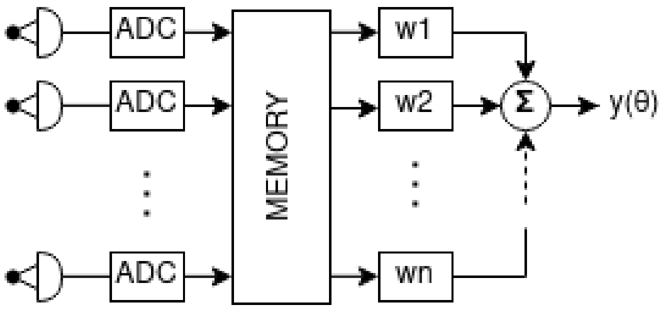


Figure 1.7: Digital Beamforming structure.

The *ADC* sampled data is processed with the weighing phases in what is referred as *Angular Compression*. The *Angular Compression* consist in applying to each sampled channel data ($\vec{s_{rc}}$) the proper phase shift, thus obtaining the *Angular Compressed Signal* ($s_{rc,ac}$) as shown below:

$$s_{rc,ac} = \vec{s_{rc}}^{T} \cdot \vec{b}(\psi_i) \tag{1.21}$$

The phase shift vector is obtained by adopting the expression 1.20 with the proper antenna index as phase multiplicity factor. This results in multiplying to the sampled data ($\vec{s_{rc}}$) the vector 1.22 to each antenna coordinate.

$$\vec{b}(\psi_i) = \begin{pmatrix} e^{j0\cdot\Delta\phi(\psi_i)} \\ e^{j1\cdot\Delta\phi(\psi_i)} \\ e^{j2\cdot\Delta\phi(\psi_i)} \\ . \\ . \\ . \\ e^{j(N-1)\cdot\Delta\phi(\psi_i)} \end{pmatrix} \tag{1.22}$$

For what regards the *Angular Resolution*, it can be computed as the first null of the *Angular Compressed Signal*. Thus the first null ψ_{fn} is obtained as follows:

$$sin(\psi_{fn}) = \frac{1}{N\frac{d}{\lambda}} \tag{1.23}$$

When the adopted *Antenna Array* has an *half-wavelength* distance between each element, the *Field of View* is maximized. The resulting FoV is from $-90°$ to $90°$ and has no ambiguity. Furthermore the first null ψ_{fn} computation reduces to just a dependence on the amount of *Antenna Elements* N, as shown below:

$$sin(\psi_{fn}) = \frac{2}{N} \tag{1.24}$$

The importance of the amount of *Antenna Elements N* is directly related to the *Angular Resolution*, meaning that having a smaller *First Null Beamwidth* makes the *RADAR* able to detect more targets per *Range Bin*. The larger the amount of *Antenna Elements*, better the *Angular Resolution*. Recalling the *Angular Resolution*, different targets must dwell in different *Angular Bins*, so up to N targets can be successfully identified in each *Range Bin*.

1.1.7 Velocity Estimation in Pulsed RADAR

RADAR system can measure a target's velocity by means of the *Doppler Frequency Shift*[2]. This velocity affects the transmitted signal by modulating its frequency. This effect occurs due to the time compression of the reflected signal that occurs when it hits on a moving target. The time compression alteration of the signal in the *time domain* results in a frequency shift in the *frequency domain*. A transmitted signal as

$$x(t) = y(t) \cdot cos(\omega_0 t) \tag{1.25}$$

is received as

$$x_r(t) = y(\gamma t - \Psi_0) \cdot cos(\gamma \omega_0 t - \Psi_0) \tag{1.26}$$

where γ is

$$\gamma = \frac{2v}{c} \tag{1.27}$$

The received signal represented in the *frequency domain* is

$$X_r(\omega) = \frac{1}{2\gamma}(Y(\frac{\omega}{\gamma} - \omega_0) + Y(\frac{\omega}{\gamma} + \omega_0)) \tag{1.28}$$

As can be noted, the time compression factor γ modulates the frequency. This is equivalent to a frequency shift by the amount f_d.

$$f_d = \frac{2v}{c} f_0 = \frac{2v}{\lambda} \quad [Hz] \tag{1.29}$$

This phenomenon allows to also determine when a target is approaching (*closing target*), in which case the shift is positive and thus the frequency increases. If instead a target is leaving (*opening target*), the shift is negative and thus the frequency decreases. Each velocity measured with the *Doppler* shift is relative with respect to the *RADAR*.
Another way to consider the *Doppler Frequency Shift* is by its effect on the signal phase. The target space variation due to its velocity makes the echo signal phase change at a faster (if *closing*, slower if *opening*) rate than the transmitted signal. Thus the frequency shift can also be expressed as:

$$f_d = \frac{1}{2\pi}\frac{d\Phi}{dt} = \frac{1}{2\pi}\frac{4\pi}{\lambda}\frac{dR}{dt} \rightarrow f_d = \frac{2 \cdot v}{\lambda} \quad [Hz] \tag{1.30}$$

The *RADAR* acquires information about this shift at each *PRI*, thus it samples the *Doppler* frequency shift at a sampling rate of *PRF* ($\frac{1}{PRI}$). This fact limits the maximum velocity that the *RADAR* can acquire without ambiguity at velocities whose *Doppler* shift not exceeds the half of *PRF*.

$$v_{max} = \frac{\lambda \cdot PRF}{4} \quad \left[\frac{m}{s}\right] \tag{1.31}$$

In case targets with velocities which are faster than the maximum are not correctly recognized, and thus are *Ambiguous*. An ambiguous velocity is under-sampled by the *RADAR* and thus results in a velocity lesser than the real one. In order to overcome the ambiguities, an higher *PRF* or more complex *Processing* must be adopted. Recalling the *Maximum Range*, an higher *PRF* leads to a reduced *Maximum Range*.
The *Velocity Resolution* instead is achieved with the amount of samples collected.

$$\Delta f_d = \frac{1}{N \cdot PRI} \quad [Hz] \rightarrow \Delta v = \frac{\lambda}{2 \cdot N \cdot PRI} \quad \left[\frac{m}{s}\right] \tag{1.32}$$

1.2 General RADAR Architecture

The essential architectures used in *RADAR* systems are outlined in this proposed section, which includes an examination of the devices involved and their roles within this type of electronic system. Modern applications are shifting towards a fully integrated RADAR system within a System on a Chip (SoC), which incorporates all the necessary hardware units to carry out its measurement tasks. The proposed figure 1.8 displays the block diagram of the primary architecture for a pulse RADAR without compression.

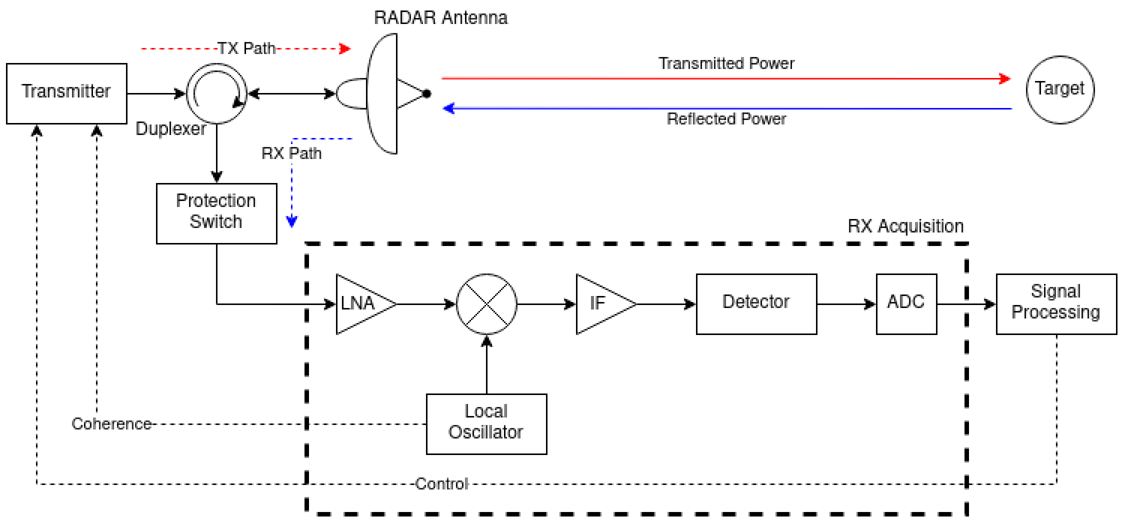


Figure 1.8: *RADAR* architecture block diagram (Real-Only Mixing).

As can be seen in figure 1.8, within the *RADAR* architecture the signal flow can be subdivided in two main section: the transmitter chain and the receiver chain. The former is responsible for the generation of the signal to be transmitted, and thus propagated in the environment through the antenna, while the latter is responsible for collection of the echoes scattered back to the antenna.
In the proposed architecture, the transmitter and receiver chains share the same antenna, thus a circuit separator (like the *Duplexer*) is needed. The *Protection Switch* avoids the strong power leakages and reflections that may occur on the antenna's end during the signal transmission, that may enter in the receiver chain and damage it. In case the *RADAR* adopts different antennas for the transmission and reception, the two chains are fully separated.
The receiver chain is connected to the antenna through the *Protection Switch* during the echo reception. After that, the *Low Noise Amplifier* is employed in order to restore the echo signal's power level before mixing it. The *Mixer* is used in order to down-convert the echo signal since usually the transmitted signal is

up-converted during its generation. After mixing the echo signal with the *Local Oscillator* (*LO*), the obtained *Intermediate Frequency* (*IF*) is amplified by the *IF Amplifier* in order to restore the signal levels for the downstream blocks. Nowadays the *Detector* block is avoided, since modern *RADARs* perform the detection and all the other processing in the *Digital Domain*. Thus, *ADCs* ad *DSPs* are needed in order to perform the required *Signal Processing*.

1.3 Pulse Doppler Processing

In order to correctly perform *Doppler* processing, the *RADAR* must adopt a *Coherent* mixing. The *Coherent* mixing employs the same *Local Oscillator* which is fed to the *Mixer*. This allows to keep phase coherence between the transmitted signal and the received echo. In figure 1.9, a Coherent architecture example is presented.

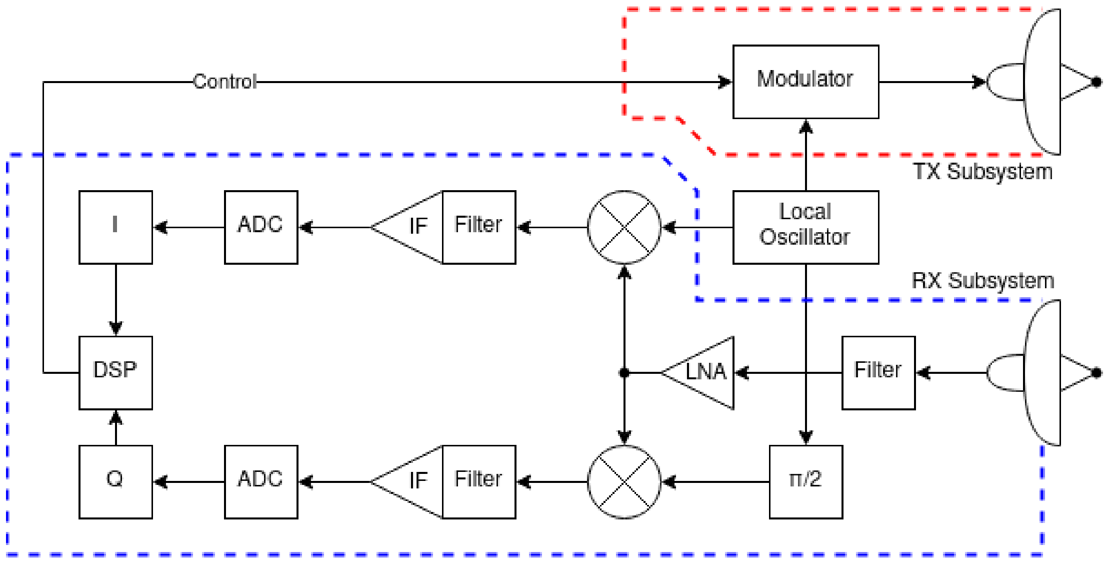


Figure 1.9: Coherent *RADAR* architecture with receiver-only I-Q mixing.

The adoption of an *I-Q* structure allows to relax the *ADC* performance, thus needing an *ADC* with a sampling frequency just as the highest frequency content of the *IF*. Instead twice the hardware and more complex processing is needed.

Since now the *RADAR* also collects the information about the *Doppler* shift as a set of echoes which are used to sample it, the collected data can be organized in a *2-D* matrix.
The *Range* samples obtained directly from the *ADC* constitute the *Fast Time*

coordinate, while the set of sampled echoes (for each *Range Bin*) constitutes the *Slow Time* coordinate.
If the *RADAR* also adopts an *Antenna Array* architecture with separated receiving chain, it can also collect the *2-D* data matrix for each antenna channel. This collected data structure is now *3-D*, having added the *Antenna* coordinate. This structure is known as *RADAR Cube* and can be visualized as the one presented in figure 1.10.

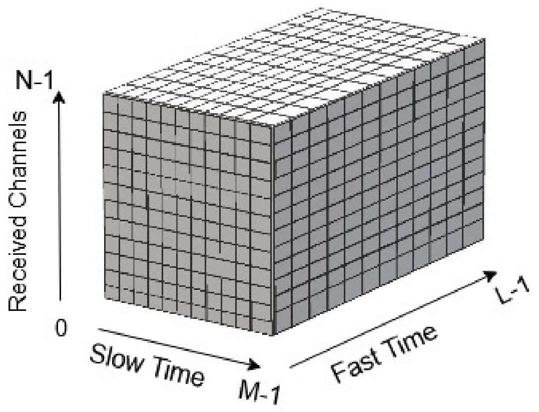


Figure 1.10: Graphical visualization of the *RADAR Cube* structure.

Whenever a full frame data is acquired, consisting of *Range* and *Doppler* samples, the *Doppler Processing* is applied. While the *Range Compression* can be performed when acquiring the data from the *ADCs* and transfering it on the *fast time* coordinate, the *Doppler Processing* on the *slow time* coordinate must be performed after the full frame has been acquired. This computation result in the *Range-Doppler Map* extraction.
This map exhibits peaks where a target is located, both on the *Range* and *Doppler* coordinates. An example for this process is provided in figure 1.11.

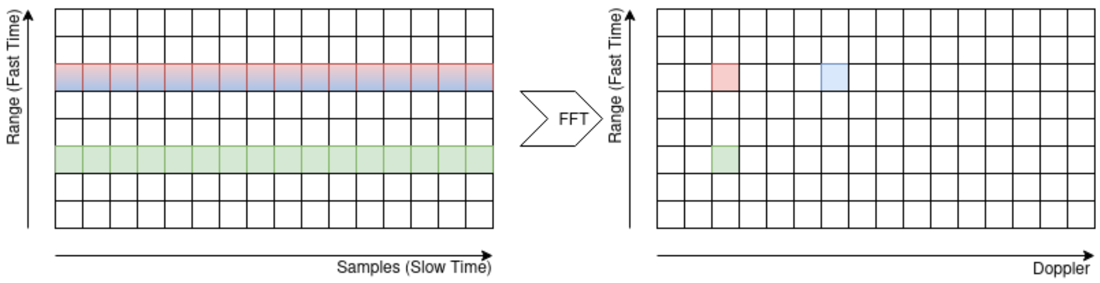


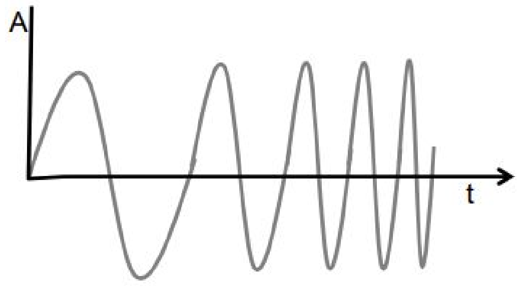
Figure 1.11: Graphical visualization of the *Doppler-FFT* processing.

If multiple receiving channels are adopted, a further processing step is needed in order to compute also the *Angle Processing* (Azimuth and Elevation). *Range Compression* processing depends on the *RADAR* technique adopted. Digitally modulated *RADARs* adopt a correlation processing step, while the *FMCW RADARs* adopt an *FFT* on the *fast time* coordinate for the same *Range Compression*. The *Doppler Processing* instead always requires the *FFT* on the *slow time* coordinate. For what regards the *Angle Processing*, also in this case the complex *FFT* can be adopted in order to retrieve the phase information.

1.4 FMCW Imaging RADAR

The *FMCW RADAR*[2][5] is a particular type of *RADAR* that exploit a frequency modulated transmitted signal for its task. The adoption of this technology allows to increase the signal's *Bandwidth* while using a longer transmission time.
This solution provides at the same time both the required *Range Resolution*, bounded to the signal bandwidth, and improve the *Detectability* with the larger transmitted energy (1.1.4).

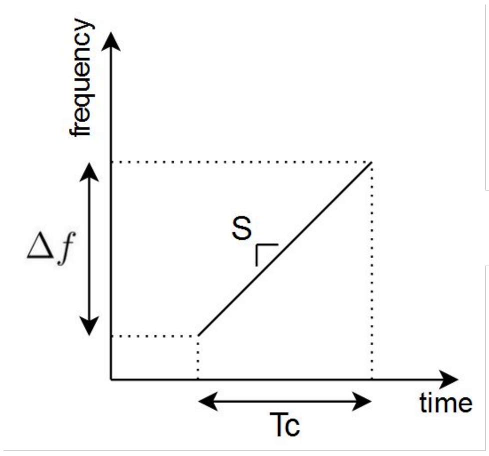
The *FMCW RADAR* increases the *bandwidth* by adoptiong a *chirp* signal, which is an oscillating signal whose frequency is linearly swept from a minimum to a maximum. Its mathematical model is provided in figure 1.12.



$$A(t) = cos(2\pi f_0 t + \pi \mu t^2)$$

Figure 1.12: *Chirp* signal representation in time domain.[5]

The rate at which the frequency is swept is referred to as μ and can be modeled as shown in figure 1.13. The T_c term refers to the *chirp* duration.



$$\mu = \frac{\Delta f}{\Delta t} = \frac{B}{T_c} \quad \left[\frac{Hz}{s}\right]$$

Figure 1.13: *Chirp* signal frequency sweep against its duration time.

In *FMCW RADARs*, the compression gain is thus increased by the *frequency sweep* and the *chirp duration*, as modeled in (1.33).

$$\text{compression gain} = B \cdot T_c \tag{1.33}$$

A key feature of the *FMCW*'s receiving chain is that the usage of a *mixer* allows to obtain a relationship with the *Range* as a frequency beat at *IF* (f_{IF}), whose frequency is proportional to the target's *Range* through the received echo time delay. Thus its optimal response can be obtained applying an *FFT* on the *fast time* coordinate. The relationship between the returned echo's time delay and its *Range* is proportional to the *FMCW* parameters as modeled in (1.34).

$$R = \frac{c \cdot f_{IF}}{2 \cdot \mu} \quad [m] \tag{1.34}$$

Thus the *Maximum Range* depends on the *FMCW* parameters as modeled in (1.34). The restraining parameter regards the data acquisition for the *IF*, thus means that the *ADC* performance is the limiting factor for the *Maximum Range*. The k parameter depends on the actual receiving chain technology: $k = 2$ for a *Real-part-only* chain, $k = 1$ for a *I-Q* chain. In the former case the *ADC* must satisfy the *Sampling Theorem*, thus sampling at least twice the *IF* bandwidth. The latter does not have to thanks to the *Real* and *Imaginary* components which carry both *magnitude* and *phase* information.

$$R_{max} = \frac{c \cdot f_{ADC}}{2k \cdot \mu} \quad [m] \tag{1.35}$$

For what regards the *Range Resolution*, it depends only on the transmitted signal's swept bandwidth.

$$\Delta R = \frac{c}{2 \cdot B} \quad [m] \tag{1.36}$$

In order to correctly separate two *targets*, those must be separated in the *IF* in terms of their beatings. Two targets' beating difference must be larger than the inverse of the *chirp*'s duration. This is due to the fact that very close targets' echo have almost null time difference, thus the minimum frequency difference is dominated by the *chirp* duration. A graphical representation can be seen in figure 1.14.

$$\Delta f = \frac{1}{T_c} \tag{1.37}$$

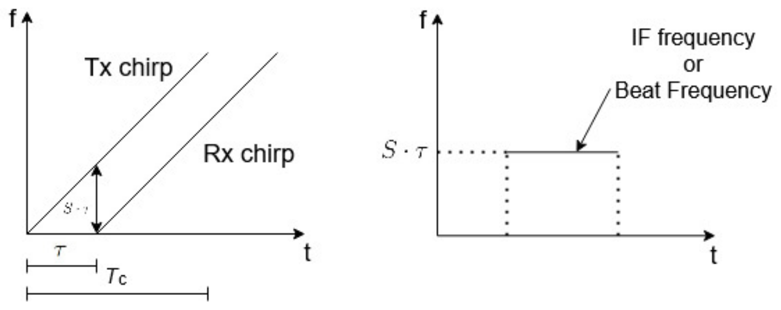


Figure 1.14: Target *Range* separation in *FMCW RADARs*.

The *Doppler* shift of a target can be acquired in way which resembles the case of the *Pulsed RADAR*. A set of chirps is transmitted and their echoes are collected and then processed. The *Doppler* effect induced by the targets act on the chirp signal phase, as shown in figure 1.15.

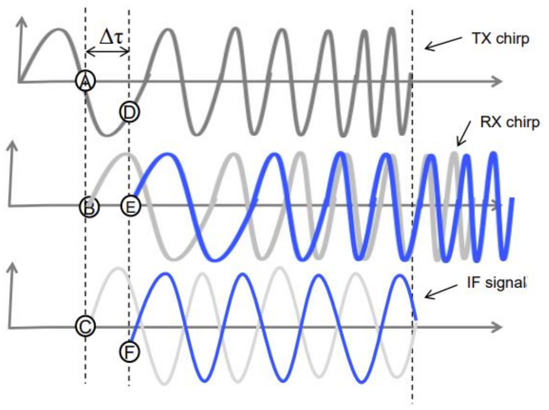


Figure 1.15: Phase changing of a small round-trip variation [5]

Thus the phase shift can be retrieved as (1.38):

$$v = \frac{c \cdot \Delta\phi}{4\pi \cdot T_c \cdot f_0} \quad \left[\frac{m}{s}\right] \tag{1.38}$$

In order to correctly determine the *Doppler*, the *Sample Theorem* must be respected, thus the *Doppler*-induced phase shift must not change more than π [rad] between the chirp observations. Thus this constraints the *Maximum Velocity* to (1.39):

$$v_{max} = \frac{c}{4 \cdot T_c \cdot f_0} \quad \left[\frac{m}{s}\right] \tag{1.39}$$

When not respected, the acquired velocity results in an ambiguous velocity. For what regards the *Velocity Resolution*, it results in an approach which resembles the *Pulsed RADAR*. Thus the chirps are used to sample the phase shift with N samples and this leads to the following formula (1.40):

$$\Delta\phi = \frac{2\pi}{N} \quad [rad] \tag{1.40}$$

Since N chirp signals are sent in order to sample the phase variation, the required time to complete the acquisition is (1.41):

$$T_f = N \cdot T_c \quad [s] \tag{1.41}$$

According to the *FFT* characteristics, the *FFT Accuracy* depends on the observation time, which ultimately leads to the *Velocity Resolution* as (1.42):

$$\Delta v = \frac{c}{2 \cdot T_f \cdot f_0} \quad \left[\frac{m}{s}\right] \tag{1.42}$$

In cases where multiple targets dwell the same *Range* and *Doppler Bins*, a further coordinate is needed. The most widely adopted coordinate for this purpose is the *Angle*. This concept can be applied to both *Azimuth* and *Elevation* angles. With this processing it is possible to determine the *Angle of Arrival* (*AoA*) of a target's echo. In order to acquire this information, more receiving chains are needed.

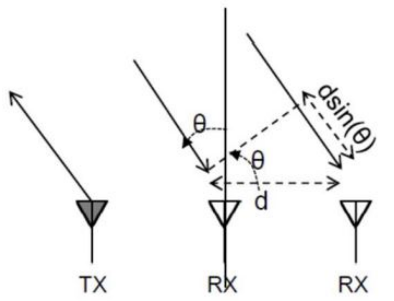


Figure 1.16: Graphical visualization of the phase difference between antennas receiving the same transmitted signal[5]

The *AoA* information of a target is contained in the phase difference between the receiving antennas. This phase difference is caused by the spacing d between the receiving antenna elements. The relationship with the *AoA* θ can be expressed as (1.43):

$$\Delta\phi = \frac{2\pi \cdot d \cdot sin(\theta)}{\lambda} \quad [rad] \tag{1.43}$$

Thus the *AoA* can be obtained as (1.44):

$$\theta = sin^{-1}\left(\frac{\lambda \cdot \Delta\phi}{2\pi \cdot d}\right) \quad [rad] \tag{1.44}$$

For small angles the formula (1.44) can be approximated to a linear model, but inaccuracy occurs for large angles. Also for this data ambiguities can occur. This happens when the phase difference between neighboring antennas is larger than π $[rad]$. Thus the target's *Maximum AoA* can be obtained as (1.45).

$$\theta = sin^{-1}\left(\frac{\lambda}{2d}\right) \quad [rad] \tag{1.45}$$

Considering the equation (1.44), its domain is $[-\frac{\pi}{2}\ ,\ \frac{\pi}{2}]$ $[rad]$. Thus it is possible to detect angles in that range, but only under the condition that the *Field of View* (*FoV*) is maximized. This condition verifies when the antenna spacing is *half-wavelength.*

$$d = \frac{\lambda}{2} \quad [m] \tag{1.46}$$

For what regards the *Angle Resolution* it depends upon the number or receiving antennas. When the *FoV* is maximized and the *AoA* can be approximated to 0, it assumes the provided form (1.47).

$$\Delta\theta = \frac{2}{N} \quad [rad] \tag{1.47}$$

1.5 MIMO Systems

The *MIMO*[6] technique is a telecommunication technology which improves the performance of a *Radio-Frequency* system given the same amount of antennas. It makes use of multiple antennas during the transmission and reception.
In order to achieve this result, the multiple transmitting antennas are exploited. The signal of each antenna is collected separately by all the receiving antennas and then combined together. This solution exploits the different transmitting antenna position, thus more phase information is collected sequentially.

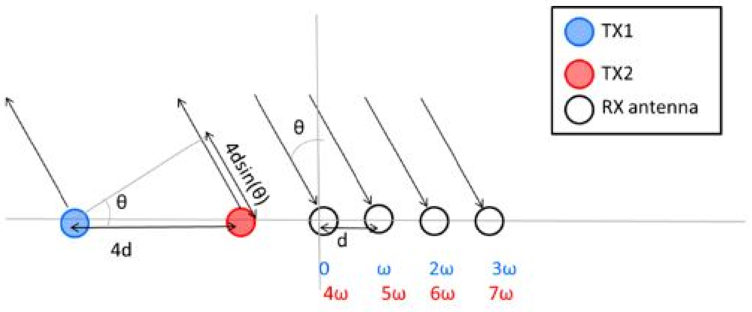


Figure 1.17: Graphical representation of the *MIMO* concept[6].

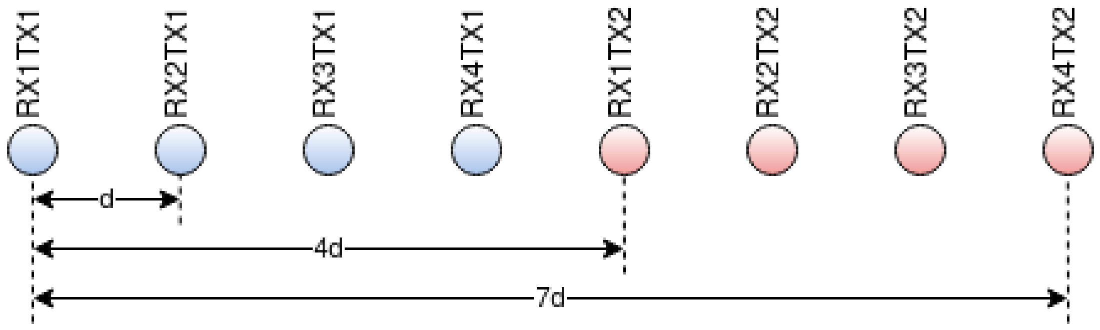


Figure 1.18: Graphical representation of the *MIMO Virtual Array* composition[6]

The combination of the data collected by the receiving antennas constitutes what is referred to as *Virtual Array*, which is equivalent to an *Antenna Array* with a larger amount of elements. The equivalent length of the *Virtual Array* (1.48) is the product of the antenna elements for the transmission (N_{TX}) and for the reception (N_{RX}).

$$N_{\text{Virtual Array}} = N_{TX} \cdot N_{RX} \tag{1.48}$$

In order to not overlap any of the *Virtual Array*'s elements, the distance in between the transmitting antennas must be as much as the distance between the first receiving antenna and the last, as noted in figure 1.17.
Furthermore in order to unequivocally collect each transmitted signal, those must be made orthogonal to each other. In order to establish the orthogonality of the transmitted signals, different techniques can be adopted: *Time Division Multiplexing* (TDM), *Carrier Frequency Multiplexing* (CFM) and *Code Division Multiplexing*, often performed with *Binary Phase Modulation* (BPM).
The *TDM* technique consist in transmitting the signals in different time slots, while

the *CFM* sends them on different carrier frequencies and the *BPM* sends them with a different phase-modulated code.

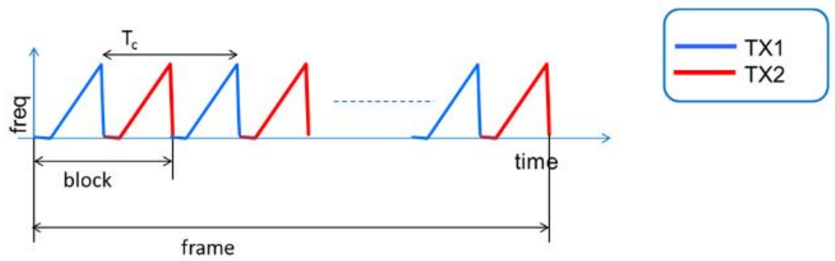


Figure 1.19: Graphical representation of the *TDM* technique with the transmitted *FMCW* chirps[6].

1.6 Minimum Redundancy

The *Minimum Redundancy* is a concept which refers to the efficient usage of the resources by avoiding redundancy and duplicates of a certain set. This concept pertains to a large number of fields, but in this specific context it refers to the possibility of reducing the amount of antennas without losses about the angle information.
The *Minimum Redundancy Linear Array*[7] is a type of *Antenna Array* whose elements are placed in a particular space arrangement. The receiving antennas adopt a spacial arrangement with unequal distance between the elements of the array. This configuration avoids the redundancies, thus providing the possibility to rearrange the redundant antennas. By doing so it is possible to increase the array length, thus improving its performance.
An *Antenna Array* composed by N elements can be arranged in order to obtain an equivalent linear array of N_{MR} elements (1.49).

$$N_{MR} = \frac{1}{2}N(N-1) \tag{1.49}$$

The *Minimum Redundancy* concept applied to four receiving antennas leads to the arrangement shown in figure 1.20. The *Virtual Array* is obtained after the correct phase processing is applied in order to extract the antennas which are missing.

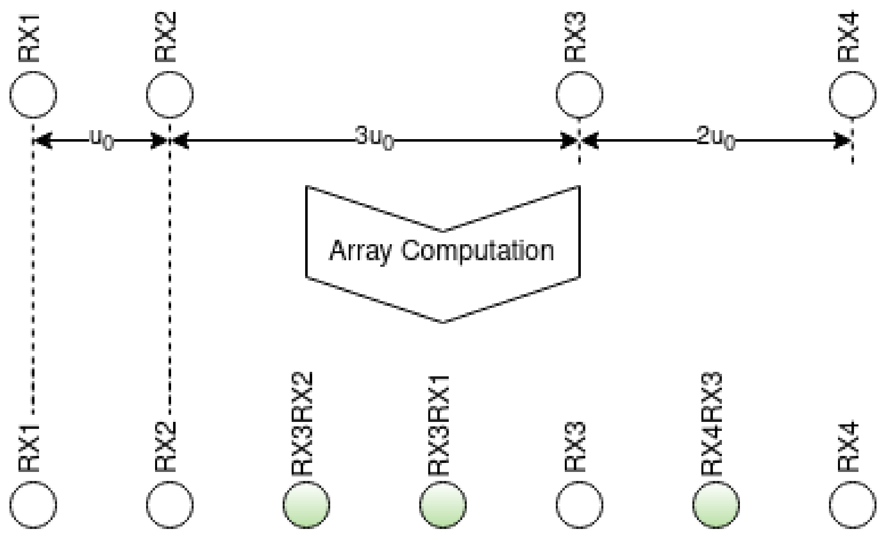


Figure 1.20: Graphical visualization of *Minimum Redundancy* structure applied to an *Antenna Array* of four elements. The spacing unit u_0 is the responsible for the *Array Phasing.*

The *Minimum Redundancy Linear Array* concept can be exploited also in *MIMO* configurations, while providing better *main-lobe* interference rejection at a cost of a negligible *side-lobe* interference rejection's degradation[8].

1.7 RANSAC Algorithm

RANSAC[9] is a statistical algorithm commonly adopted in computer vision and image analysis. It is an iterative algorithm which estimates the parameters of a mathematical model from a set of observed data points that contains both *inliers* and *outliers*. The former are considered to be the correct points used to fit the model, while the latter are considered to be the points affected by gross errors and thus should not be used for the model fitting purposes. The *RANSAC* algorithm is able to filter out the outliers by iteratively taking a random chosen subset of points from the main data set. This set is then used to fit the mathematical model to the data subset, and then verifying its adequacy in terms of error with respect to the remaining data points. This process ends when the best fit is found or the limit of attempts is reached.

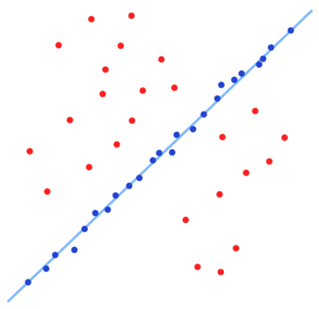


Figure 1.21: Example of RANSAC fitting a line[10].

Chapter 2

The AWR1843 Chip

This chapter presents the *SoC* system on which the software has been tested and validated. The routine has been designed in order to abstract the *Hardware* architecture in order to be more flexible, but considerations on the *Hardware* are still needed for its integration in the *Radar Software Stack* for Testing and Validation purposes. Thus the *SoC* structure is presented.

2.1 SoC Architecture

The AWR1843 *SoC* is a single-chip *Radar* solution which implements most of the *Radar* hardware on the same chip. It implements the *Radio-frequency* front-end, the signal processing module with a dedicated *C674x DSP* and a general-purpose *Cortex-R4F Core.*
These units can be analyzed in their respective *Macro-Block* context. As can be seen in figure 2.1, three main *functional blocks* can be identified:

- RADAR Subsystem which contains the *RF Front-end* and the *Radio Processor*
- DSP Subsystem which contains the *DSP*
- Master Subsystem which contains the *Cortex-R4F Core*

As can be seen in figure 2.1, the only missing *Hardware* part on this specific chip is the *Antenna Array.* Other chips also implements *Antenna on Package* solutions.

The AWR1843 chip is an *Automotive Radar* chip which complies to the latest regulations for *Automotive Radars.* It can operate in the frequency range 76-81 GHz, which enables it to be employed for *Short-Range Radars (SRR)* and *Long-Range Radars (LRR).* The former have the operating bandwidth 76-77 GHz, the latter the 77-81 GHz.

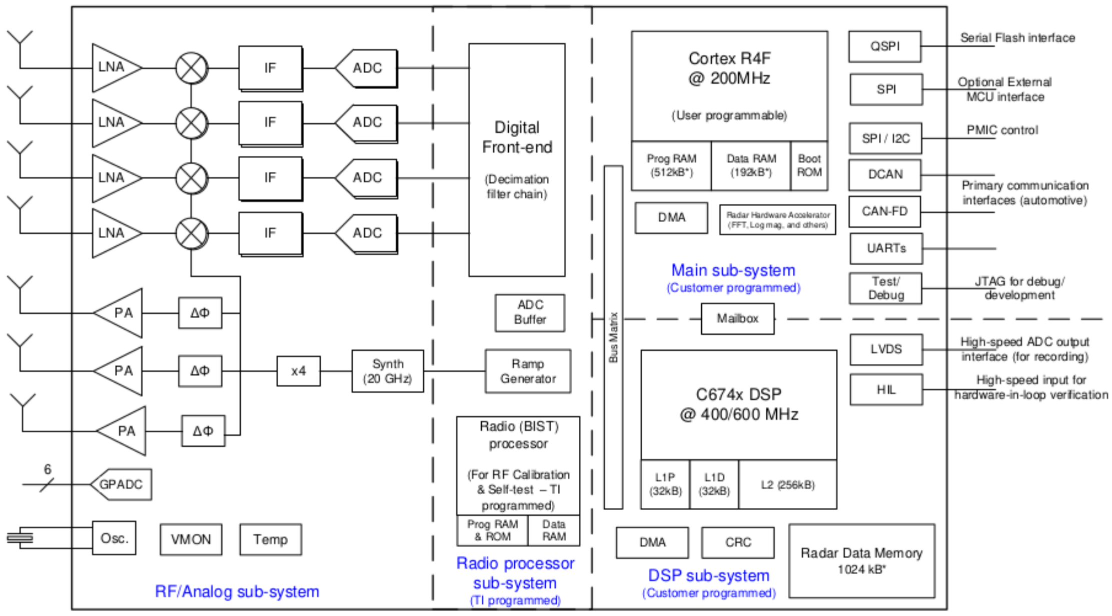

LNA
IF
ADC
PA
ΔΦ
x4
Synth (20 GHz)
GPADC
Osc.
VMON
Temp
RF/Analog sub-system
Digital Front-end (Decimation filter chain)
ADC Buffer
Ramp Generator
Radio (BIST) processor (For RF Calibration & Self-test – TI programmed)
Prog RAM & ROM
Data RAM
Radio processor sub-system (TI programmed)
Bus Matrix
Cortex R4F @ 200MHz (User programmable)
Prog RAM (512kB*)
Data RAM (192kB*)
Boot ROM
DMA
Radar Hardware Accelerator (FFT, Log mag, and others)
Main sub-system (Customer programmed)
Mailbox
C674x DSP @ 400/600 MHz
L1P (32kB)
L1D (32kB)
L2 (256kB)
CRC
DSP sub-system (Customer programmed)
Radar Data Memory 1024 kB*
QSPI
SPI
SPI / I2C
DCAN
CAN-FD
UARTs
Test/ Debug
LVDS
HIL
Serial Flash interface
Optional External MCU interface
PMIC control
Primary communication interfaces (automotive)
JTAG for debug/ development
High-speed ADC output interface (for recording)
High-speed input for hardware-in-loop verification

Figure 2.1: AWR1843 Functional Block Diagram[11]

2.1.1 RADAR Subsystem

The *RADAR Subsystem* is the chip's subsystem where the *RF Front-end* and the *Radio Processor* are contained. As can be seen in figure 2.1, the *RF Front-end* contains the *Analog* portion of the *Radar* transmitting and receiving chain. In this structure the *SoC* packs the three transmission chains and the four transmission chains. This schema allows the *Radar* using this chip to operate in a *MIMO* configuration.
This subsystem exposes an *API* used to let the *Master Subsystem* configure and manage it.

2.1.1.1 Frequency Synthesizer

The frequency synthesizer is used by the *RF Front-end* to generate the *FMCW* chirp. In order to do so, it takes the input clock reference and, by means of a *Clean-Up PLL*, it filters the external clock. This filtered clock is also used to provide the *SoC* clock signal. The *Timing Engine* is used to drive the *RF Synthesizer* which generates the *Local Oscillator* reference for both the *Transmitter* and *Receiver*. The *Timing Engine* also provides the synchronization signals, such asr the receivers' *ADCs* controls and transmitters' phase modulation. In figure 2.2 the block diagram shows the internal structure:

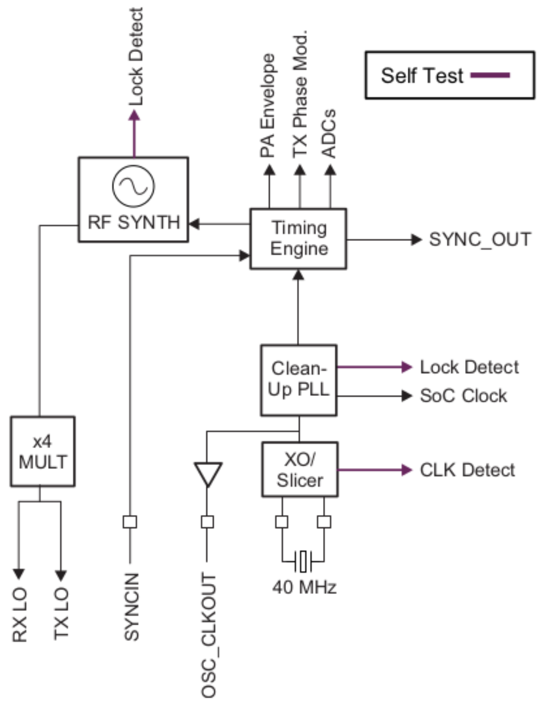


Figure 2.2: AWR1843 RF Clock Management[11]

2.1.1.2 Transmitter Chain

The transmitter chain is the block which is used as antenna driver. It takes the *Local Oscillator* signal and increases its power through the usage of a set of *RF Power Amplifiers*. This chain has also the possibility to introduce a configurable phase modulation to the *Local Oscillator* signal before increasing its power. The *RF Power Amplifiers* are then interfaced to the *SoC* output pins. A total of three transmitter chains are available on this *SoC*. In figure 2.3 the block diagram shows the internal structure:

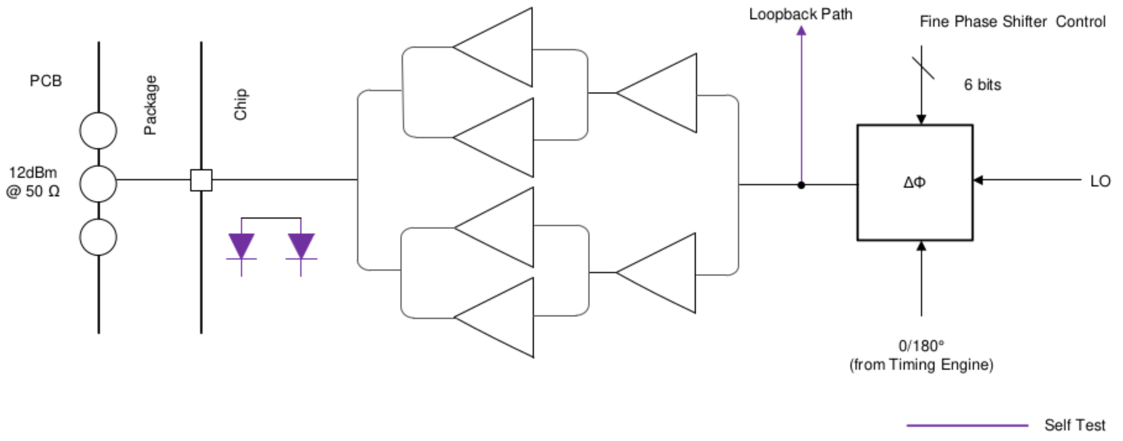


Figure 2.3: AWR1843 Transmitter Chain[11]

2.1.1.3 Receiver Chain

The receiver chain is the block which is used to acquire the data from the receiver antenna. It takes the *RF* signal from its channel's antenna, it amplifies that signal by using a *Low Noise Amplifier* and then it performs *I/Q Mixing*. Each *I/Q* channel has its *IF* filtered and sampled with its respective *ADC*. After the *ADC* acquisition, the data is decimated, corrected and then saved into its respective buffer. In figure 2.4 the block diagram shows the internal structure:

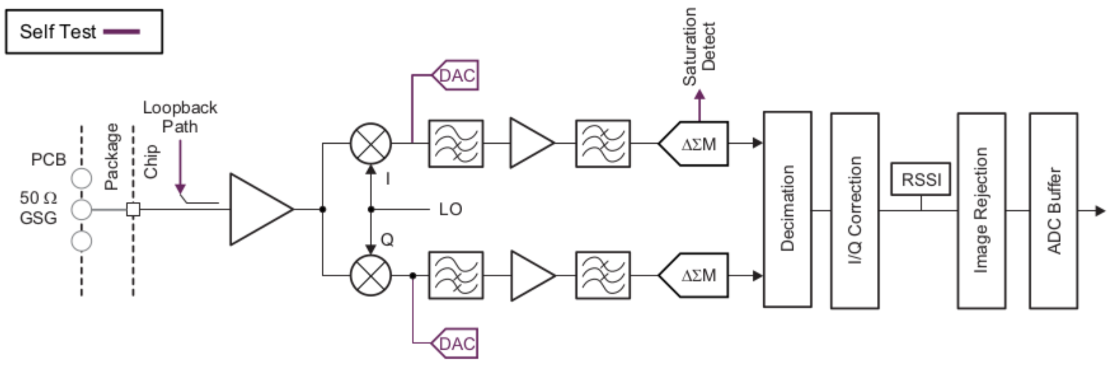


Figure 2.4: AWR1843 Receiver Chain[11]

2.1.2 DSP Sub-System

The AWR1843 *SoC* integrates in its architecture a *DSP* which belongs to the *Texas Instrument*'s family of *C674x DSPs*. This *DSP* has a *VLIW* micro-architecture, with both *Fixed Point* and *Floating Point* capabilities. It runs at 600 MHz and has two independent *data paths* which leads to a total of six *ALUs* and two *Multipliers*, with the capability to fetch up to eight instructions per cycles. In order to satisfy the performance requirements for this hardware unit, a multi-level *RAM* architecture is employed, where the farthest and largest level is shared with the *Master Sub-System*. Thus a complex *BUS* architecture has been adopted, as can be seen in figure 2.5:

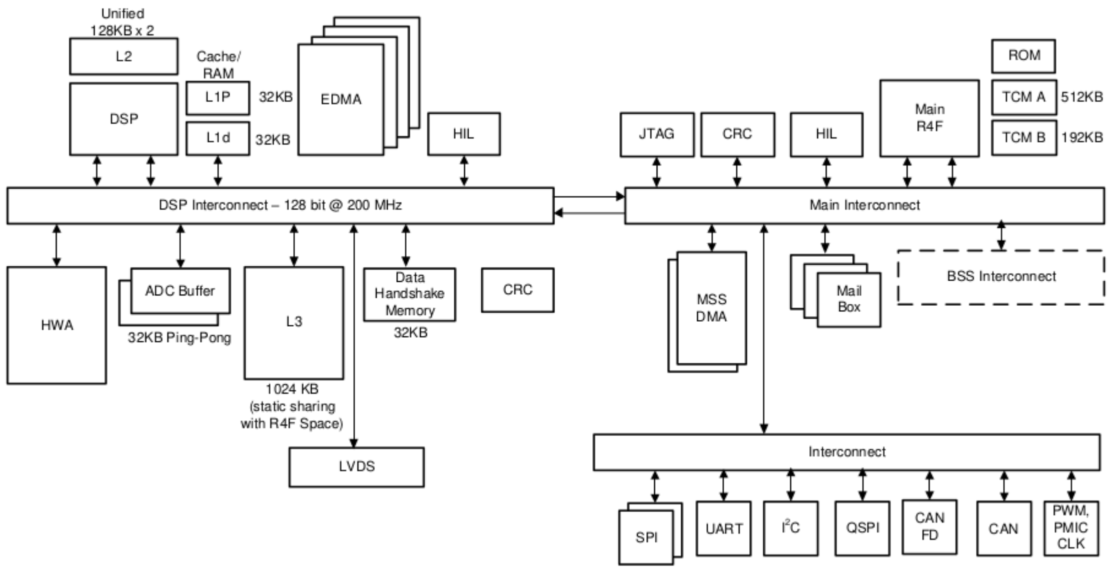


Figure 2.5: AWR1843 DSP and Master subsystem *BUS* interfaces[11]

2.1.2.1 Extended Direct Memory Access Unit

The *EDMA* is the hardware unit which is used to move data between peripherals in order to relieve the *DSP*, and possibly also the *Cortex-R4F Core*, from the heavy task of moving data. In this way it is possible to invest the computational time in a more efficient way. This means that data can be transferred while performing other calculations. It can be used in a *Radar Software Stack* in order to move the data from the *ADC* buffers into the *RAM* memory for later calculations, while calculating a previously transferred *ADC* buffer content.

2.1.3 Master Sub-System

The AWR1843 *SoC* integrates in its architecture a programmable *Cortex-R4F Core* which its main role is to manage the *SoC* and the chip's interfaces to the external world. It requires lesser performance than the *DSP*, thus it is clocked at 200 MHz and it accesses directly the 1 MB main memory in a single-level schema. The main memory is shared with the *DSP Subsystem*. For what regards the peripheral access it adopts a peripherals memory mapping schema, like a *micro-controller*. In order to let the *Cortex-R4F Core* to manage the whole *SoC*'s hardware resources, a complex *BUS* architecture has been adopted, as it is shown in figure 2.6:

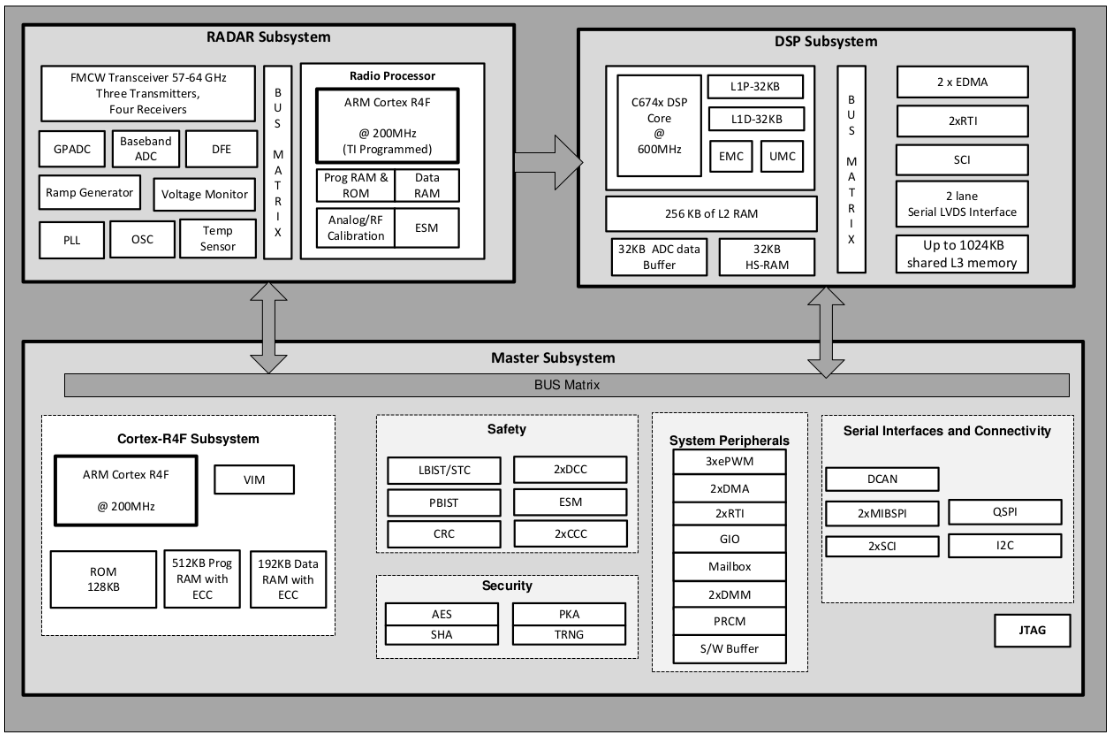


Figure 2.6: AWR1843 RADAR, DSP and Master subsystem[12]

As can be seen in figure 2.6, this *SoC* integrates a lot of peripherals, *Hardware Accelerators* and controllers. This is justified by the fact that *Radar* systems are a class of systems which have to manage a lot of data while performing heavy computation. Furthermore also the peripherals must be managed together with all the other hardware resources. Thus all this architectural complexity aims to drastically increase the performance capabilities of the AWR1843 *RADAR SoC*.

Chapter 3

Software Design and Development

This chapter presents the process flow which has been adopted in order to design and develop the *Ego Velocity* estimation routine, taking into account various *Software* evaluation figures during the whole process. The design takes into account how a *Radar* collects the data to be processed, with particular care on how the system used as test-bench was designed. Starting from those considerations, the routine has been made more general and independent from the platform which adopts it. The design and development are then followed by the Technical Overview specific to egoVelocityEstimLSOW, after which the most appropriate changes to the initial *Software* implementation has been made, until a satisfactory result has been obtained. Thus, this *Design and Development* process is strongly driven by the results.

3.1 Software Overview

The *Radar Software Stack* which has been used as development platform is *Radar* system for a car lamp, for which a research effort has been already done. It provides an advanced *Radar* processing environment, exploiting multiple sub-frames and chirp parameters in order to increase the maximum unambiguous Doppler velocity, perform Doppler-Phase compensation needed by the TDM technique, while exploiting a minimum redundancy MIMO array to increase angular resolution. Thus, it provides an interesting processing context to be adapted in order to make that system able to extract the *Radar Ego Velocity*, without the need for a third-party system to inform it.
The main issue which involves the *Ego Velocity* is its effect on the Tracking Algorithm, which is discussed in section "Tracking Algorithm Issue".

3.1.1 Acquisition Schema

The RADAR Subsystem sends, during an acquisition sub-frame, a set of chirp signals over its transmitting chain in a time-division channel multiplexing schema. This allows the receiving chain to collect simultaneously a chirp on all the receiving antennas. The DSP Sub-System is in charge of processing the raw data acquired by the receiving chain. Double-buffering is exploited in order to have enough time to process a chirp Range-FFT on Fast Time coordinate, while receiving the next one. Once all the chirps of a sub-frame have been collected, it is possible to perform the Doppler-FFT on the Slow Time coordinate. After this step the velocity is known, but not yet disambiguated.

3.1.2 Detection Schema

The actual detection of targets is performed in three steps: first the detection is performed using the Range coordinate, then a later detection using the Doppler coordinate and finally on the Azimuth coordinate. This is achieved by performing a *CFAR* on each coordinate used to separate the measured points.

3.1.3 Disambiguation Schema

The velocity disambiguation is performed after its dedicated sub-frames collection. Thus by applying the "Chinese Remainder Theorem" it is possible to determine the velocity of a given point because it has different ambiguous values for each chirp profile. For this sub-frame a single transmitting antenna is used in order to avoid the phase error introduced by the time-division multiplexing of a full MIMO array. In case it is possible to find a common multiple within a reasonable velocity tentative span, the velocities are disambiguated. Thus it is also possible to compensate the Doppler-induced phase error on all the receiving antennas of the previous frame.

3.1.4 Association Schema

The association between the targets of different sub-frames is performed in terms of velocity and minimum distance. During the MIMO sub-frame acquisition, the points corresponding to a peak in the detection are defined as the master of that group, while the other nearby points which have simply crossed the CFAR threshold are classified as their closest master's slaves. Then, once the velocity has been disambiguated for each master in the dedicated sub-frame, it is associated to its corresponding master from the MIMO sub-frame by assigning the disambiguated velocity to all of its group's points.

3.1.5 Tracking Algorithm

The tracking algorithm is in charge of combining a sequence of targets observations into the most appropriate set of tracks. So it is possible to monitor targets evolution across multiple physical dimensions like time, position, velocity, but others exist. The tracking algorithm needs a sufficiently large set of measurement points in order to have reliable result and performance. Thus, it is important to have a large set, but even more important that those points' measure is accurate in order to have an accurate tracker behavior. This is why the TDM Doppler-Phase error must be compensated, so the spatial position is not affected by this error.
This particular *Tracker* is implemented using a Kalman filter on the measured point group. It associates a moving group of points to the closest track in terms of distance and updates the next prediction. If instead no close tracks are present a new one is pre-allocated, ready to receive the later observations . A track is instead freed if the tracked *Target* disappears, which means it either reached a region outside the field of view or it stopped.
The following frames are the ones presented as the snapshot for the "Static host, closing target: Track Start", in which a track allocation can be seen. Due to the *SPI* data-rate limit, the shown targets are the one for the second subframe, used for the "*Disambiguation*" and for the "*Ego Estimation*". Only the static points, the masters and ??? can be seen. The amount of targets in the MIMO sub-frame is too large to be transferred, thus the grouping effect of the *Tracking* algorithm cannot be displayed.

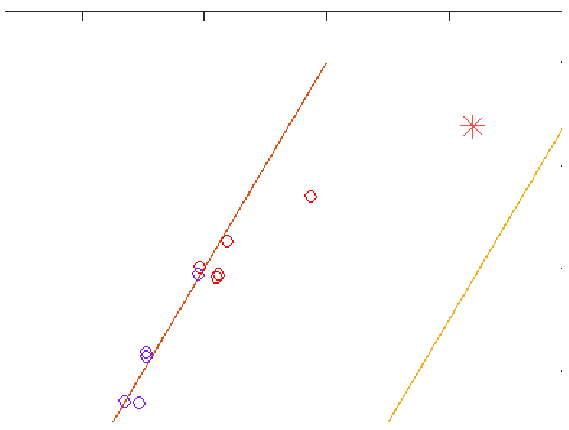


Figure 3.1: Initial detection of a Master during frame 53. A master which can start a track is represented with the red asterisk, the purple circle are the almost static targets and the red circles are the static targets

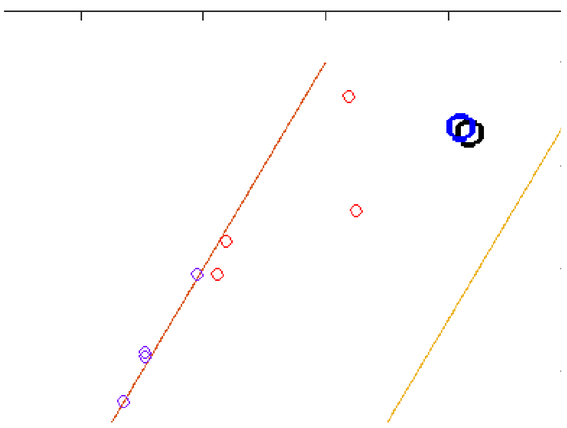


Figure 3.2: Track allocation for the group during frame 54. The leading master is represented with the blue circle, the track with the black circle and the sub-frame slaves as green circles

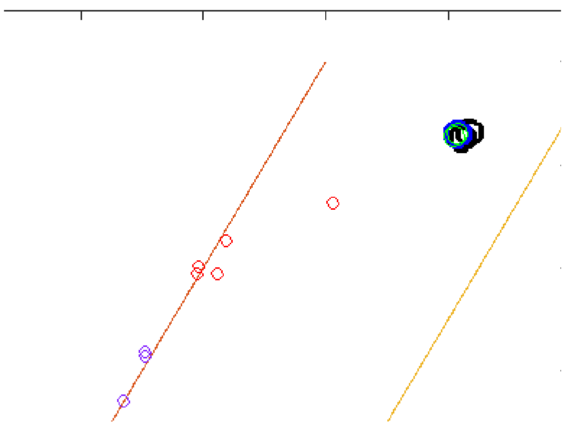


Figure 3.3: First track update after allocation, during frame 55. The green circles are the master's slaves of the *2nd* sub-frame

3.2 Tracking Algorithm Issue

The tracking algorithm is thus used to track and log the evolution of moving targets. If the *Radar* itself is moving because it is mounted on a moving host, also the static targets are now moving. This largely increase the computation and memory resources needed in order to process all the moving targets, whether being true targets or the scenery. In order to overcome this situation, the tracker needs a system far more performing than needed or to process lesser points, thus reducing the tracking capabilities. The optimal solution is to have the largest set of points and just process the *real* targets. This solution can only be implemented if the *host* velocity is known. Once said velocity is known, the points which have the same velocity as the *Ego Velocity* can be ignored by the tracking algorithm.

3.3 Ego Velocity Estimation Design

The routine which is in charge of estimating the Radar's *Ego Velocity* has been designed and developed in order to be interfaced with an already-working software stack. The way this estimation routine has been designed is to be flexible enough so that the usage in this specific stack does not limit its fields of employment, but rather present a possible application in situations where the other *Radar Routines* already exist, thus being already verified and validated. So this routine simply extends a *Radar Software Stack*.
This routine's design takes inspiration on the previous work of Dominik Kellner, Michael Barjenbruch, Jens Klappstein, Jürgen Dickmann and Klaus Dietmayer. As shown in their article[1] various possibilities have been explored, but the actual proposed solution is the application of a *RANSAC* algorithm. The RANSAC algorithm is used to fit a model for a set of data. This is its most common usage, as proposed in the work of Martin A. Fischler and Robert C. Bolles. The authors propose this algorithm in an article[9], where it is used to fit a model from a set of data in order to solve a *Location Determination Problem*. By applying the concepts behind RANSAC, this algorithm can be adopted for different purposes.
By analyzing both these works, the algorithm can be reduced to an estimator function which extracts the statistical *Mode* of a set of values. From this perspective, four possible estimator function implementations have been derived from the original concept and presented in the section "Ego Velocity Estimation Development". Starting from an initial implementation, all the later ones have been developed as an evolution of that, as a consequence of analysis which involves the quality of results, computational cost, memory footprint and timing constraints. Each test and result evaluation have been analyzed and are presented in section "Test Campaign".

3.3.1 Ego Velocity Estimation: Requirements

This *Software* routine performs the *Radar's Ego Velocity Estimation*. This estimation is based on the detected targets' characteristics. Thus this function needs a cloud of measured targets.
For what concerns the targets, those must have the following physical dimension fields:

- Radial Velocity (represented as a *single-precision floating point number*)

- Spatial Position (a set of Cartesian coordinates where all its members are represented as *Qx fixed point number*)

Furthermore, in order to also match the flexibility goals introduced in the "Ego Velocity Estimation Design" section, the routine implementations have been designed considering that the cloud of measured target data has its memory layout provided "as is" by the already existing *Radar Software Stack*. Thus the *Ego Velocity Estimation* must satisfy the following requirements, in order to meet the desired level of *flexibility* and *scalability*:

- use an abstraction layer in order to access targets' data fields
- the actual algorithm implementation can be chosen at compile time for each call

This set of *Requirements* is then translated into an implementation, whose architecture is discussed in Ego Velocity Estimation: Architecture Overview.

3.3.2 Ego Velocity Estimation: Architecture Overview

The routine is organized as a self-contained procedure. This is achieved by encapsulating its code in a new function, then ensuring that the input data accesses are not destructive and that the output data is written only once. Thus this means that each implementation accesses the cloud of target points only to read them and only after the *Ego Velocity* has been estimated, it is written into a dedicated data structure.
This data access schema ensures that this routine can be optionally integrated and scheduled into any *Radar Software Stack*.

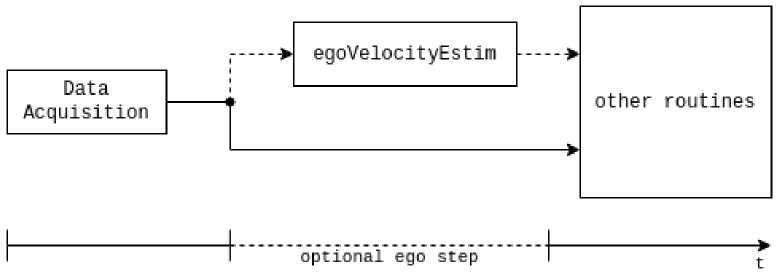


Figure 3.4: Ego estimation scheduling visualization.

In order to achieve the *flexibility* and *scalability* goals, the abstraction layer is used inside the *Ego Velocity* estimation routine as the only way to read-access external data. Thus the routine is able to access the target physical dimension fields regardless of the actual target data layout. This layer makes possible the data retrieving from various list of suitable points, whether from a raw *Radar Cube*, a list of points or even post-processed targets. These data access methods are provided in their dedicated library. Furthermore, this interfacing library can be overridden at compile time so that the *Ego Estimation* can be integrated into other projects without the need to actually modify the interface library content or even the routine's code. This design choice provides also an increased *portability* to other *Radar Software Stacks*.

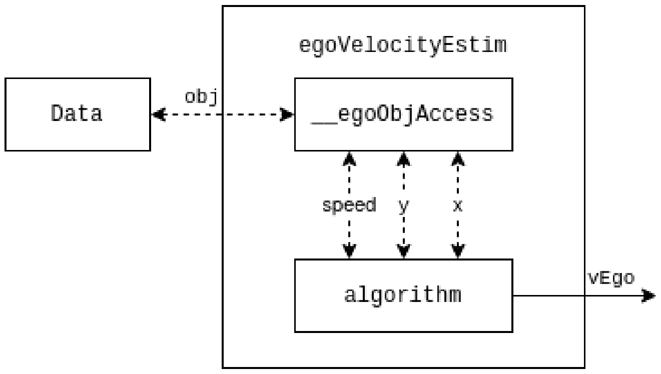


Figure 3.5: Ego estimation data access visualization.

The usage of said layer allows also an homogeneous input data access for the algorithm, without the need to copy said data into a different data structure or even to change the routine's *code-base* whenever a different data representation is used. In this way, lesser memory and lesser *code* edits are needed.

3.3.3 Ego Velocity Estimation: Technical Overview

The *Ego Velocity* routine is present in various versions, but all have the same exposed API entry-point. This Entry Point is provided as a *C macro*. Its first argument specifies the actual implementation to use for that algorithm instance, while the others are just the arguments for the estimator algorithm itself. This is possible due to the fact that every implementation has been designed to have an homogeneous interface, thus having the same set of arguments and using the same names and symbols. This API relationship with respect to its possible implementation can be seen as the one proposed in figure 3.6.

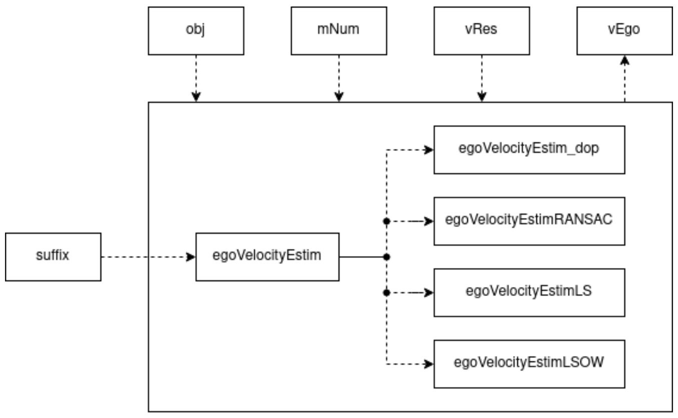


Figure 3.6: Visual representation of an algorithm instance's implementation choice.

The abstraction layer consist in a set of *C macros* that describe how to access a single target's field, given the target's memory address. A set of macros is provided in a dedicated library, where the methods used to access the target's velocity and Cartesian position coordinates can be found.

3.4 Ego Velocity Estimation Development

The development of a proper *Ego Velocity Estimation* routine started from an initial and straight-forward approach towards its implementation. After the routine has been implemented, it has been tested and validated. Whenever the result were not satisfactory, the routine underwent to another development stage iteration. This development iterations have been done until a stable, reliable and performing algorithm has been found. Due to this iterative approach, a total of four *Ego Velocity* estimation algorithm versions have been implemented. The result-driven iterative nature of this process introduces a new set of requirements to each new implementation. All of the implementations can be recognized because they have the same radix name and differ by the suffix, which hints about their actual algorithm. Since the *Radar* measures all the points using its bore-sight as the reference axis, all the algorithms consider also the *Radar*'s host direction and project all the points' velocity to this axis during the respective evaluation step.

3.4.1 The egoVelocityEstim_dop routine

This *Ego Velocity Estimation* routine was the first approach towards the problem. It is implemented as a voting algorithm which bases its vote on the amount of occurrences among the measured points.

This algorithm's concept is backed by the statistics of a *Radar* acquired frame: every frame consist of a set of targets inside a crowded cloud of points, which most of them belongs to the environment.
This means that the set of target points is lesser than the scenery points and these latter ones exhibit a velocity which is like the *Host*'s velocity, but projected along their position. Thus this means that the *Mode* is the statistical parameter which better represents the points belonging to the environment. Furthermore, the projection of all the points back to the *Host* direction is necessary because along that axis resides the *Ego Velocity*.

Since this routine is a specific implementation of the *egoVelocityEstim* estimation algorithm, it inherits the requirements laid out in "Ego Velocity Estimation: Requirements" while introducing a new set which is tightly related to this particular algorithm implementation.
For what regards the actual algorithm implementation for this routine, it is laid out in the "Technical Overview specific to egoVelocityEstim_dop" section below.

3.4.1.1 Requirements specific to egoVelocityEstim_dop

This routine has been developed in order to vote as the *Ego Velocity* the one which occurs the most in a given frame.
The reliability of this algorithm strongly depends on the statistics of a given acquired frame, which is later discussed in the "Further Considerations" section. The obtained result states that this implementation is not suitable for target-dense scenarios or environments with targets which produce largely-sparse return echos, which can shadow the *Clutter*.

3.4.1.2 Technical Overview specific to egoVelocityEstim_dop

The algorithm is structured in order to log the amount of points, from the given list, which have a certain velocity. This goal is achieved by logging a new record whenever a not yet recorded velocity has been found, and by incrementing the respective velocity's occurrence counter whenever a previously recorded one is found. Then it ultimately reports the velocity which has the highest count in its associated occurrence counter.
Due to the structure of this algorithm, the computational cost and the memory accesses scale as $\mathcal{O}(n^2)$. The required temporary memory scales instead as $\mathcal{O}(n)$. Another nice feature is that the loop kernel is rather small, compared to the other implementations, with few and simple calculations. Furthermore it also has a more consistent timing and performance since it always use all the given targets.
Thus these characteristics can be resumed as follows:

PROs	**CONs**
+ Small loop kernel with few simple instructions allows for fast iteration execution	- Large computational and memory access cost model
+ More deterministic algorithm behavior	- Multi-modal point sets can cause mis-evaluation
+ More consistent timing performance	- Target-dense environments make the *Mode* shift

In figure 3.7, the graphical representation of the algorithm flow is provided.

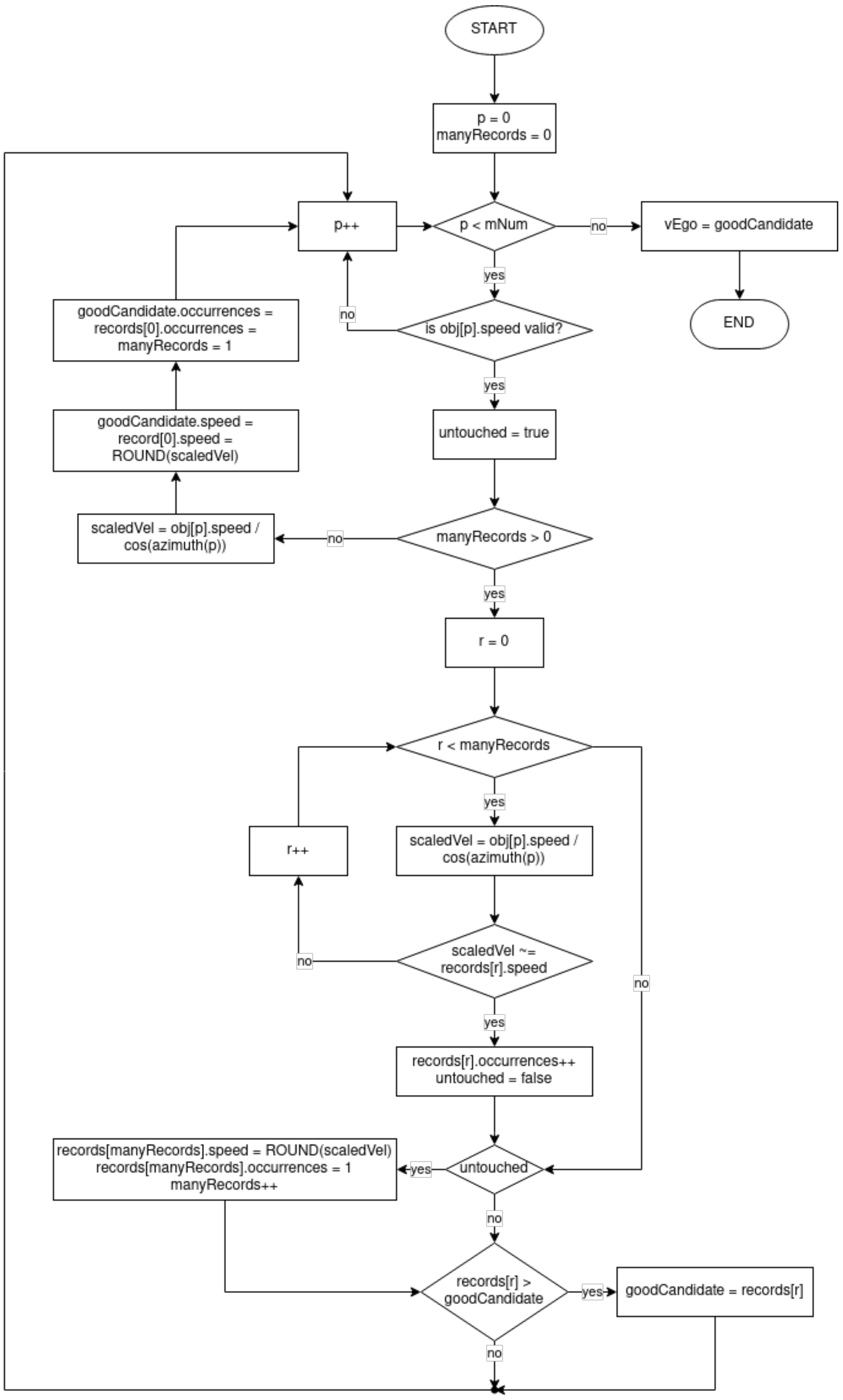


Figure 3.7: Ego estimation "_dop" algorithm flow.

3.4.2 The egoVelocityEstimRANSAC routine

This routine follows more closely the implementation proposed in the article[1]. This *Ego Velocity Estimation* routine is implemented as a voting algorithm which bases its vote on the amount of error of a given point with respect to the set of points. The *RANSAC* algorithm bases its effectiveness to the statistics of a distribution of points. It is done by taking a sample made as a subset of points randomly picked from the larger set. This new set can still be meaningful if the points of the smaller one are taken randomly from the larger one.
By doing so, the smaller set has a certain probability of having its *Mean* velocity converge to the *Mode* of the larger set. In order to increase that probability, the estimation is performed more than once, finally picking the *Mean* velocity of the subset that produced the smallest error. This implementation has been developed in order to mitigate the target shadowing which affects the "egoVelocityEstim_dop" implementation, while trying to reduce the computational cost model.

Since also this routine is a specific implementation of the *egoVelocityEstim* estimation algorithm, it inherits the requirements laid out in "Ego Velocity Estimation: Requirements" while introducing a new set which is tightly related to this particular algorithm implementation.
For what regards the actual algorithm implementation for this routine, it is laid out in the "Technical Overview specific to egoVelocityEstimRANSAC" section below.

3.4.2.1 Requirements specific to egoVelocityEstimRANSAC

This routine has been developed in order to vote as the *Ego Velocity* the one which best fits the distribution of a subset of points, taken randomly from a larger set. This algorithm relies to the fact that most of the points are inliers, which means that they belong to the scenery, being the scenery the most populated group of points. Thus, the sub-set has a high probability of being largely composed by inliers. For what concerns the fitting, the proposed fitting is a uni-dimensional *least square error* on the points' velocity, which leans towards the distribution's *Mean* velocity. The estimation is reliable if the subset of points is largely populated by inliers, hence the *Mean* value leans towards the *Mode* of the point gathering. This constraint limits its flexibility because the actual performance depends on the pseudo-random number generator and on the points' distribution characteristics.

3.4.2.2 Technical Overview specific to egoVelocityEstimRANSAC

The algorithm is structured in order to pick a random set of points from the given list of targets. Then, it proceeds to evaluate which one of these selected points has the least velocity distance with respect to the others. Many attempts are

performed by trying to take a different set of points for the sub-set. The point which accumulated the least amount of error has then its velocity reported as the chosen one.
Due to the structure of this algorithm, the computational cost and the memory accesses scale as $\mathcal{O}(m * k^2)$, where m is the total amount of *RANSAC* trials and k is the amount of points which are picked for the sub-set. The required temporary memory scales instead as $\mathcal{O}(k)$.
While this implementation relies on a loop kernel which is not so large, it packs a more complex calculation than the "*egoVelocityEstim_dop*" implementation. Furthermore this algorithm has the potential of substantially cut down the actual number of iterations due to its heuristic approach on a reduce set of data. However it has timing issues due to the random number generation.
The random number generation has no guarantee that it won't ever generate a number for a point which was already taken, thus forcing it to try to generate another one. The extreme case is that the random number generator will generate an infinite sequence of numbers which index to already chosen points. Furthermore, the larger the amount of points for the subset, the higher the probability to select an already chosen point.
This result in a *Real-Time error*, which means that the *Radar* fails to end all the frame processing during the time slot which is allocated for this purpose.
Thus these characteristics can be resumed as follows:

PROs	**CONs**
+ Total number of iterations can be optimally cut down	- Not consistent timing performance with possibility of timing failure
+ Computational and memory access cost model lesser than $\mathcal{O}(n^2)$	- Subsets can be composed all by nearby outliers
	- Target-dense environments increase the probability to pick outliers

In figure 3.8, the graphical representation of the algorithm flow is provided.

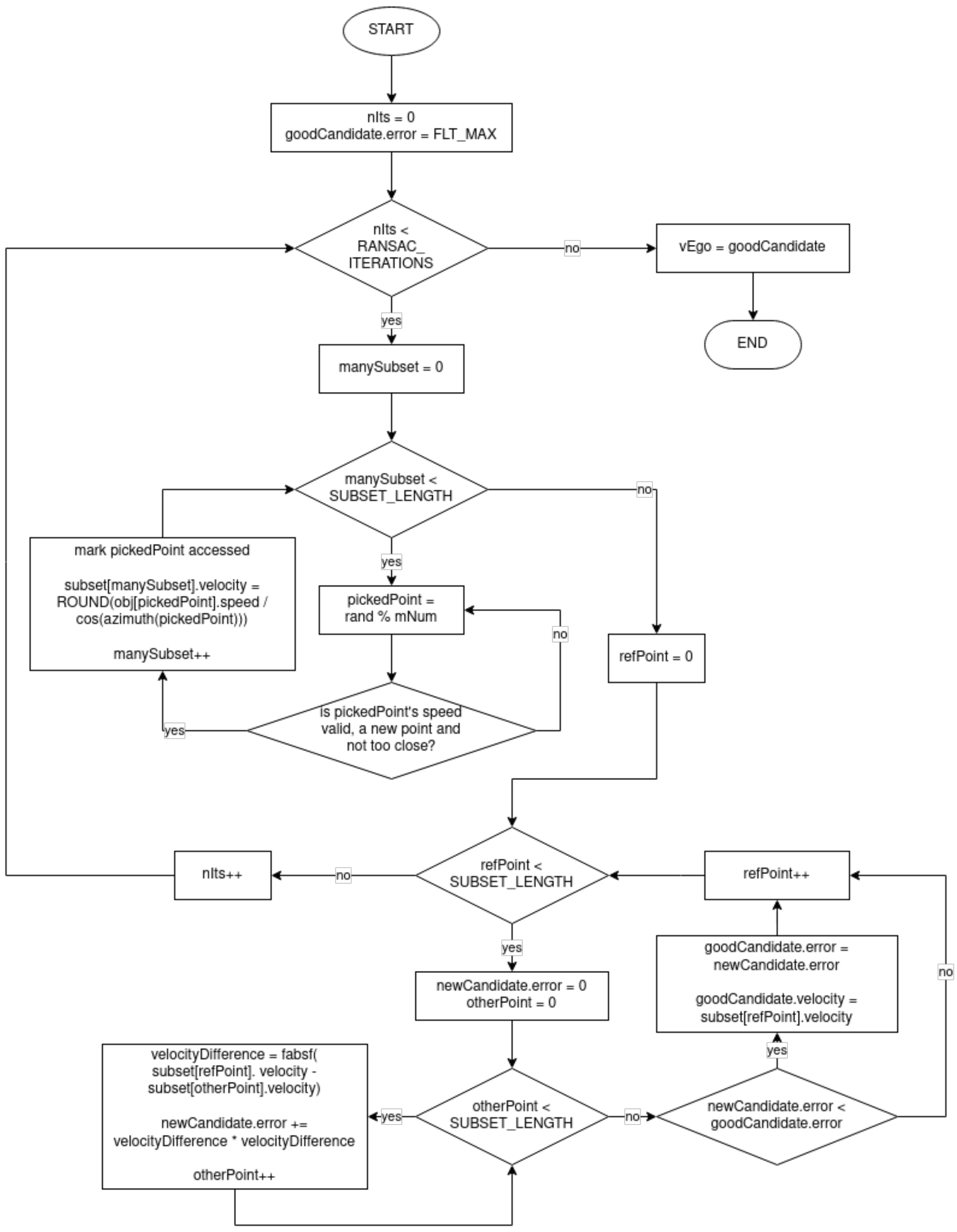

START
nIts = 0
goodCandidate.error = FLT_MAX
nIts < RANSAC_ ITERATIONS
no
vEgo = goodCandidate
END
yes
manySubset = 0
manySubset < SUBSET_LENGTH
no
yes
pickedPoint = rand % mNum
no
refPoint = 0
mark pickedPoint accessed
subset[manySubset].velocity = ROUND(obj[pickedPoint].speed / cos(azimuth(pickedPoint)))
manySubset++
yes
is pickedPoint's speed valid, a new point and not too close?
nIts++
no
refPoint < SUBSET_LENGTH
refPoint++
yes
newCandidate.error = 0
otherPoint = 0
goodCandidate.error = newCandidate.error
goodCandidate.velocity = subset[refPoint].velocity
no
yes
velocityDifference = fabsf(subset[refPoint]. velocity - subset[otherPoint].velocity)
newCandidate.error += velocityDifference * velocityDifference
otherPoint++
yes
otherPoint < SUBSET_LENGTH
no
newCandidate.error < goodCandidate.error

Figure 3.8: Ego estimation "RANSAC" algorithm flow.

3.4.2.3 Random Number Generation mitigation for the egoVelocityEstimRANSAC

The issues about the random number generation which were addressed during the "Technical Overview specific to egoVelocityEstimRANSAC" section, hints that the actual targets access schema can be re-arranged in order to mitigate the possibility of picking an already chosen point. In order to remove this possibility, a temporary data structure must be introduced. This data structure consist in a tracker array made by an empty double-linked list element. This double linked list has an amount of elements equal to the amount of targets. Thus this array has as an address offset correspondence with the array of targets.
Then whenever a target has to be selected, the obtained random number is used to unroll the same amount of elements in the double linked list. Once reached that number, the double-linked-list element's address is used to calculate the equivalent offset for the target list. After picking that target, the taken element from the double-linked-list is removed by making its respective previous element's *next* pointer take its *next* pointer, while its respective next element's *previous* pointer take its *previous* pointer.
The following figures 3.9 and 3.10 shows the structure usage:

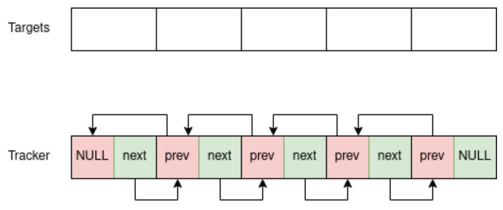


Figure 3.9: The double linked list elements amount is the same for the target list.

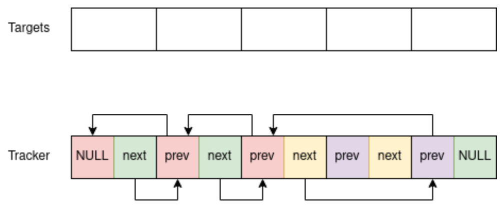


Figure 3.10: The double linked list element removal consist in pointer switching.

While this mitigation avoids the consecutive selection of an already picked point, it drastically increases the amount of memory accesses. This solution provides a memory access cost for each target which is linear with respect to their position in the list. Thus, the last target results in a worst case access time which scales as $\mathcal{O}(n)$. In order to reduce the cost, lesser *RANSAC* attempts must be performed, thus reducing the estimation quality of this algorithm.
Furthermore, this mitigation also increases the temporary memory footprint as $\mathcal{O}(n)$.

3.4.3 The egoVelocityEstimLS routine

This *Ego Velocity Estimation* routine is implemented as a voting algorithm which bases its vote con the amount of error of a given point with respect to all the others. The used estimator is the error, like the RANSAC case, but it is performed on all the set's points. This algorithm relies on the statistic relationship between the *Mean* and *Mode*: the more a distribution is concentrated, the more the two parameters tend to assume the same value. This implementation has been developed in order to mitigate the unreliability of the *RANSAC* algorithm which depends on the random-number-generator performance.

Since also this routine is a specific implementation of the *egoVelocityEstim* estimation algorithm, it inherits the requirements laid out in "Ego Velocity Estimation: Requirements" while introducing a new set which is tightly related to this particular algorithm implementation.
For what regards the actual algorithm implementation for this routine, it is laid out in the "Technical Overview specific to egoVelocityEstimLS" section below.

3.4.3.1 Requirements specific to egoVelocityEstimLS

This routine has been developed in order to vote as the *Ego Velocity* the velocity of the point which exhibits the least velocity error with respect to all the other points, thus pointing to the distribution's *Mean* velocity. The estimation is reliable if the set of points is outstandingly aggregated around the *Mode*. This tight limitation drastically reduce its field of employment.

3.4.3.2 Technical Overview specific to egoVelocityEstimLS

The algorithm is structured in order to calculate the error for each point. It does so by picking a new reference point and accumulating the squared velocity difference with respect to all the others, until all the points have been used as reference. Then it proceeds to evaluate which of the points has the least accumulated error, in which case its velocity is reported as the *Ego Velocity*.
Due to the structure of this algorithm, the computational cost and the memory accesses scale as $\mathcal{O}(n^2)$. The required temporary memory scales instead as $\mathcal{O}(1)$.
While this implementation relies on a loop kernel which is not so large compared to "*egoVelocityEstim_dop*", it still packs a more complex calculation. This algorithm actually has a more consistent timing and performance since it always use all the given targets. Nevertheless it exhibits an estimation error due to its implementation.
The fact that this algorithm votes the point's velocity which is the closest to the *Mean* velocity, means that it has an intrinsic tendency to wrong estimations. Thus

it is less robust with respect to the other implementations for what concerns the targets' statistics, *Multi-path reflections* and velocity disambiguation errors. Thus these characteristics can be resumed as follows:

PROs

+ Reasonable size loop kernel
+ More deterministic algorithm behavior
+ More consistent timing performance
+ Lesser temporary memory footprint

CONs

- Large computational and memory access cost model
- Estimation errors occur even with few targets

In figure 3.11, the graphical representation of the algorithm flow is provided.

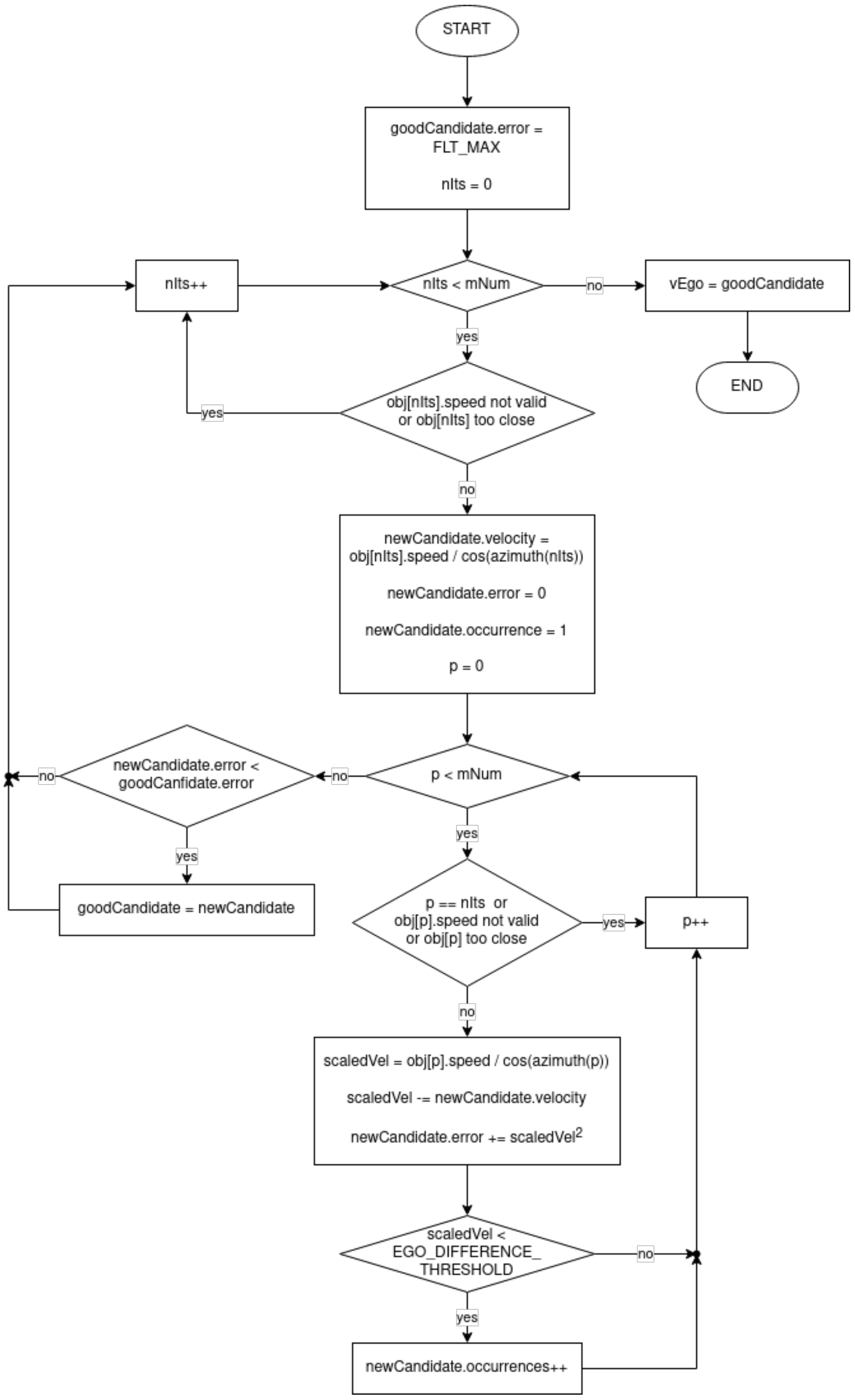

START
goodCandidate.error = FLT_MAX
nIts = 0
nIts < mNum
no
vEgo = goodCandidate
END
nIts++
yes
obj[nIts].speed not valid or obj[nIts] too close
yes
no
newCandidate.velocity = obj[nIts].speed / cos(azimuth(nIts))
newCandidate.error = 0
newCandidate.occurrence = 1
p = 0
p < mNum
no
newCandidate.error < goodCanfidate.error
no
yes
goodCandidate = newCandidate
yes
p == nIts or obj[p].speed not valid or obj[p] too close
yes
p++
no
scaledVel = obj[p].speed / cos(azimuth(p))
scaledVel -= newCandidate.velocity
newCandidate.error += scaledVel2
scaledVel < EGO_DIFFERENCE_ THRESHOLD
no
yes
newCandidate.occurrences++

Figure 3.11: Ego estimation "LS" algorithm flow.

3.4.4 The egoVelocityEstimLSOW routine

This routine has been developed after some considerations about the other ones that have been previously developed. It is introduced in order to extract a dispersion indication, given the limited set of statistical information about the set of targets. This *Ego Velocity Estimation* routine is implemented as a voting algorithm which bases its vote on the amount of error of a given point with respect to all the others and weighted by their squared occurrences in the set. This algorithm relies on the statistics of a distribution of points: the proposed estimator combines the error and occurrences properties in order to have an indication of that class of velocity's dispersion with respect to the other ones.
This algorithm relies on the *Clutter* return echo properties. As can be seen in the proposed *Range-Doppler Map* in figure 3.12, the *Clutter* return echo is usually the strongest, the broadest and the most spread along all the *Range* bins. This means that the *Clutter* echo is quite condensed around a certain *Doppler* index. This implementation has been developed in order to mitigate the bias of "*egoVelocityEstimLS*" towards the *Mean* velocity.

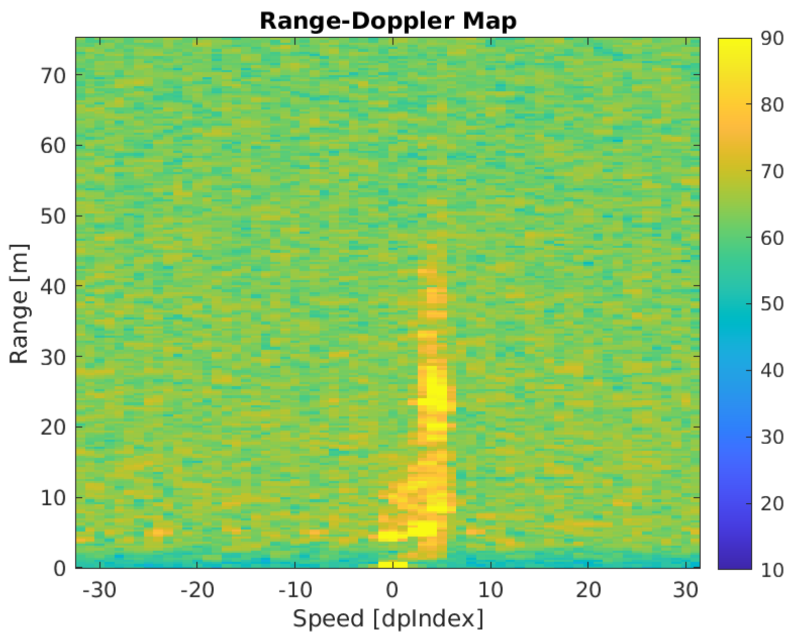


Figure 3.12: Example of a *Range-Doppler* map when the *Host* is moving.

Since also this routine is a specific implementation of the *egoVelocityEstim* estimation algorithm, it inherits the requirements laid out in "Ego Velocity Estimation: Requirements". For this implementation, no particular requirement is needed.
For what regards the actual algorithm implementation for this routine, it is laid

out in the "Technical Overview specific to egoVelocityEstimLSOW" section below.

3.4.4.1 Technical Overview specific to egoVelocityEstimLSOW

The algorithm is structured in the same way as the "*egoVelocityEstimLS*". The key difference with respect to that implementation is that the error for a given point's velocity is averaged by the amount of its occurrences. Furthermore this routine adopts the pre-computation of the velocity projection over the *Host* axis. The previous "*egoVelocityEstimLS*" algorithm instead projected the velocities in the loop kernel, thus scaling as $\mathcal{O}(n^2)$. The point which accumulated the least amount of averaged error has its velocity ultimately reported as the *Ego Velocity*.
Due to the nature of this algorithm, the computational cost and memory accesses scale as $\mathcal{O}(n^2)$. The required temporary memory scales instead as $\mathcal{O}(n)$. While this implementation relies on a compact loop kernel, it performs a quite complex calculation, but thanks to the mentioned pre-computation, the computation cost has been cut down by a scale of $\mathcal{O}(n)$, while the temporary memory footprint increased by the same factor of $\mathcal{O}(n)$.
Thus these characteristics can be resumed as follows:

PROs

+ More deterministic algorithm behavior
+ More consistent timing performance
+ Improved robustness against targets' distribution

CONs

- Large computational and memory access cost model
- Estimation errors occur even with few targets
- Largest loop kernel among all the implementations

In figure 3.13, the graphical representation of the algorithm flow is provided.

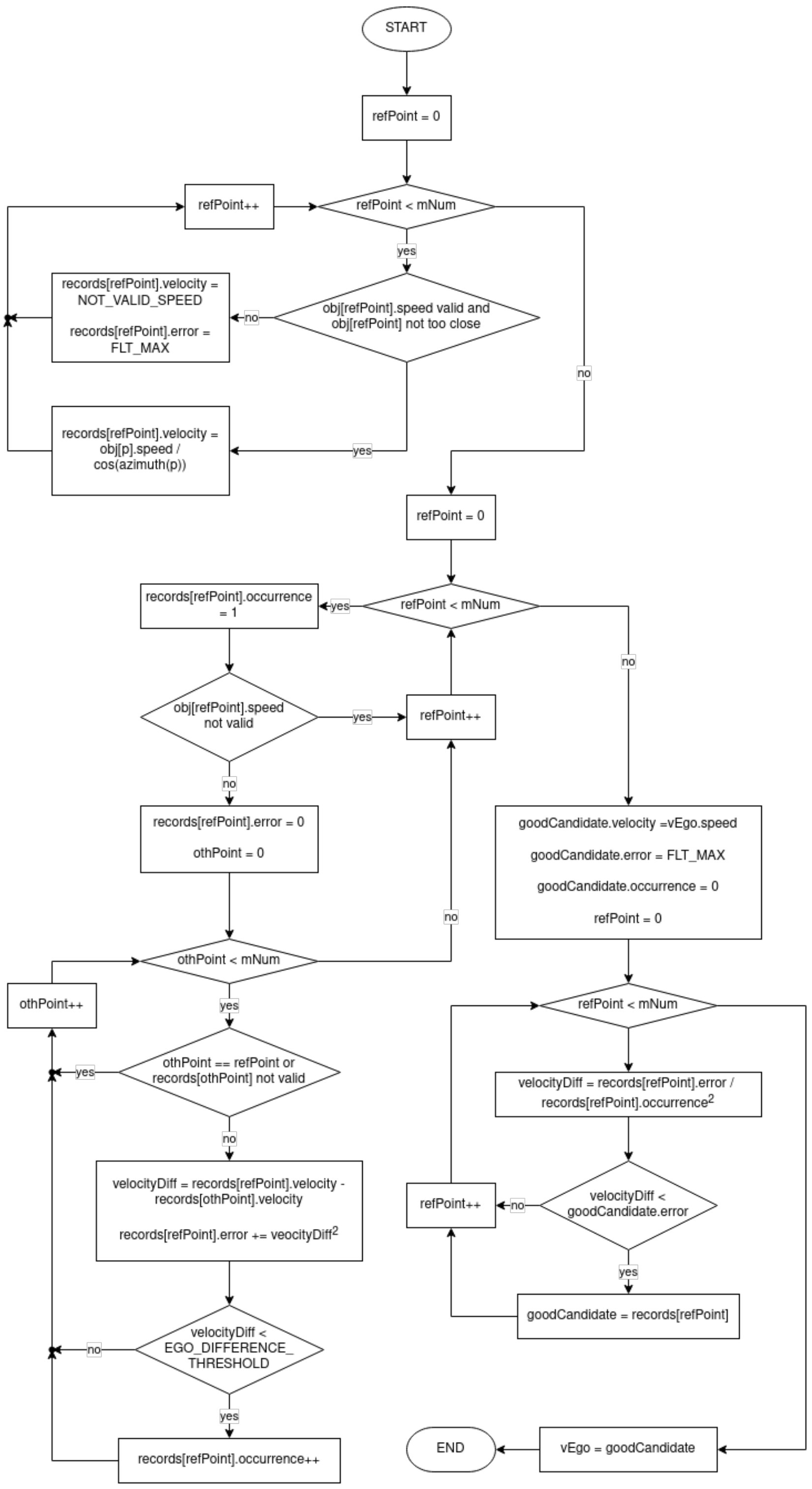

START
refPoint = 0
refPoint++
refPoint < mNum
yes
records[refPoint].velocity = NOT_VALID_SPEED
records[refPoint].error = FLT_MAX
no
obj[refPoint].speed valid and obj[refPoint] not too close
no
records[refPoint].velocity = obj[p].speed / cos(azimuth(p))
yes
refPoint = 0
records[refPoint].occurrence = 1
yes
refPoint < mNum
no
obj[refPoint].speed not valid
yes
refPoint++
no
records[refPoint].error = 0
othPoint = 0
goodCandidate.velocity =vEgo.speed
goodCandidate.error = FLT_MAX
goodCandidate.occurrence = 0
refPoint = 0
no
othPoint < mNum
othPoint++
yes
refPoint < mNum
othPoint == refPoint or records[othPoint] not valid
yes
velocityDiff = records[refPoint].error / records[refPoint].occurrence^2
no
velocityDiff = records[refPoint].velocity - records[othPoint].velocity
records[refPoint].error += veocityDiff^2
refPoint++
no
velocityDiff < goodCandidate.error
yes
goodCandidate = records[refPoint]
velocityDiff < EGO_DIFFERENCE_THRESHOLD
no
yes
END
vEgo = goodCandidate
records[refPoint].occurrence++

Figure 3.13: Ego estimation "LSOW" algorithm flow.

Chapter 4

Software Testing and Validation

This chapter presents the sets of tests which have been performed on the various estimations' implementation in order to evaluate their respective result quality and performance. The test were used as a feedback mechanism during the development for each routine, mainly for debug purposes. When a routine version reaches a point where can be considered reasonably stable, with respect to software bugs, a test campaign is performed in order to validate it. The *Validation* process consist in documenting whether the *Software* meets the specification and requirements which were used to design it. The fact that the *Software* behavior has correspondence with its documented intent, means that it fulfills its intended purpose and thus it is a valid implementation.

4.1 Development & Validation Testing Platform

The *Validation* process consist of a set of tests performed on a testing platform in a context which must resemble as much as possible the one for which a product is intended to work during its regular usage. The *egoVelocityEstim* routine has been designed in order to be interfaced to various possible scenarios, but in order to be validated it must be tested on different testing platforms and their environments. Each one of those must emulate their specific use case for which this *Software* routine has been integrated. Considering that the development platform is given, namely the one presented in the "Software Overview" section, it is also adopted as the *Validation Testing Platform*.
Regarding the *RADAR* development platform's hardware is presented in figure 4.1. The *RADAR* hardware was designed in order to be integrated inside a *car lamp*, as can be seen on figure 4.2.

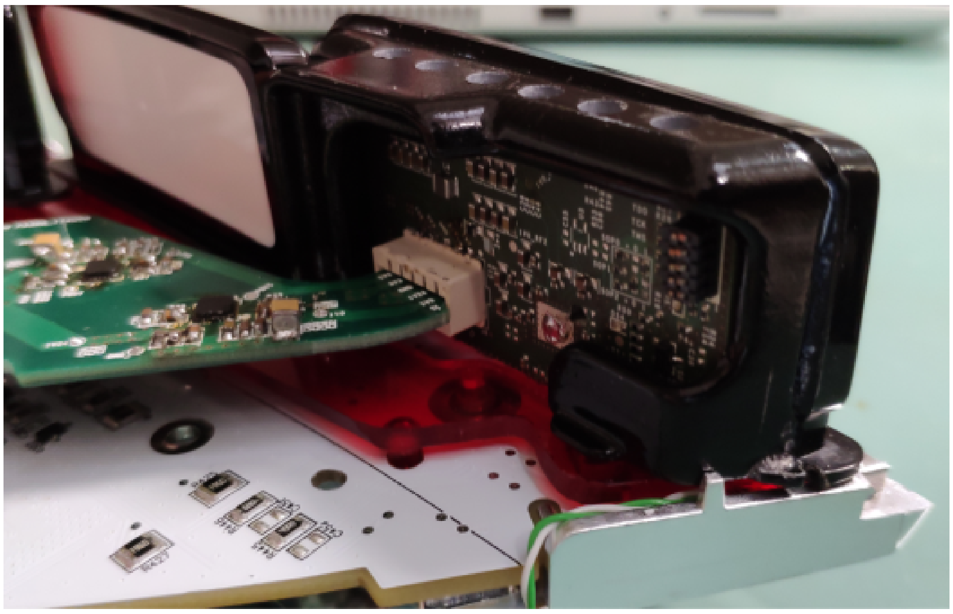


Figure 4.1: The *RADAR* platform's hardware (inside view).

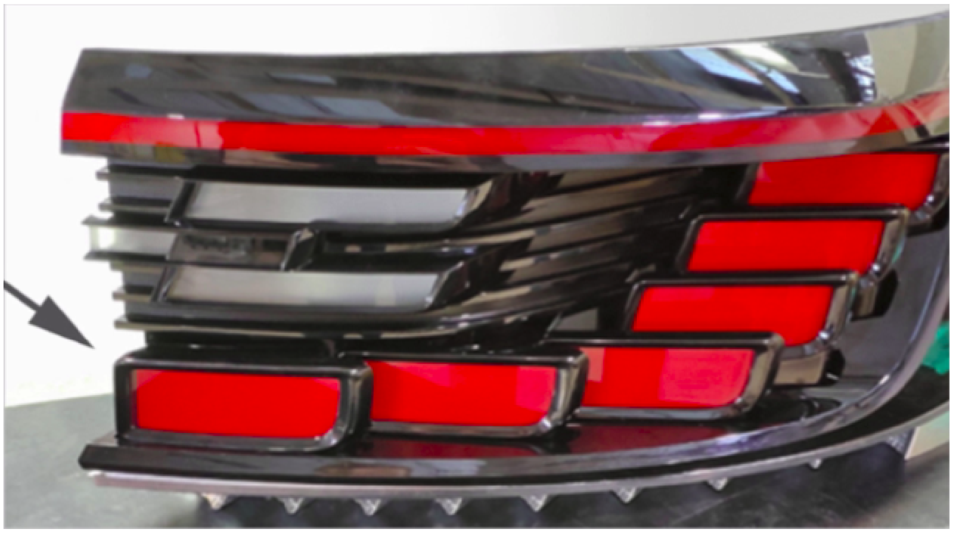


Figure 4.2: The *RADAR* platform's hardware (outside view).

For what concern the actual test, since the adopted *RADAR* is tailored to automotive applications, it has been tested in the *Real-World* by mounting it in a car which has been driven in a suitable environment scenery while the *RADAR*

acquired the data.
The aim of the setup, shown in figure 4.3, was to emulate a realistic case which involves cars and a urban scenery, whose arrangement is shown in figure 4.4.

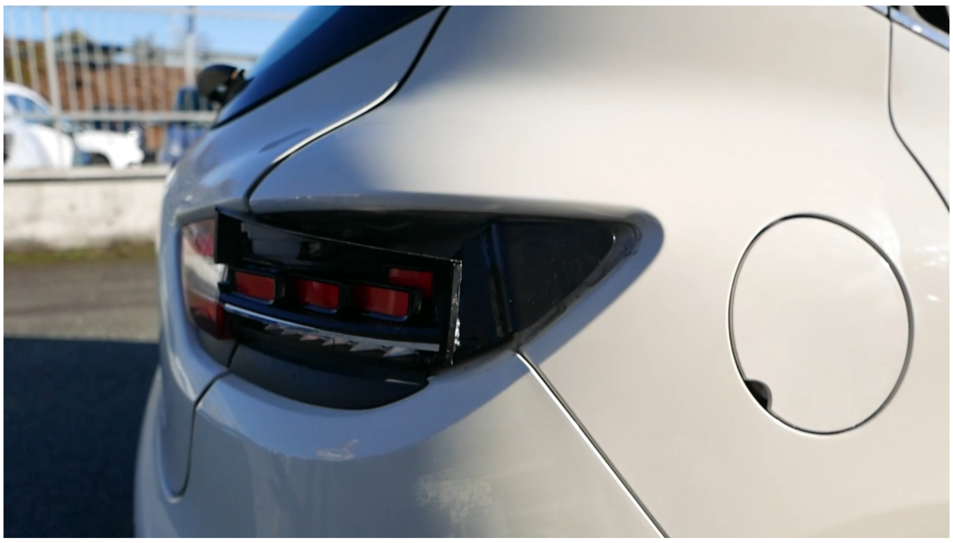


Figure 4.3: *RADAR* mounting layout on the host car.

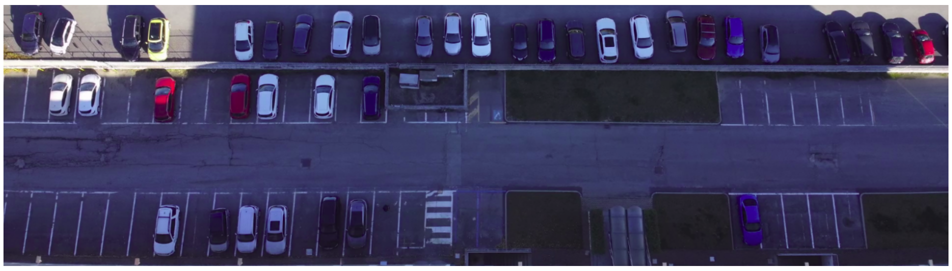


Figure 4.4: Aerial view of the environment scenery.

The *Radar* is mounted on the rear-left side of the car. It looks behind with an inclination of 35° with respect to the car's movement axis. A visual representation of the mounting configuration can be seen in figures 4.5.
The chosen scenery is a parking lot where rows of parked vehicles can be found on each side and a building can be found on one of the two sides of the road. The chosen target is a car which has been driven around in opening, closing and stationary movements.

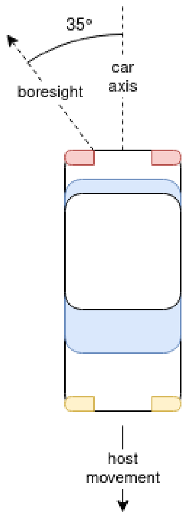


Figure 4.5: *RADAR* orientation on the host car.

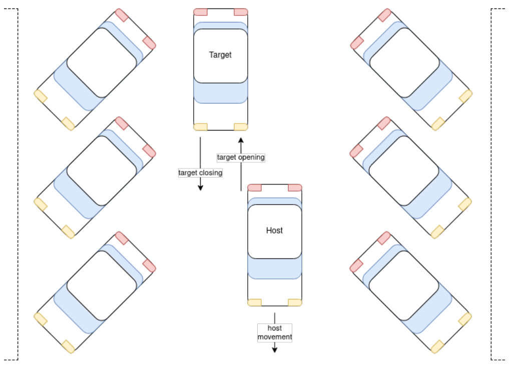


Figure 4.6: Context representation with Environment, Host and Target interaction.

4.2 Test Campaign

The test campaign is a crucial aspect during a product validation process. It must tailor the requirements for which the product was designed and has to deal with the fact that some intrinsic limitation are present due to the nature itself of the environmental factors and technological capabilities of the product itself.
The employed *Radar platform* has an intrinsic limit on the amount of data that can be transferred during its regular operation. This is caused by the fact that the data transfer is performed on an *SPI* interface which sends data to a computer. This constraints the amount of data that can be transferred to the data-transfer rate times the frame acquisition time. If not respected, a transfer of large amounts of data causes a synchronization signalling miss in timing terms, thus triggering a system-level *Real-Time Error* which blocks the *Radar*.

In order to have a comparable metric between all the algorithm's implementations, all the algorithms should run using the same set of data as input. This means to run all the algorithms on the *Radar* on the same frame's data. But unfortunately this solution cannot be adopted due to the same timing issues because the *Radar* fails the same synchronization mechanism as above.

Taking into account these timing issues, the test campaign has been performed for a single *egoVelocityEstim* routine, then the data accessed by the algorithm is saved on a computer. The saved data is later loaded into the *Radar* during each algorithm test. This testing schema emulates a *Real-Time* test of all the *egoVelocityEstim* running sequentially on the same set of data.

Figure 4.7: The acquired data is collected on a *PC*.

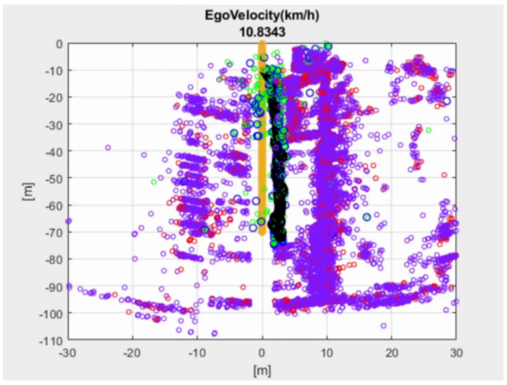


Figure 4.8: Example of acquired data. In this frame *all* information is plotted.

The set of tests are performed considering possible scenarios with respect to the single target in the scenery, as presented in "Development & Validation Testing Platform". These tests are obtained as a permutation of situations with the *target* which is either missing, approaching or leaving and the *host*, which is either still or moving. This permutations result in the following testing context:

- static host background acquisition
- static host closing target acquisition
- static host opening target acquisition
- accelerating host background acquisition
- dynamic host background acquisition
- dynamic host closing target acquisition
- dynamic host opening target acquisition

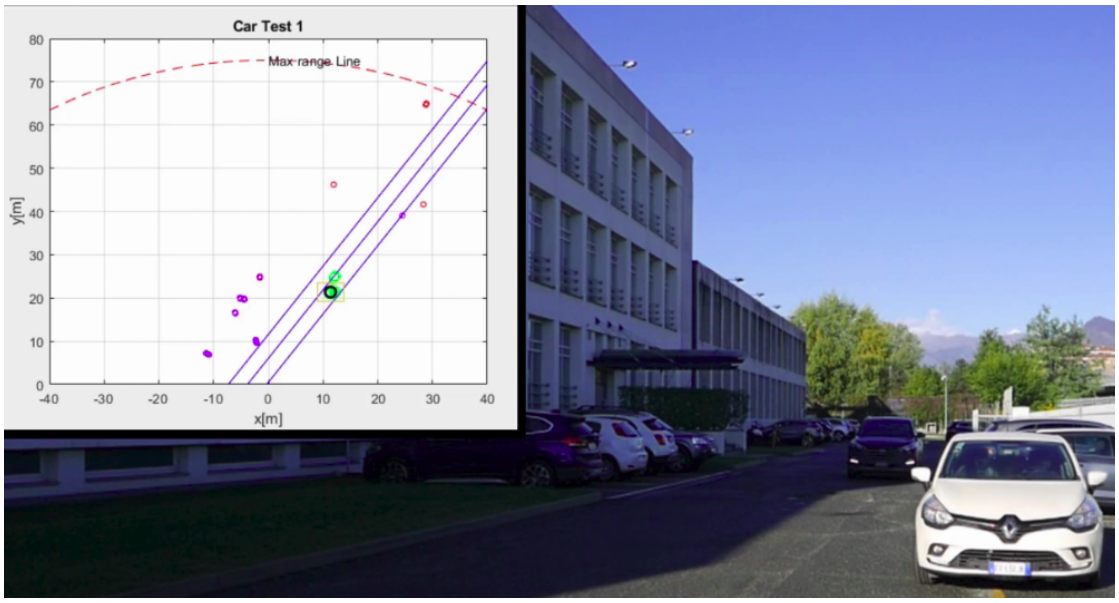


Figure 4.9: Static host and closing target acquisition.

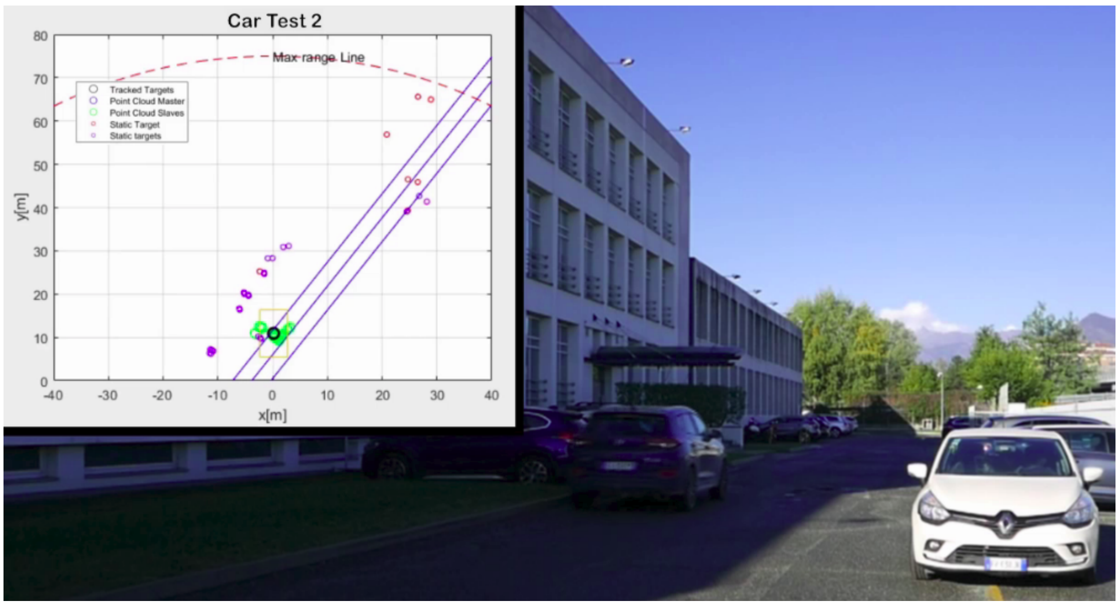


Figure 4.10: Static host and opening target acquisition.

All the *Host* and *Target* motions have been performed in a straight line. Furthermore, the acceleration test with a target is avoided due to difficulties in synchronization between the drivers of the two cars.

4.3 Static host, background acquisition

For this acquisition the *Host* car was parked on the *Host*'s left side of the road, with the *Radar* looking behind. For the sake of readability, a snapshot of just three consecutive frames are presented. As can be seen in both figure 4.11 and table 4.1, all the implementations conform to estimating the *Host* velocity as $0\,\mathrm{km\,h^{-1}}$ (the same applies for the pair of figure 4.13 with table 4.2 and figure 4.15 with table 4.3).

By observing this snapshot, all the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like no estimation algorithm was applied to the original *Radar Software Stack*. This is the intended behavior when no *Target* is present and the *Host* is stationary.

4.3.1 Frame 1

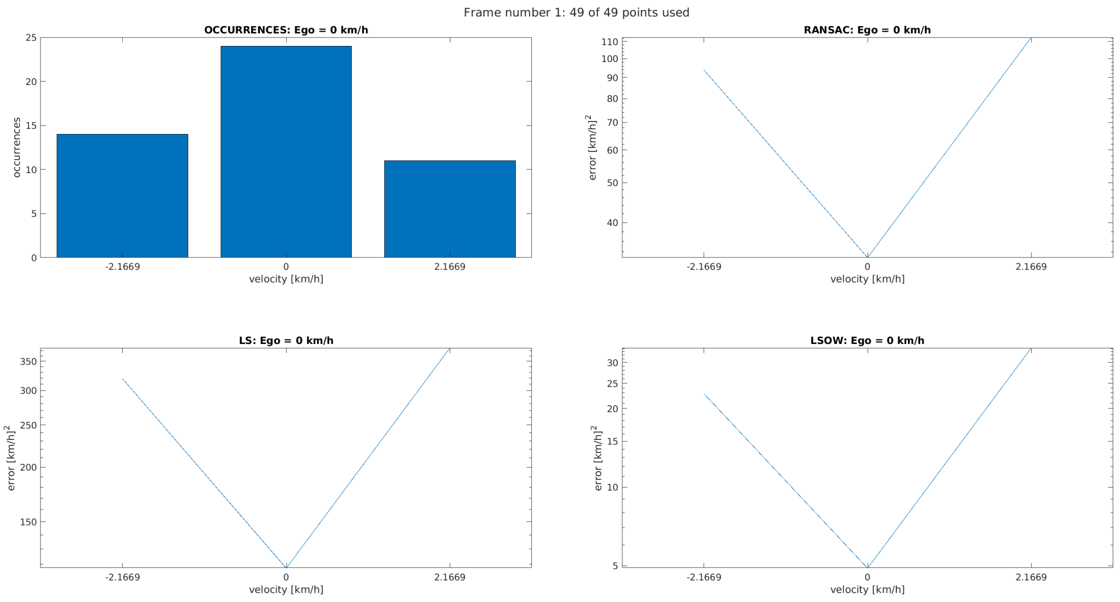


Figure 4.11: Algorithms graphical voting outcome during frame 1

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	14	1.9727	2.5042	1.358
0	24	1.5168	2.0696	0.68939
2.1669	11	2.0519	2.5748	1.5334

Table 4.1: Algorithms numerical voting outcome during frame 1

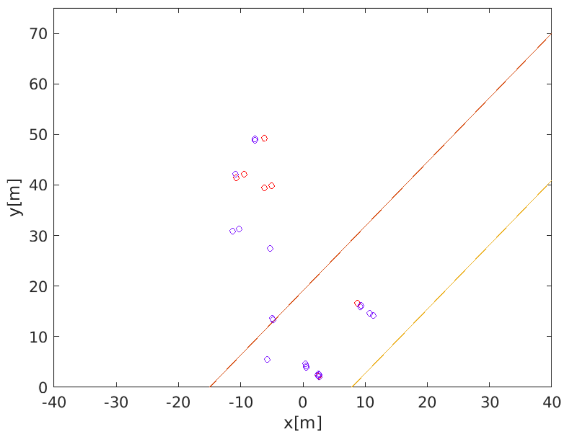


Figure 4.12: Input target cloud to the algorithms during frame 1

4.3.2 Frame 2

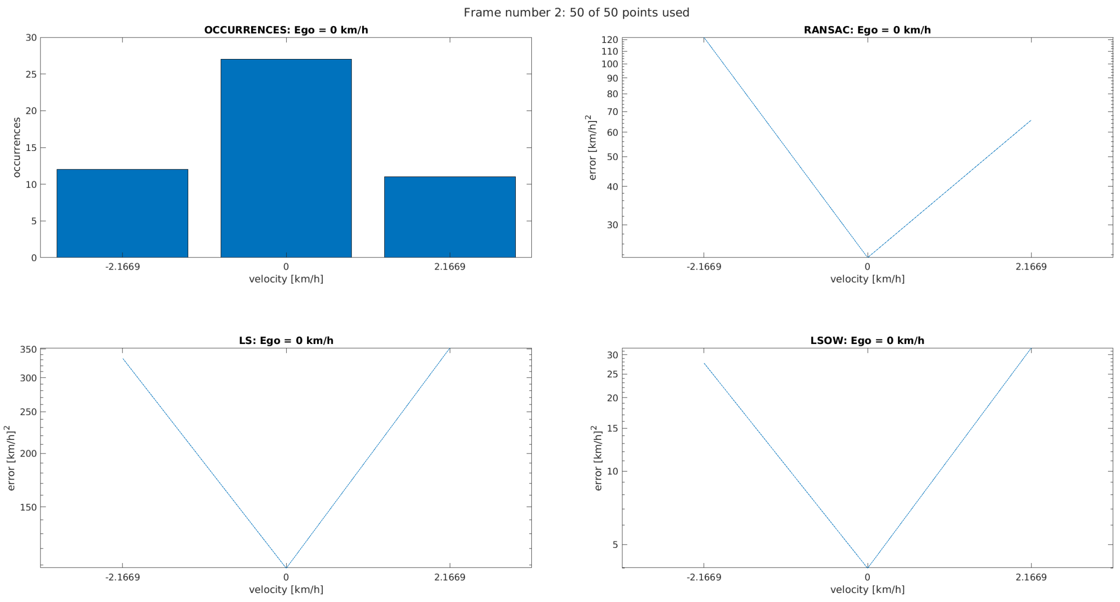


Figure 4.13: Algorithms graphical voting outcome during frame 2

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	12	2.0866	2.5229	1.4437
0	27	1.3706	2.0334	0.60202
2.1669	11	1.8178	2.5467	1.5053

Table 4.2: Algorithms numerical voting outcome during frame 2

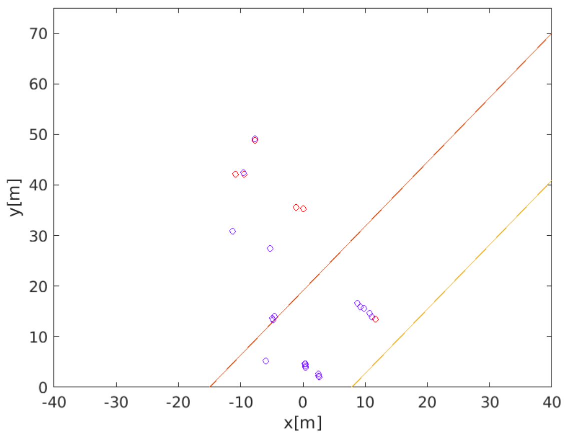


Figure 4.14: Input target cloud to the algorithms during frame 2

4.3.3 Frame 3

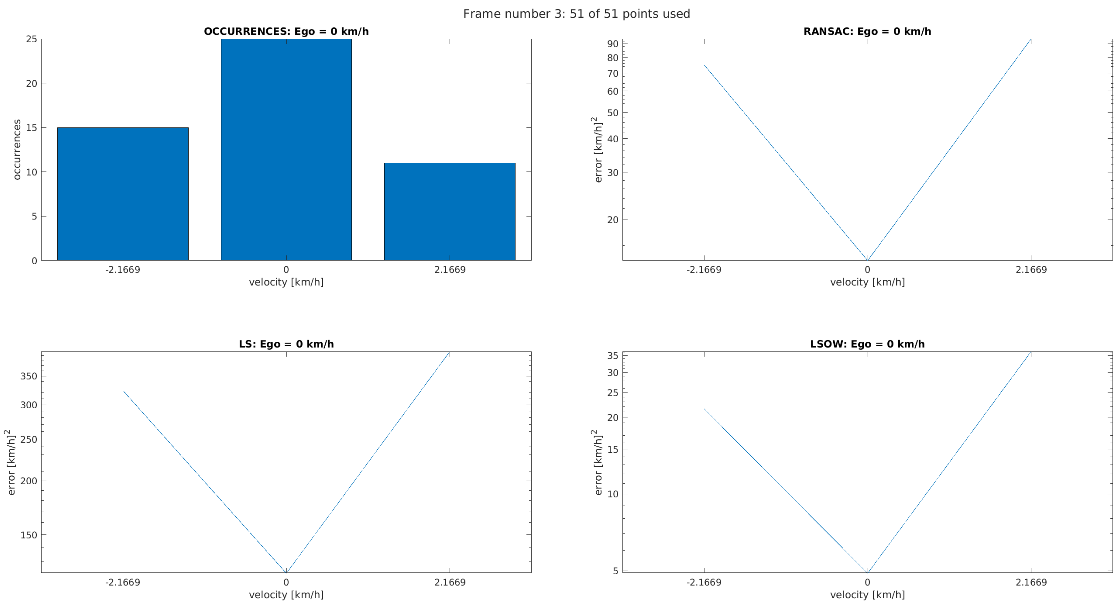


Figure 4.15: Algorithms graphical voting outcome during frame 3

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	15	1.8758	2.5105	1.3344
0	25	1.1488	2.0866	0.68869
2.1669	11	1.9727	2.6011	1.5597

Table 4.3: Algorithms numerical voting outcome during frame 3

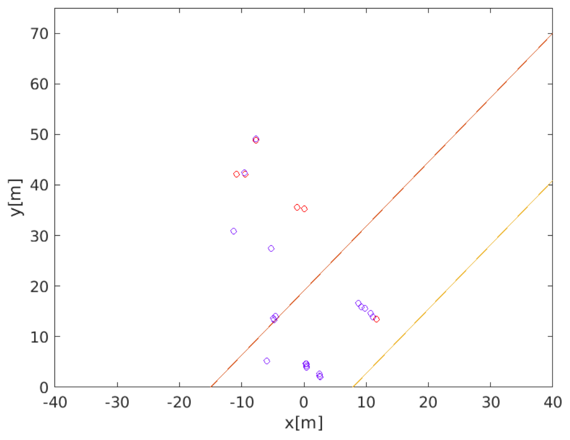


Figure 4.16: Input target cloud to the algorithms during frame 3

4.4 Static host, closing target

For this acquisition the *Host* car was still parked on the *Host*'s left side of the road, with the *Radar* looking behind. Since now there is an incoming *Target* from afar, three snapshots are taken for three situations:

- when the *Radar* started a track on the *Target* which started moving in a contour region of the field of view
- when the *Target* is at a medium distance
- when the *Radar* built the last tracks before releasing them because the *Target* is out of sight after the overtake of the *Host*

The *Target* car used in this test has been driven towards the *Host* car at about $30\,\mathrm{km\,h^{-1}}$.
The reason for the fact that this snapshot shows a frame numbering that doesn't start from one is that it has been extracted from the same acquisition used for "Static host, background acquisition". During that acquisition the *Target* car was later let in the field of view of the *Radar*.

4.4.1 Static host, closing target: Track Start

The allocation of a track for a closing target can be seen in the frames proposed in the figures 4.18, 4.20 and 4.22. The estimation of the "*LS*" algorithm is the one which is affected by an error during frame 54, as can be seen in figure 4.19 and in table 4.5. This same phenomenon happens also on frame 55.

By observing this snapshot, the *Tracking* unit correctly starts a track on a moving *Target* when no *Ego Velocity* estimation error occurs. The tracker behavior is like intended when a *Target* appears and the *Host* is stationary.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity.

4.4.1.1 Frame 53

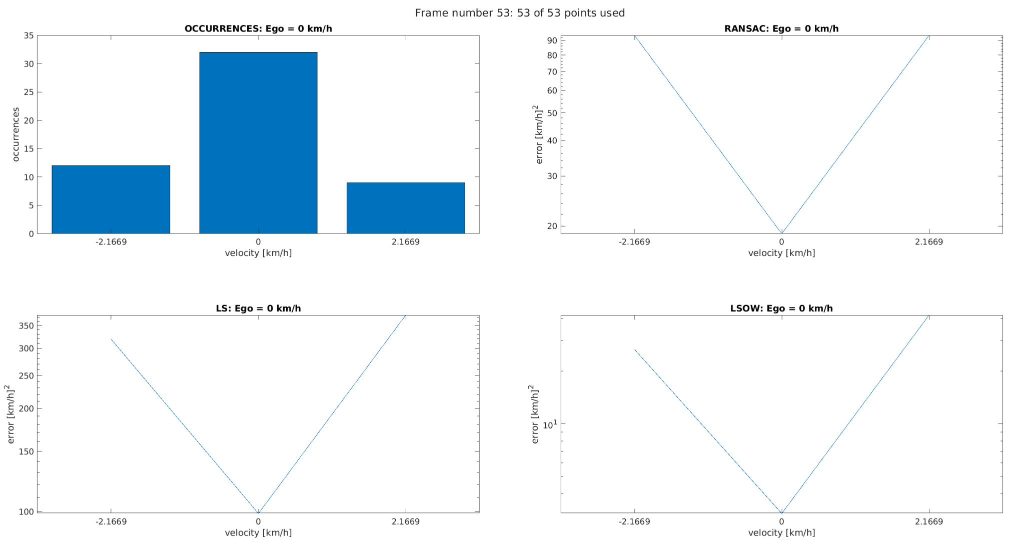


Figure 4.17: Algorithms graphical voting outcome during frame 53

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	12	1.9727	2.5042	1.425
0	32	1.2737	1.9939	0.48873
2.1669	9	1.9727	2.5748	1.6205

Table 4.4: Algorithms numerical voting outcome during frame 53

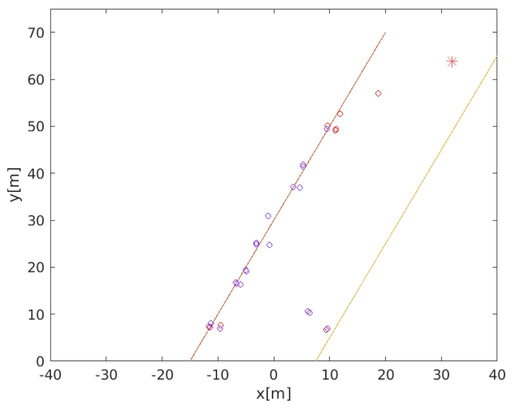


Figure 4.18: Input target cloud to the algorithms during frame 53

4.4.1.2 Frame 54

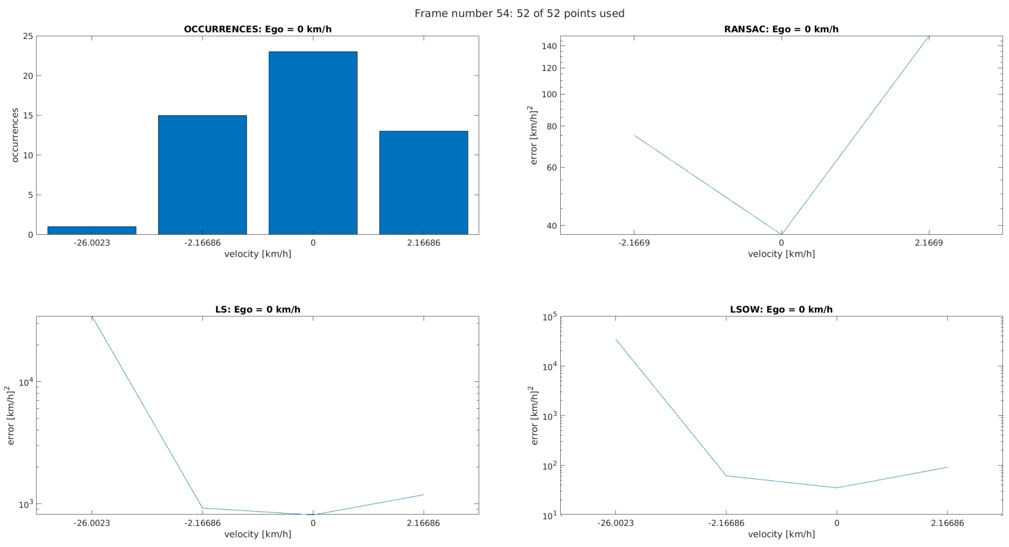


Figure 4.19: Algorithms graphical voting outcome during frame 54

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	1	-	4.5364	4.5364
-2.16686	15	1.8758	2.9639	1.7878
0	23	1.5748	2.9072	1.5455
2.16686	13	2.1768	3.0731	1.9591

Table 4.5: Algorithms numerical voting outcome during frame 54

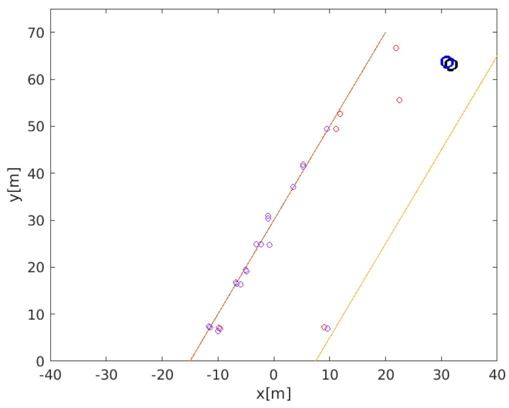


Figure 4.20: Input target cloud to the algorithms during frame 54

4.4.1.3 Frame 55

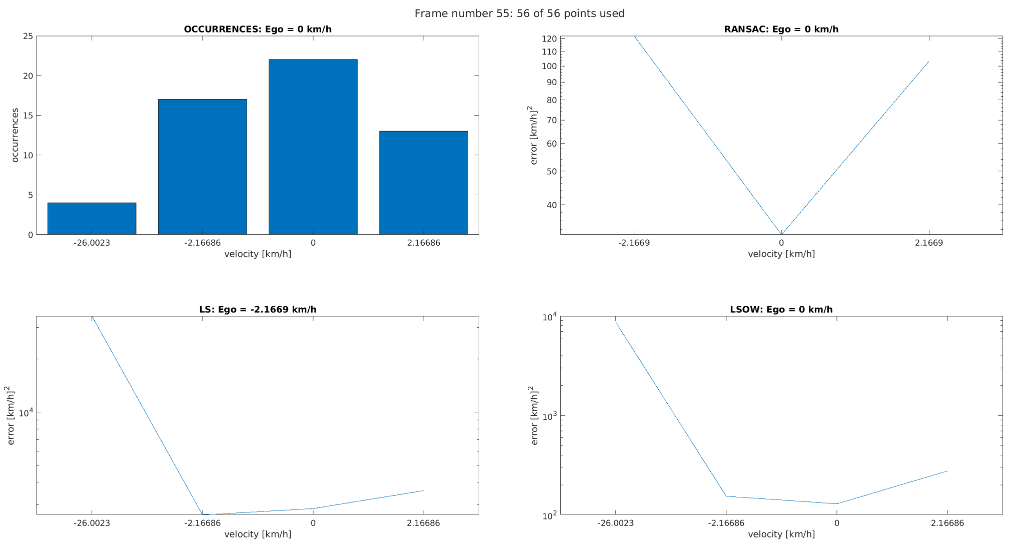


Figure 4.21: Algorithms graphical voting outcome during frame 55

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	4	-	4.5422	3.9401
-2.16686	17	2.0866	3.4183	2.1878
0	22	1.5168	3.4541	2.1117
2.16686	13	2.0141	3.5559	2.4419

Table 4.6: Algorithms numerical voting outcome during frame 55

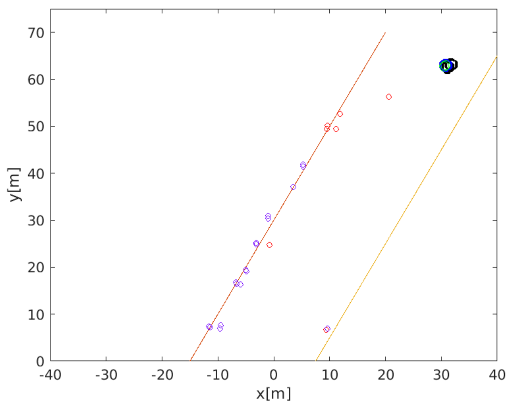


Figure 4.22: Input target cloud to the algorithms during frame 55

4.4.2 Static host, closing target: Track until midway

The snapshot frames proposed in the figures 4.24, 4.26 and 4.28, present the track for the target which started in the "Static host, closing target: Track Start" section. In this case it is presented when the targets has reached the midway between the position from which the track started and to which the track is released. The estimation error this time affects the whole snapshot, but still only for the "*LS*" implementation.
By observing this snapshot, the *Tracking* unit correctly keeps track on a moving *Target* when no *Ego Velocity* estimation error occurs. The tracker behavior is like intended when a tracked *Target* is already present and the *Host* is stationary.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity, where the impact of the *Target* is not negligible because its large velocity made the *Mean* velocity shift away from the *Mode*.

4.4.2.1 Frame 107

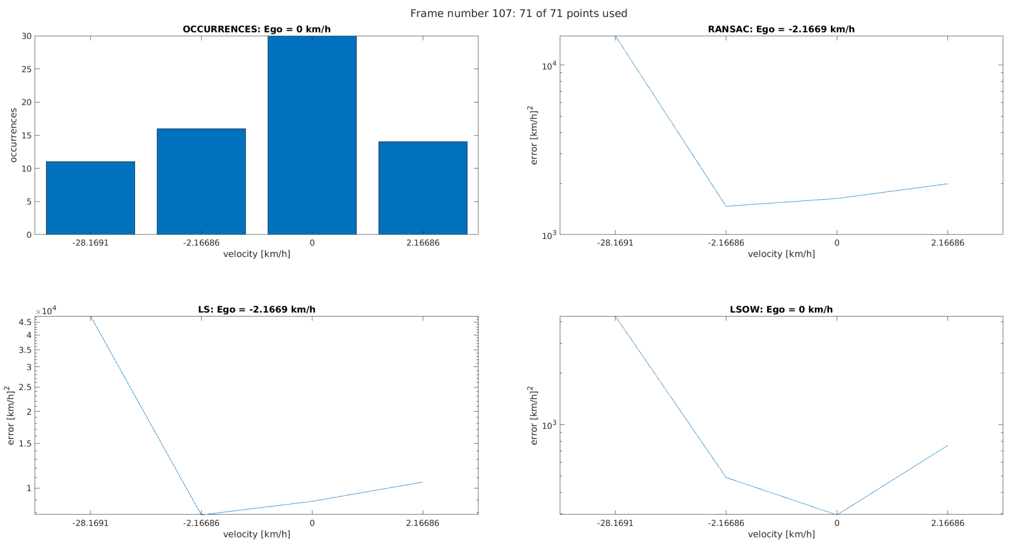


Figure 4.23: Algorithms graphical voting outcome during frame 107

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-28.1691	11	4.1726	4.6768	3.6354
-2.16686	16	3.1672	3.8944	2.6903
0	30	3.2132	3.9479	2.4708
2.16686	14	3.3	4.0238	2.8777

Table 4.7: Algorithms numerical voting outcome during frame 107

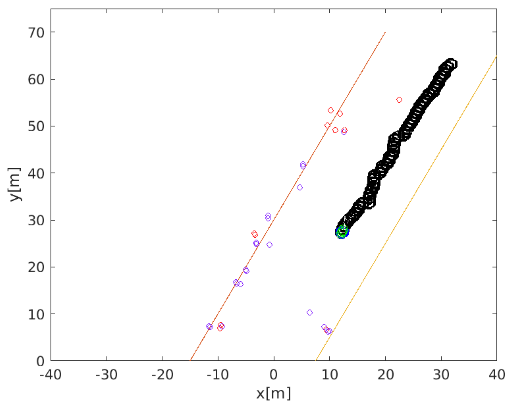


Figure 4.24: Input target cloud to the algorithms during frame 107

4.4.2.2 Frame 108

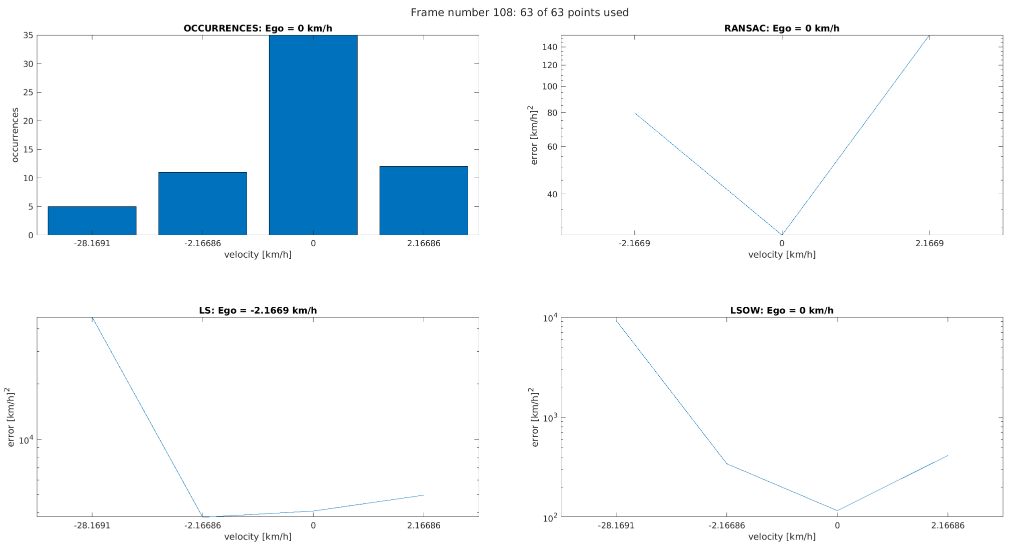


Figure 4.25: Algorithms graphical voting outcome during frame 108

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-28.1691	5	-	4.6651	3.9662
-2.16686	11	1.9021	3.5764	2.535
0	35	1.4498	3.6102	2.0661
2.16686	12	2.1902	3.6966	2.6174

Table 4.8: Algorithms numerical voting outcome during frame 108

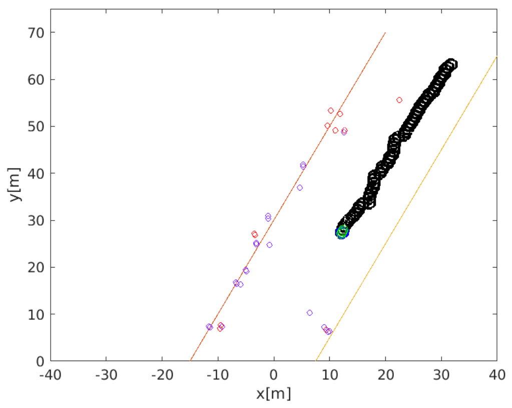


Figure 4.26: Input target cloud to the algorithms during frame 108

4.4.2.3 Frame 109

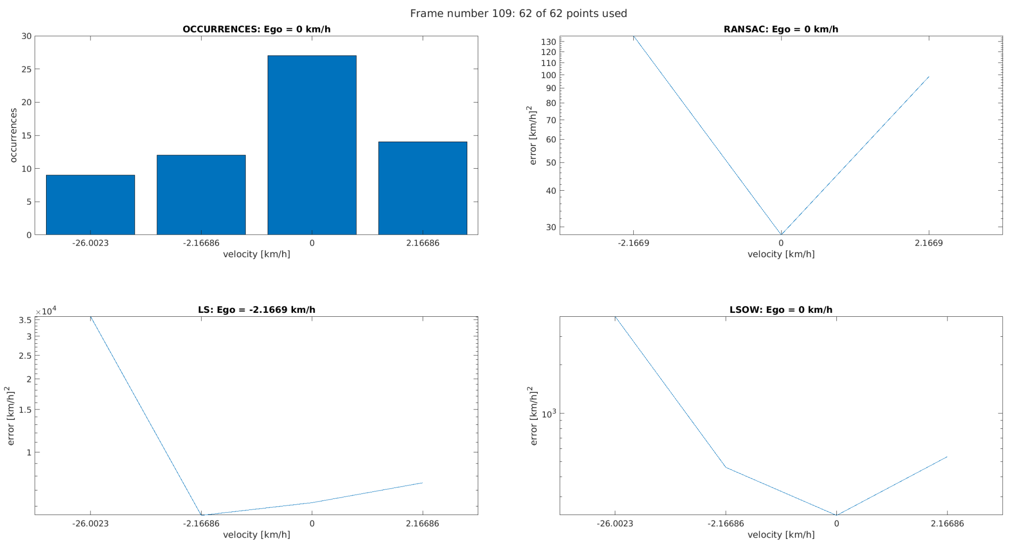


Figure 4.27: Algorithms graphical voting outcome during frame 109

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	9	-	4.5585	3.6042
-2.16686	12	2.1341	3.7406	2.6614
0	27	1.4498	3.7929	2.3615
2.16686	14	1.9939	3.8747	2.7286

Table 4.9: Algorithms numerical voting outcome during frame 109

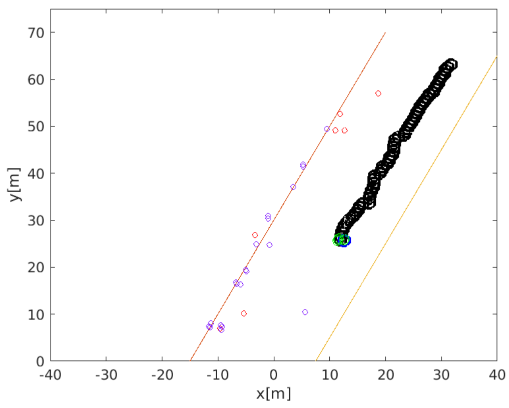


Figure 4.28: Input target cloud to the algorithms during frame 109

4.4.3 Static host, closing target: Track End

The release of the target can be seen in the frames proposed in the figures 4.30, 4.32 and 4.34. The release occurs because the target has escaped the field of view of the *Radar*, thus the *Track* resources can be released for later assignment to new tracks. In these frames the estimation error affects both the "*LS*" and "*RANSAC*" implementations.

By observing this snapshot, the *Tracking* unit correctly releases the track on a *Target* when it is no longer in the field of view. The *Ego Velocity* estimation routine has no effect on a *Track* which is released for the reason mentioned above. The tracker behavior is like intended when a previously tracked *Target* escapes the field of view and the *Host* is stationary.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity, where the impact of the *Target* is not negligible because its large velocity made the *Mean* velocity shift away from the *Mode*.
The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

4.4.3.1 Frame 140

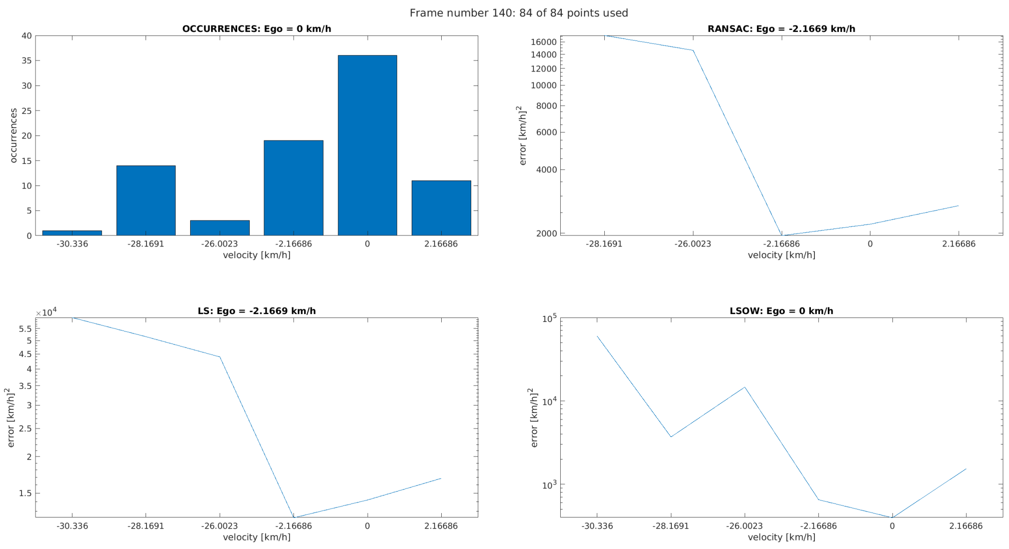


Figure 4.29: Algorithms graphical voting outcome during frame 140

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-30.336	1	-	4.7778	4.7778
-28.1691	14	4.2345	4.7123	3.5661
-26.0023	3	4.1644	4.6429	4.1658
-2.16686	19	3.2897	4.0913	2.8125
0	36	3.3438	4.1522	2.5959
2.16686	11	3.4313	4.2265	3.1851

Table 4.10: Algorithms numerical voting outcome during frame 140

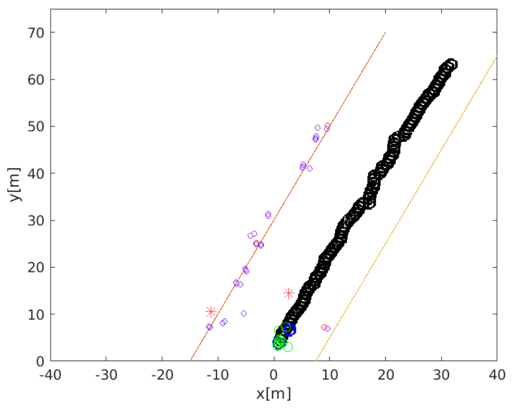


Figure 4.30: Input target cloud to the algorithms during frame 140

4.4.3.2 Frame 141

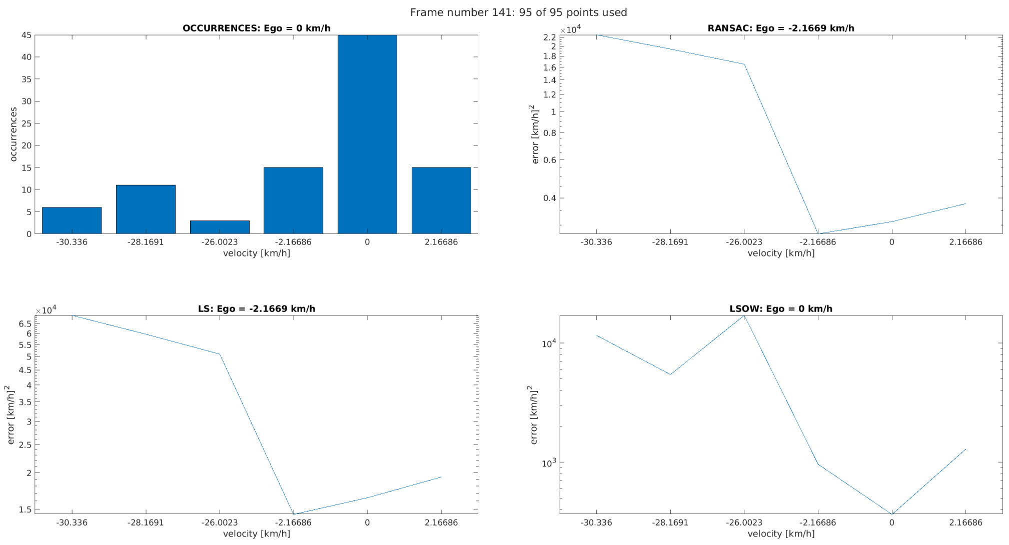


Figure 4.31: Algorithms graphical voting outcome during frame 141

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-30.336	6	4.3535	4.8405	4.0624
-28.1691	11	4.288	4.7759	3.7345
-26.0023	3	4.2181	4.7077	4.2306
-2.16686	15	3.4366	4.1582	2.9821
0	45	3.4932	4.2154	2.5621
2.16686	15	3.5758	4.2863	3.1103

Table 4.11: Algorithms numerical voting outcome during frame 141

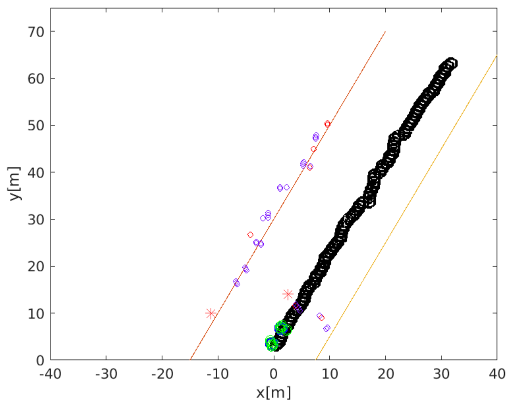


Figure 4.32: Input target cloud to the algorithms during frame 141

4.4.3.3 Frame 142

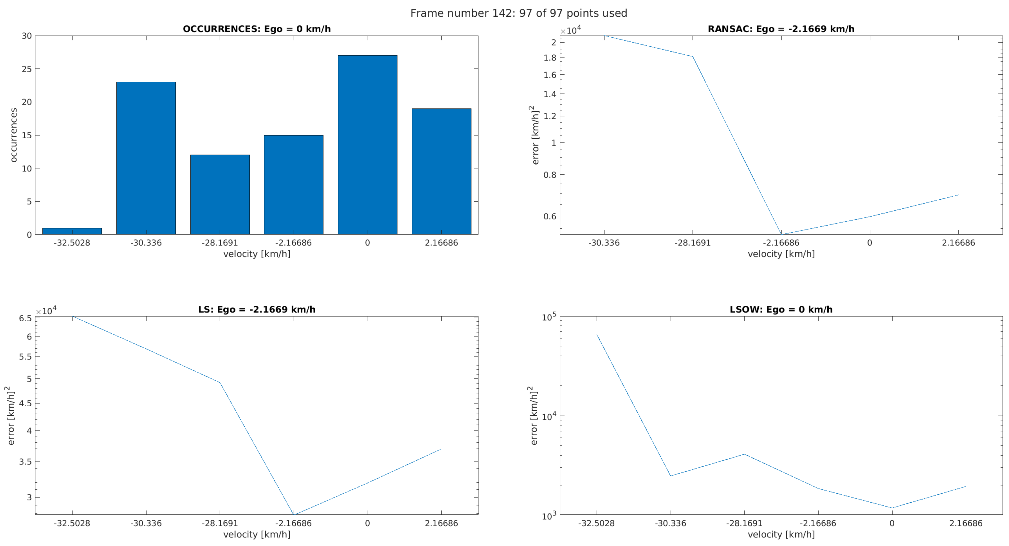


Figure 4.33: Algorithms graphical voting outcome during frame 142

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-32.5028	1	-	4.8162	4.8162
-30.336	23	4.3215	4.755	3.3933
-28.1691	12	4.2582	4.6918	3.6126
-2.16686	15	3.7224	4.4435	3.2675
0	27	3.7765	4.5038	3.0725
2.16686	19	3.8419	4.5676	3.2889

Table 4.12: Algorithms numerical voting outcome during frame 142

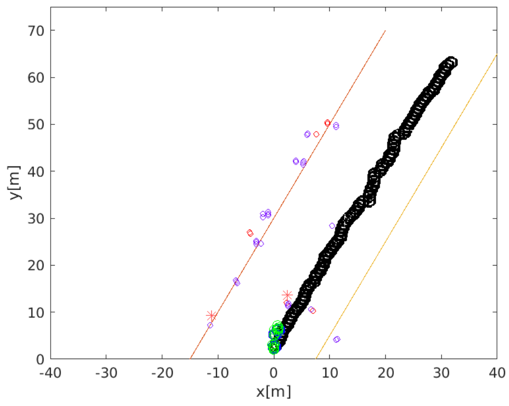


Figure 4.34: Input target cloud to the algorithms during frame 142

4.5 Static host, opening target

For this acquisition the *Host* car was still parked on the *Host*'s left side of the road, with the *Radar* looking behind. Since now there is a *Target* entering the field of view from the *Host*'s right side of the road, three snapshots are taken for three situations:

- when the *Radar* started a track on the *Target* because it entered the field of view
- when the *Target* is at a medium distance
- when the *Radar* built the last tracks before releasing them because the *Target* crossed the *Maximum Range* and thus it is out of sight

The *Target* car used in this test has been driven away from the *Host* car at about $30\,\mathrm{km\,h^{-1}}$.

4.5.1 Static host, opening target: Track Start

The allocation of a track for an opening target can be seen in the frames proposed in the figures 4.36, 4.38 and 4.40. Both the estimation of the "*LS*" and "*RANSAC*" algorithms are affected by an error during all the frames presented for this snapshot.

By observing this snapshot, the *Tracking* unit correctly starts a track on a moving *Target* when no *Ego Velocity* estimation error occurs. The tracker behavior is like intended when a *Target* enters the field of view and the *Host* is stationary.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity, where the impact of the *Target* is not negligible because its large velocity made the *Mean* velocity shift away from the *Mode*.
The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

4.5.1.1 Frame 57

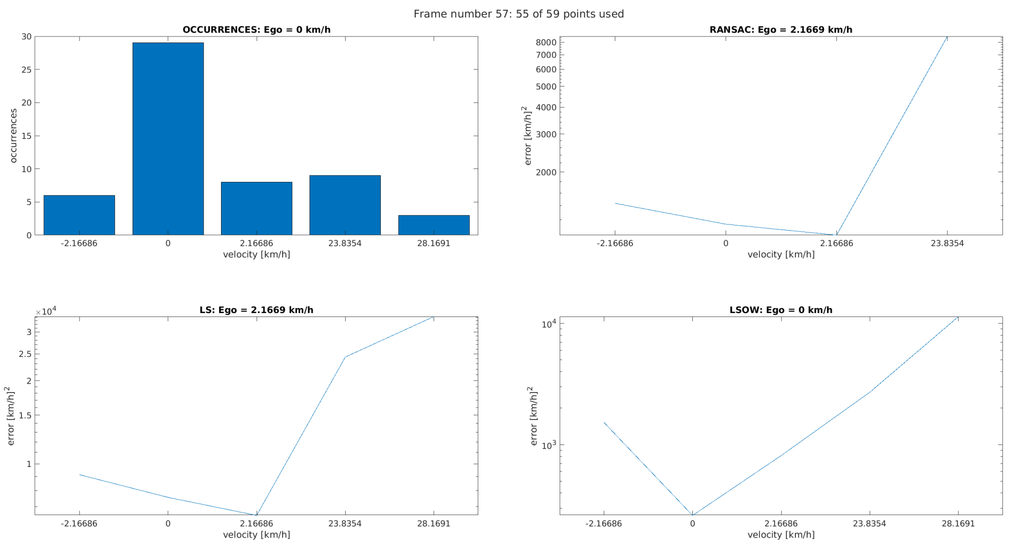


Figure 4.35: Algorithms graphical voting outcome during frame 57

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	6	3.156	3.9606	3.1824
0	29	3.0591	3.8785	2.4161
2.16686	8	3.0081	3.8131	2.91
23.8354	9	3.931	4.3864	3.4322
28.1691	3	-	4.5329	4.0558

Table 4.13: Algorithms numerical voting outcome during frame 57

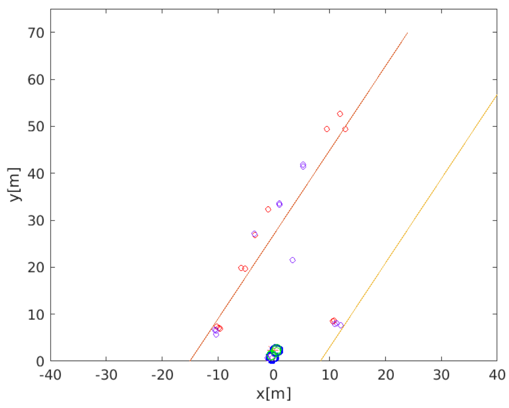


Figure 4.36: Input target cloud to the algorithms during frame 57

4.5.1.2 Frame 58

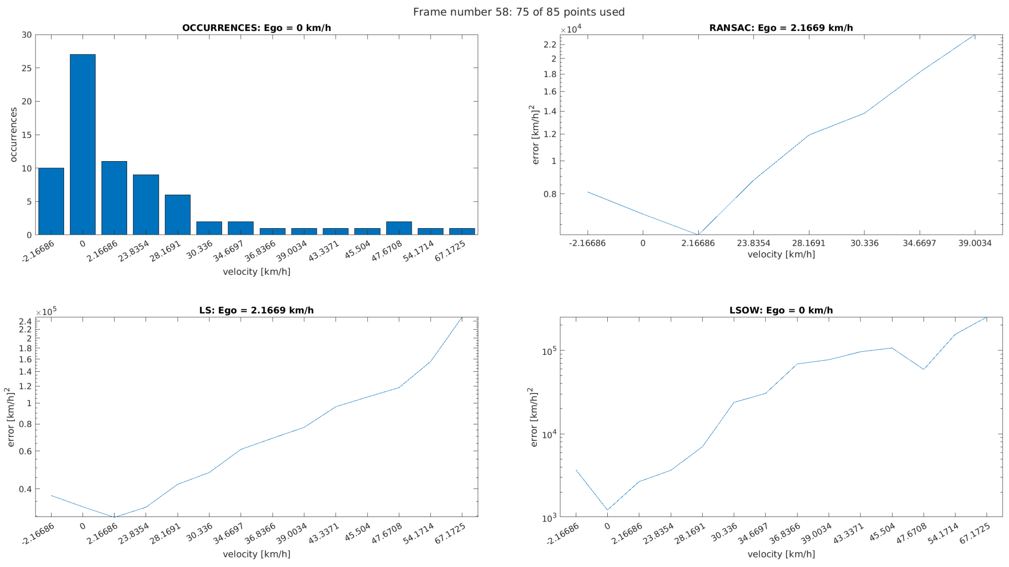


Figure 4.37: Algorithms graphical voting outcome during frame 58

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	10	3.9087	4.5714	3.5714
0	27	3.8434	4.519	3.0876
2.16686	11	3.7822	4.4698	3.4284
23.8354	9	3.9435	4.5169	3.5627
28.1691	6	4.0762	4.6233	3.8452
30.336	2	4.1402	4.6779	4.3768
34.6697	2	4.2611	4.7852	4.4842
36.8366	1	-	4.8371	4.8371
39.0034	1	4.3718	4.8874	4.8874
43.3371	1	-	4.9829	4.9829
45.504	1	-	5.0281	5.0281
47.6708	2	-	5.0717	4.7706
54.1714	1	-	5.1929	5.1929
67.1725	1	-	5.3996	5.3996

Table 4.14: Algorithms numerical voting outcome during frame 58

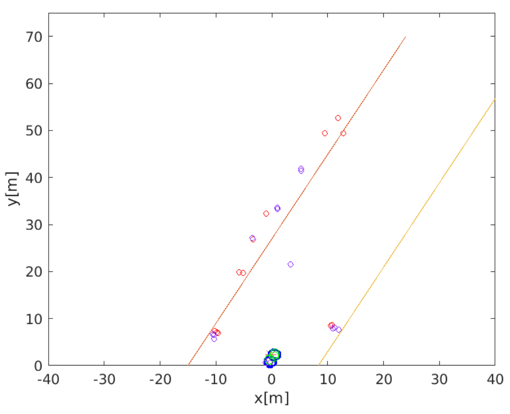


Figure 4.38: Input target cloud to the algorithms during frame 58

4.5.1.3 Frame 59

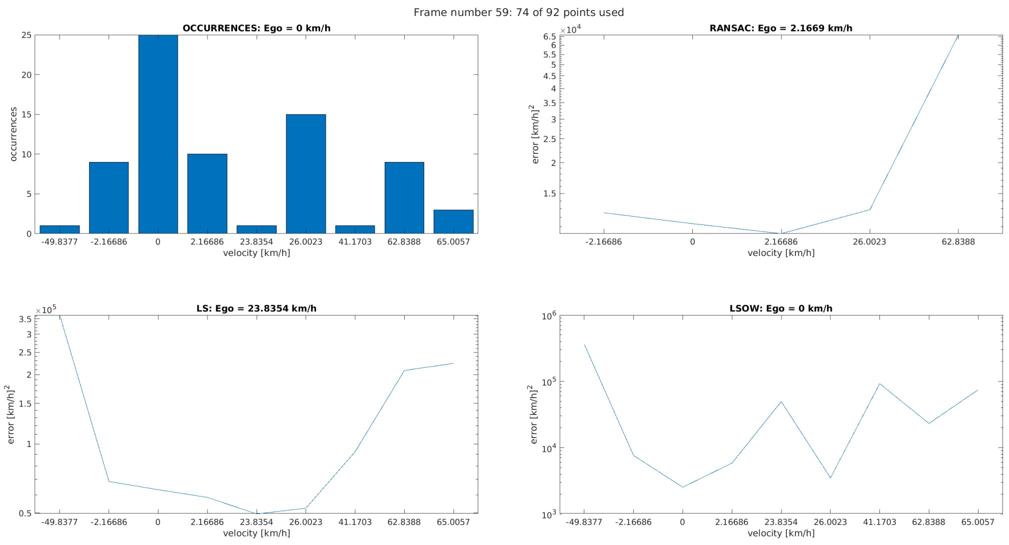


Figure 4.39: Algorithms graphical voting outcome during frame 59

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-49.8377	1	-	5.5604	5.5604
-2.16686	9	4.0975	4.8363	3.8821
0	25	4.0533	4.8007	3.4027
2.16686	10	4.0129	4.767	3.767
23.8354	1	-	4.6951	4.6951
26.0023	15	4.11	4.7201	3.544
41.1703	1	-	4.9659	4.9659
62.8388	9	4.8198	5.3193	4.3651
65.0057	3	-	5.3504	4.8732

Table 4.15: Algorithms numerical voting outcome during frame 59

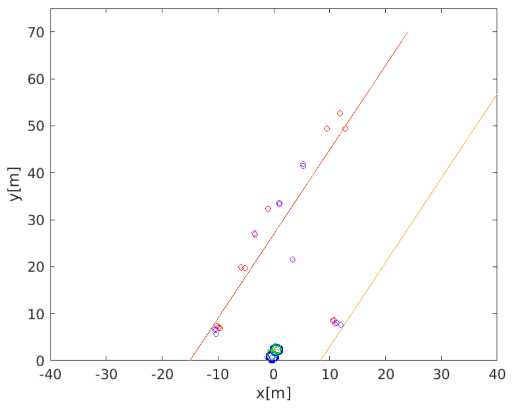


Figure 4.40: Input target cloud to the algorithms during frame 59

4.5.2 Static host, opening target: Track until midway

The snapshot frames proposed in the figures 4.42, 4.44 and 4.46, present the track for the target which started in the "Static host, opening target: Track Start" section. In this case it is presented when the targets has reached the midway between the position from which the track started and to which the track is released. The estimation error this time affects the both the "*LS*" and "*RANSAC*" implementations, and it happened during two frames as can be seen on figure 4.41 and 4.45.

By observing this snapshot, the *Tracking* unit correctly keeps keeps track on a moving *Target* when no *Ego Velocity* estimation error occurs. The tracker behavior is like intended when a tracked *Target* is already present and the *Host* is stationary.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity, where the impact of the *Target* is not negligible because its large velocity made the *Mean* velocity shift away from the *Mode*.

The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

4.5.2.1 Frame 108

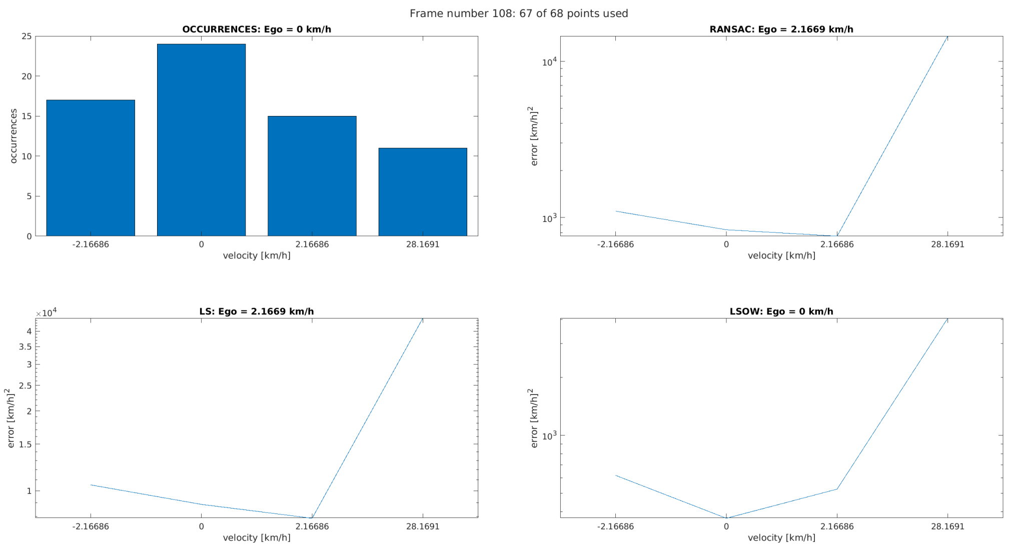


Figure 4.41: Algorithms graphical voting outcome during frame 108

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	17	3.0409	4.0219	2.7915
0	24	2.9221	3.9484	2.5681
2.16686	15	2.8812	3.8959	2.7198
28.1691	11	4.1616	4.6516	3.6102

Table 4.16: Algorithms numerical voting outcome during frame 108

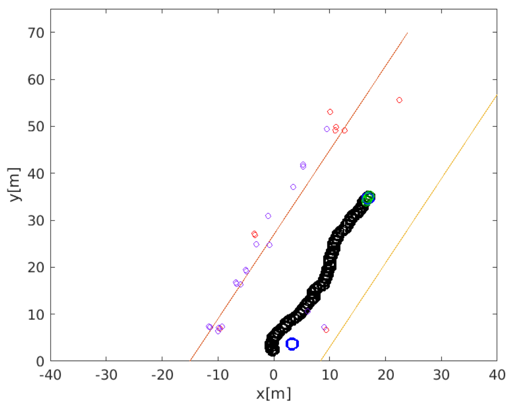


Figure 4.42: Input target cloud to the algorithms during frame 108

4.5.2.2 Frame 109

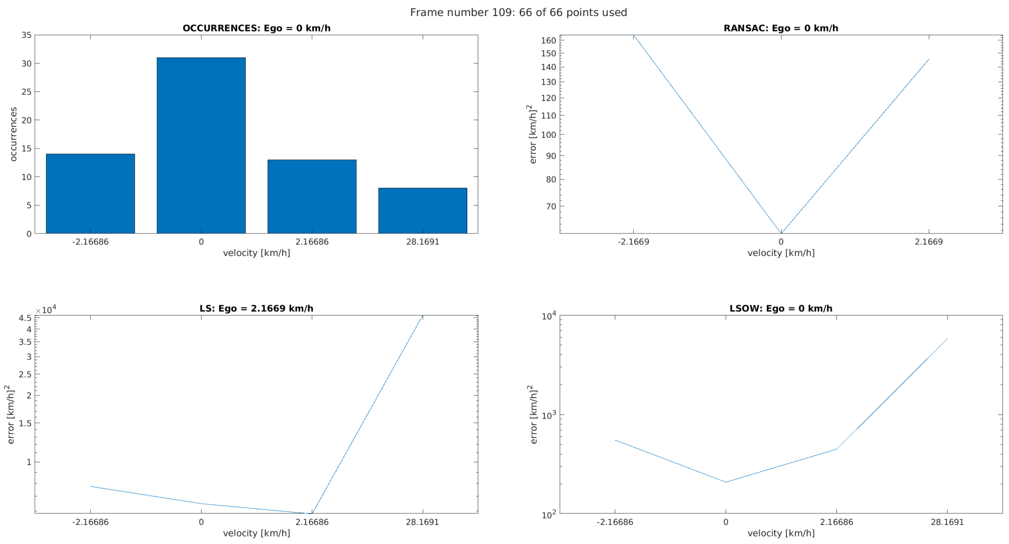


Figure 4.43: Algorithms graphical voting outcome during frame 109

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	14	2.2157	3.8894	2.7433
0	31	1.7856	3.8112	2.3199
2.16686	13	2.163	3.7647	2.6508
28.1691	8	-	4.6653	3.7622

Table 4.17: Algorithms numerical voting outcome during frame 109

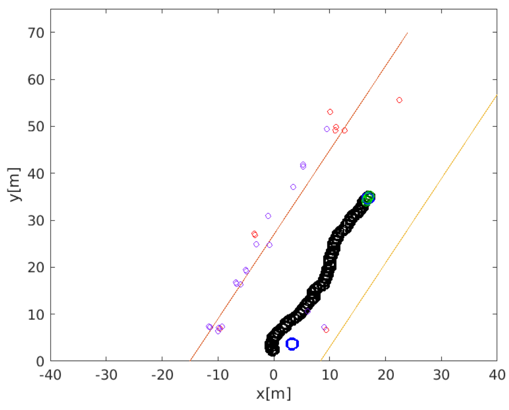


Figure 4.44: Input target cloud to the algorithms during frame 109

4.5.2.3 Frame 110

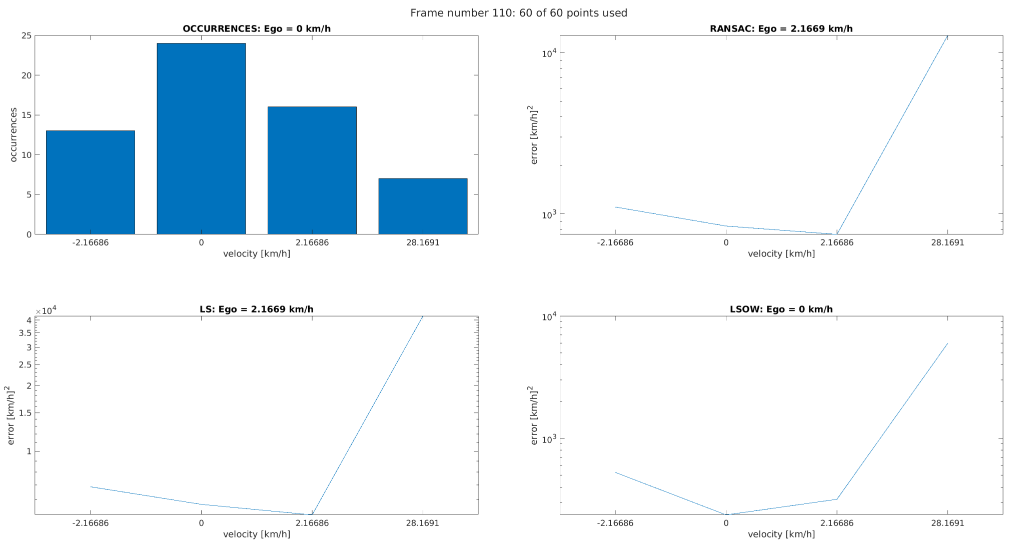


Figure 4.45: Algorithms graphical voting outcome during frame 110

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	13	3.0427	3.836	2.7221
0	24	2.9245	3.7552	2.375
2.16686	16	2.8731	3.7067	2.5026
28.1691	7	4.1073	4.6214	3.7763

Table 4.18: Algorithms numerical voting outcome during frame 110

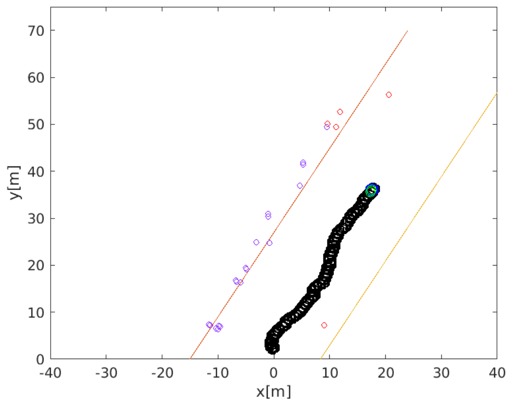


Figure 4.46: Input target cloud to the algorithms during frame 110

4.5.3 Static host, opening target: Track End

The release of the target can be seen in the frames proposed in the figures 4.48, 4.50 and 4.52. The release occurs because the target has escaped the field of view of the *Radar* beyond the *Maximum Range*, thus the *Track* resources can be released for later assignment to new tracks. In these frames all the implementations conform to estimating the *Host* velocity as $0\,\mathrm{km\,h^{-1}}$.

By observing this snapshot, the *Tracking* unit correctly releases the track on a *Target* when it is no longer in the field of view. The *Ego Velocity* estimation routine has no effect on a *Track* which is released for the reason mentioned above. The tracker behavior is like intended when a previously tracked *Target* escapes the field of view and the *Host* is stationary.

4.5.3.1 Frame 158

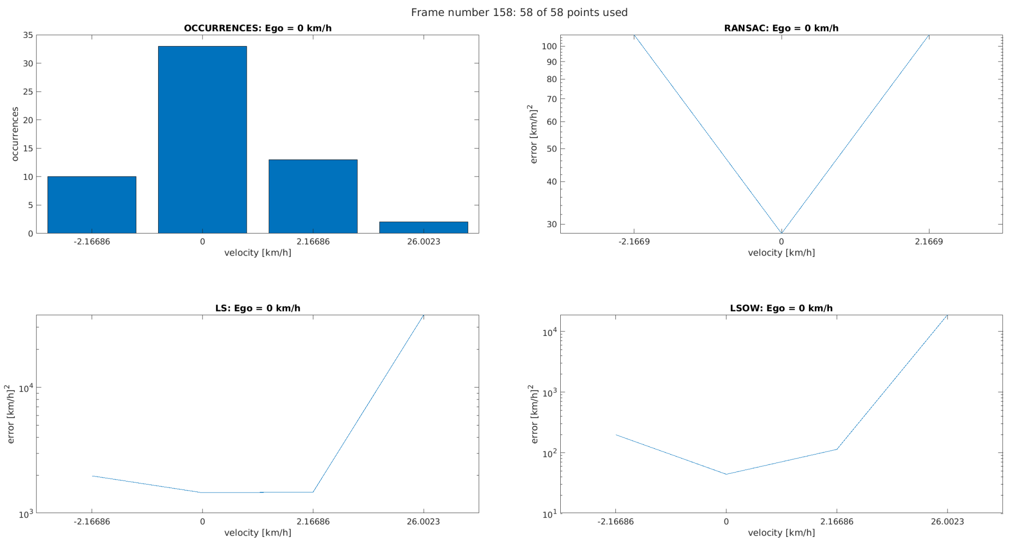


Figure 4.47: Algorithms graphical voting outcome during frame 158

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.16686	10	2.0334	3.298	2.298
0	33	1.4498	3.1644	1.6459
2.16686	13	2.0334	3.17	2.056
26.0023	2	-	4.5756	4.2745

Table 4.19: Algorithms numerical voting outcome during frame 158

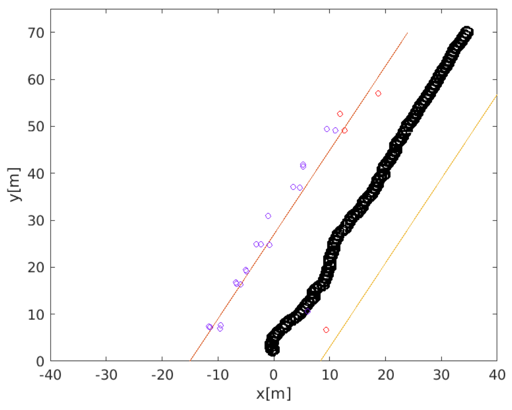


Figure 4.48: Input target cloud to the algorithms during frame 158

4.5.3.2 Frame 159

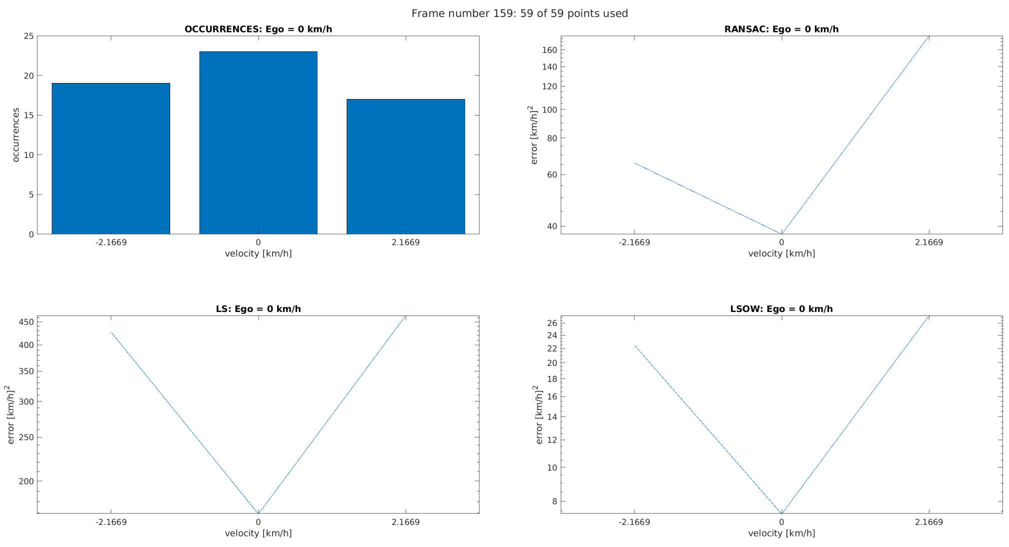


Figure 4.49: Algorithms graphical voting outcome during frame 159

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	19	1.8178	2.6307	1.3519
0	23	1.5748	2.228	0.86623
2.1669	17	2.2514	2.6673	1.4368

Table 4.20: Algorithms numerical voting outcome during frame 159

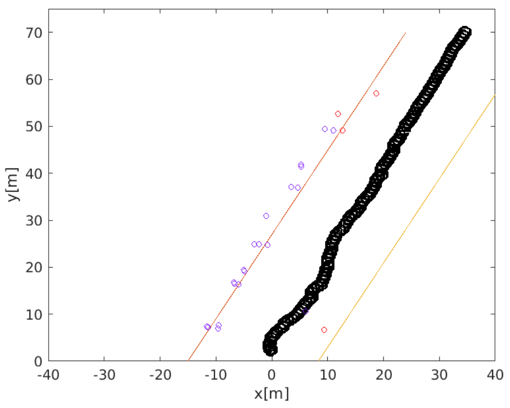


Figure 4.50: Input target cloud to the algorithms during frame 159

4.5.3.3 Frame 160

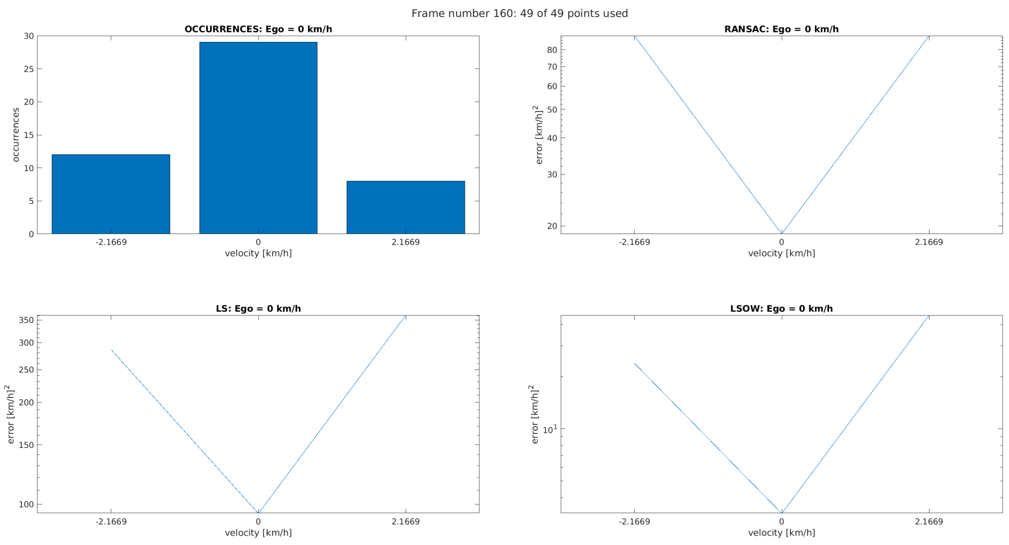


Figure 4.51: Algorithms graphical voting outcome during frame 160

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	12	1.9504	2.457	1.3778
0	29	1.2737	1.9727	0.51029
2.1669	8	1.9504	2.5582	1.6551

Table 4.21: Algorithms numerical voting outcome during frame 160

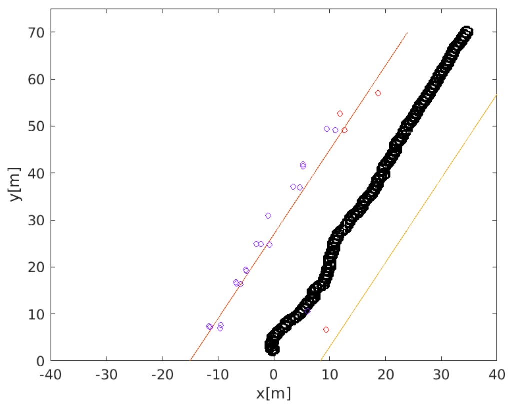


Figure 4.52: Input target cloud to the algorithms during frame 160

4.6 Accelerating host, background acquisition

For this acquisition the *Host* car was driven from an initial standstill start, up to about $10\,\mathrm{km\,h^{-1}}$. The car was uniformly accelerated as far as possible. The *Host* car moved along a straight line, in the middle of the roadway. The proposed set of frames is taken at every linear velocity increase corresponding to a resolution step. The following frames show that all the algorithm implementations conform to estimating the same *Host* velocity during each frame. Thus all the explored implementations can sustain a velocity change on the *Host*.

By observing this snapshot, all the *Ego Estimation* algorithms behave like no estimation algorithm was applied to the original *Radar Software Stack*, like intended when no *Target* is present and the *Host* is accelerating. Furthermore, now the *Tracking* unit does not allocate any track when the *Host* is moving, even in case of a uniform acceleration.

4.6.1 Frame 10

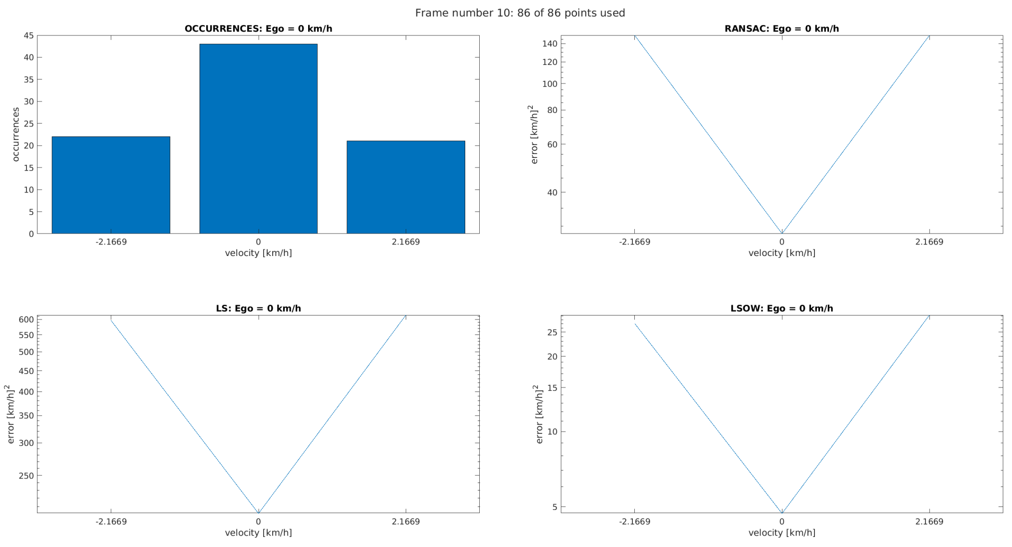


Figure 4.53: Algorithms graphical voting outcome during frame 10

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-2.1669	22	2.1768	2.7755	1.433
0	43	1.4498	2.3051	0.67166
2.1669	21	2.1768	2.7889	1.4667

Table 4.22: Algorithms numerical voting outcome during frame 10

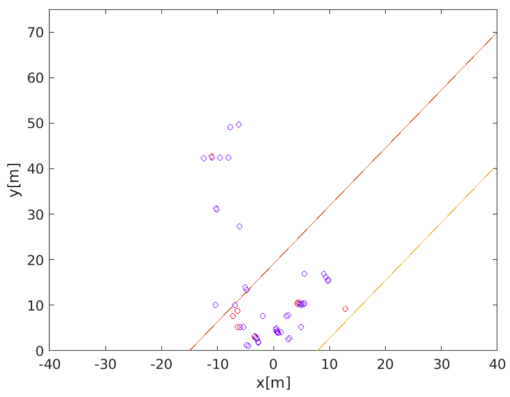


Figure 4.54: Input target cloud to the algorithms during frame 10

4.6.2 Frame 11

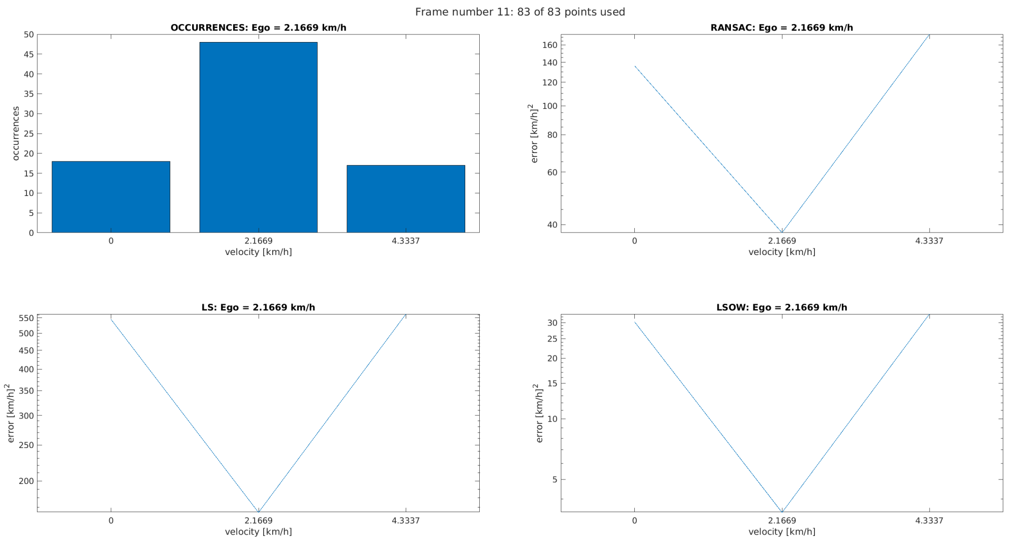


Figure 4.55: Algorithms graphical voting outcome during frame 11

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
0	18	2.1341	2.7361	1.4808
2.1669	48	1.5748	2.2157	0.53449
4.3337	17	2.2399	2.7508	1.5204

Table 4.23: Algorithms numerical voting outcome during frame 11

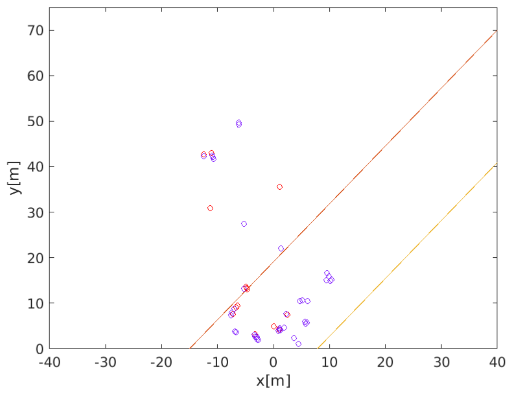


Figure 4.56: Input target cloud to the algorithms during frame 11

4.6.3 Frame 18

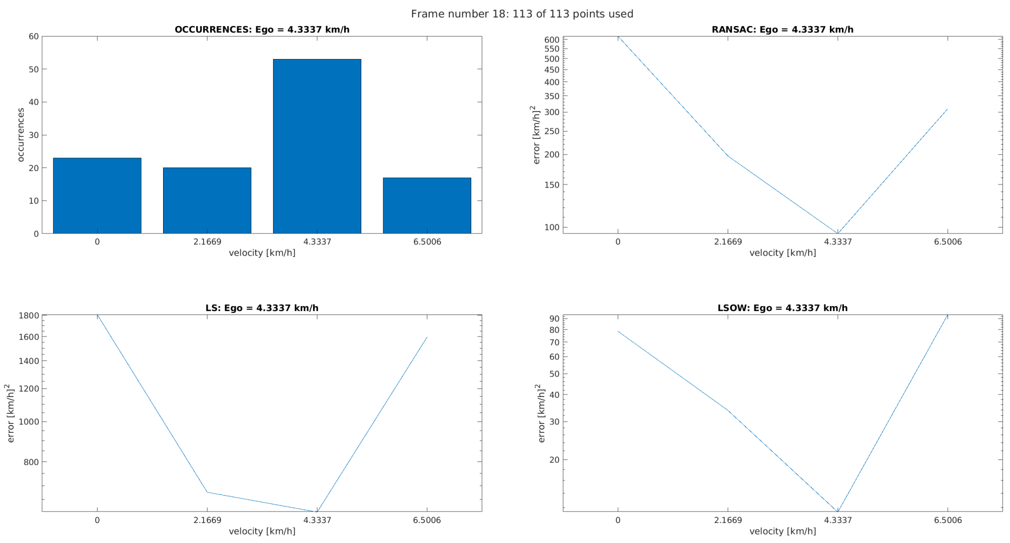


Figure 4.57: Algorithms graphical voting outcome during frame 18

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
0	23	2.7922	3.2571	1.8954
2.1669	20	2.2949	2.83	1.529
4.3337	53	1.9727	2.7822	1.058
6.5006	17	2.4912	3.2031	1.9727

Table 4.24: Algorithms numerical voting outcome during frame 18

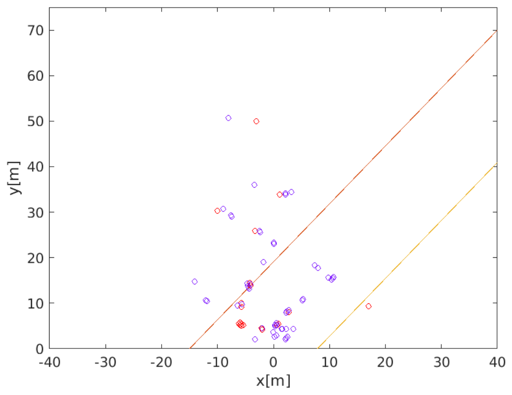


Figure 4.58: Input target cloud to the algorithms during frame 18

4.6.4 Frame 22

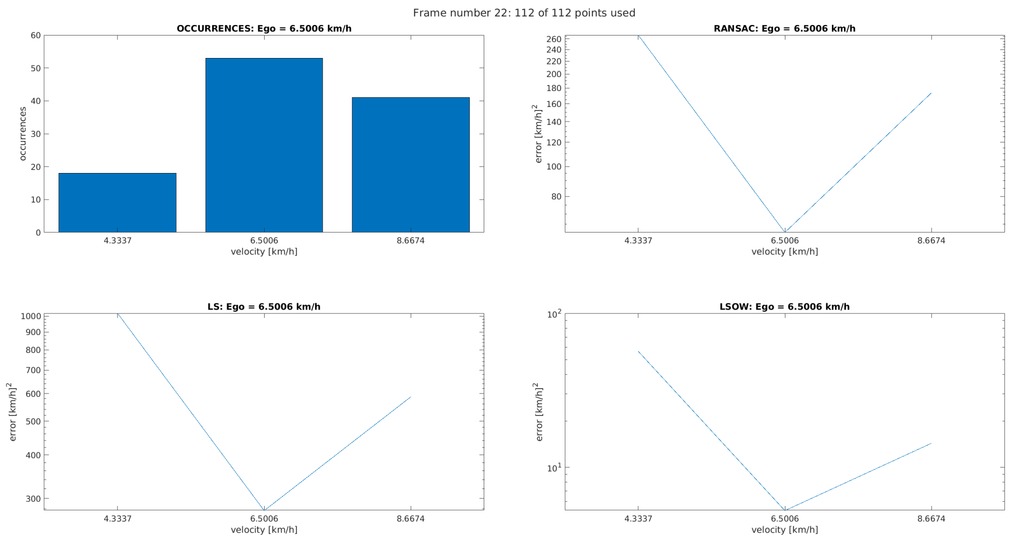


Figure 4.59: Algorithms graphical voting outcome during frame 22

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
4.3337	18	2.4275	3.0081	1.7528
6.5006	53	1.7856	2.4425	0.71824
8.6674	41	2.2399	2.7686	1.1558

Table 4.25: Algorithms numerical voting outcome during frame 22

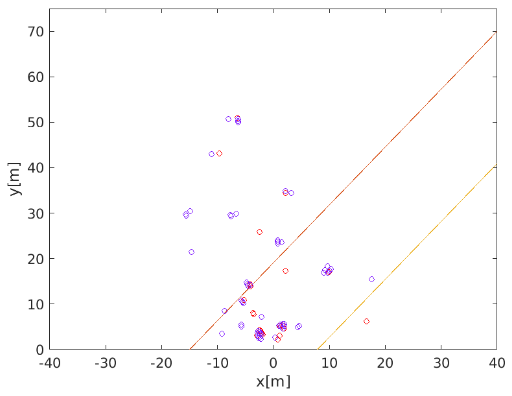


Figure 4.60: Input target cloud to the algorithms during frame 22

4.6.5 Frame 26

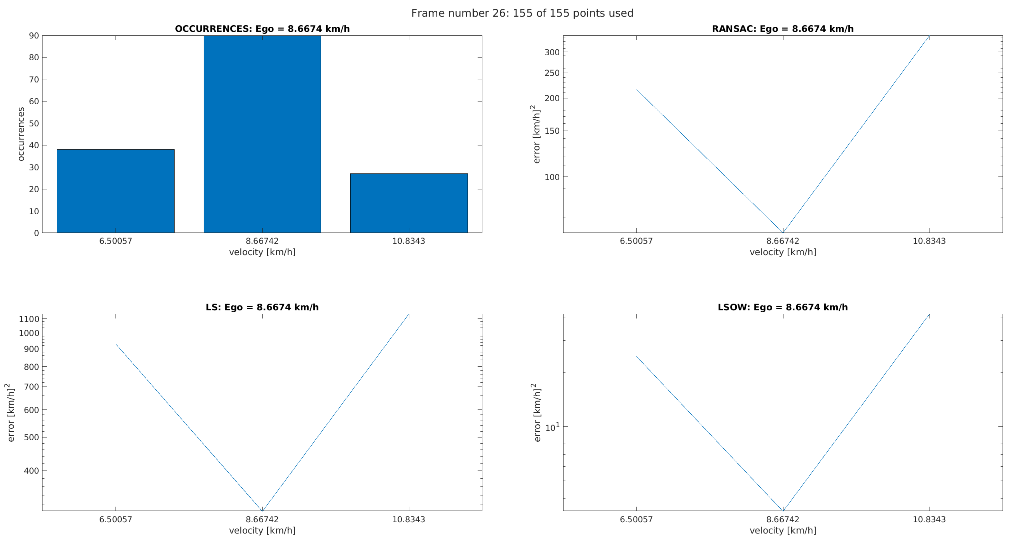


Figure 4.61: Algorithms graphical voting outcome during frame 26

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
6.50057	38	2.3344	2.9683	1.3885
8.66742	90	1.7856	2.4846	0.53033
10.8343	27	2.5409	3.0555	1.6241

Table 4.26: Algorithms numerical voting outcome during frame 26

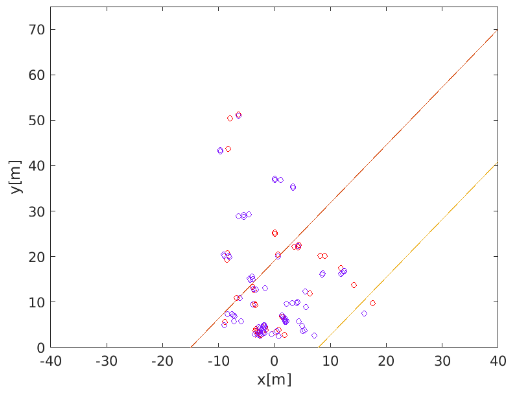


Figure 4.62: Input target cloud to the algorithms during frame 26

4.6.6 Frame 30

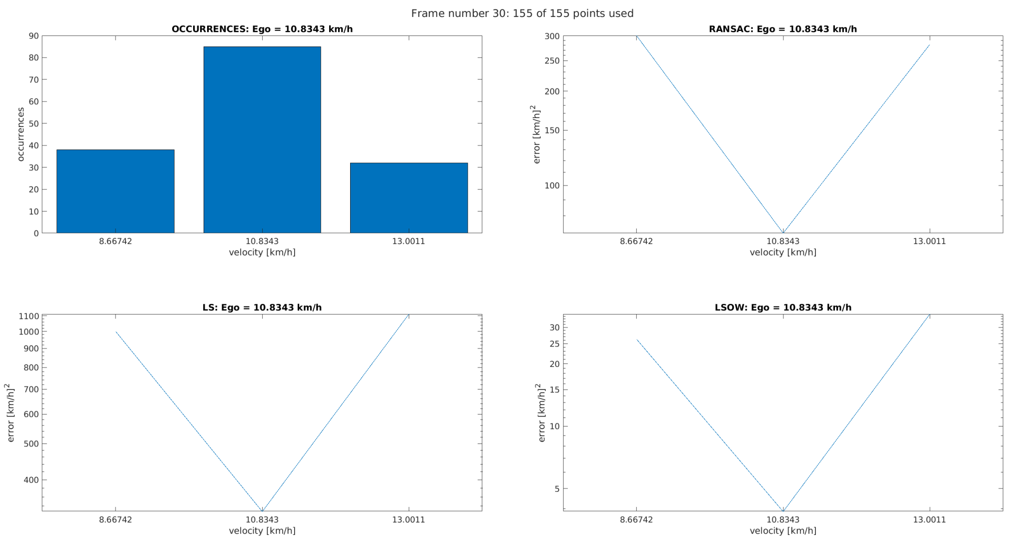


Figure 4.63: Algorithms graphical voting outcome during frame 30

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
8.66742	38	2.4778	3	1.4203
10.8343	85	1.8478	2.5168	0.58734
13.0011	32	2.4498	3.0464	1.5413

Table 4.27: Algorithms numerical voting outcome during frame 30

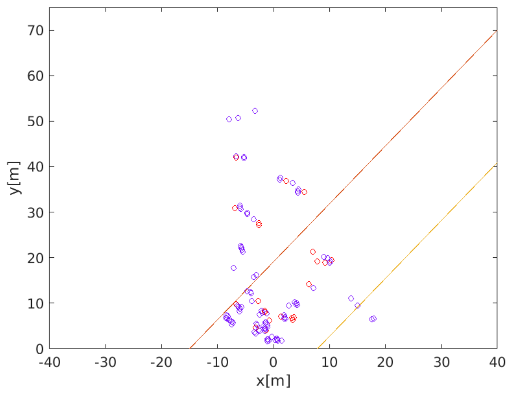


Figure 4.64: Input target cloud to the algorithms during frame 30

4.6.7 Frame 36

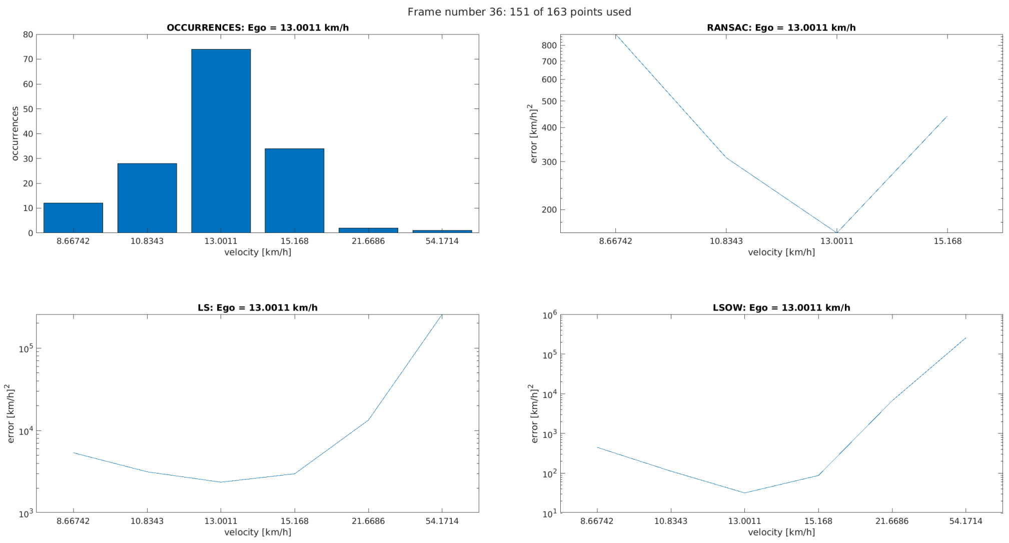


Figure 4.65: Algorithms graphical voting outcome during frame 36

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
8.66742	12	2.9435	3.7297	2.6505
10.8343	28	2.4912	3.499	2.0519
13.0011	74	2.2157	3.3732	1.504
15.168	34	2.6448	3.4751	1.9436
21.6686	2	-	4.126	3.825
54.1714	1	-	5.4094	5.4094

Table 4.28: Algorithms numerical voting outcome during frame 36

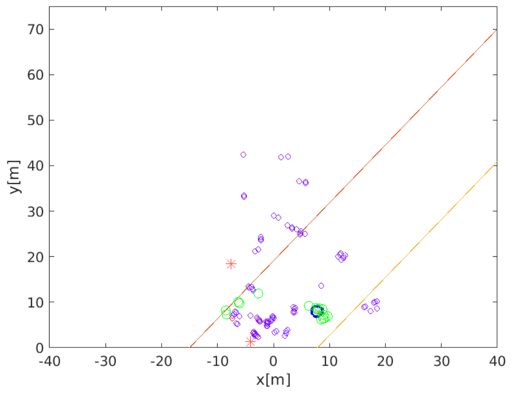


Figure 4.66: Input target cloud to the algorithms during frame 36

4.7 Dynamic host, background acquisition

For this acquisition the *Host* car was driven on a straight line along the *Host*'s left side of the road, keeping the velocity as constant as possible. The *Host* car was driven in the roadway scenery at about $10\,\mathrm{km\,h^{-1}}$. The following frames show that all the implementation conform to estimating the same *Host* velocity, except the "*LS*" implementation during frame 76. This discrepancy in shown in both figure 4.71 and table 4.31.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended when no *Ego Velocity* estimation error occurs. The desired behavior is that the *Tracking* unit won't start tracks on the moving *Targets* which belong to the scenery. Thus the absence of tracks when no *Target* is present and the *Host* is moving demonstrates that it works properly.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity.

4.7.1 Frame 74

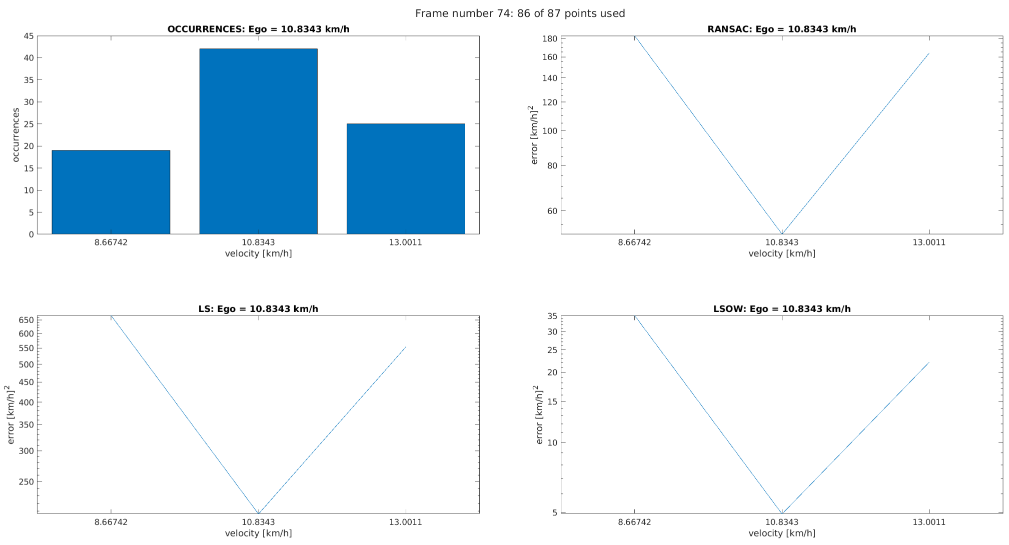


Figure 4.67: Algorithms graphical voting outcome during frame 74

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
8.66742	19	2.2627	2.8239	1.5452
10.8343	42	1.7131	2.3151	0.69186
13.0011	25	2.2157	2.7435	1.3456

Table 4.29: Algorithms numerical voting outcome during frame 74

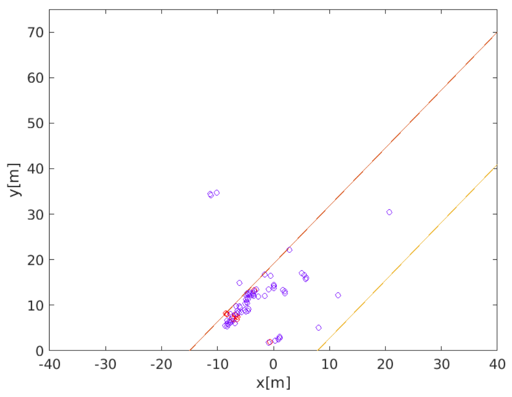


Figure 4.68: Input target cloud to the algorithms during frame 74

4.7.2 Frame 75

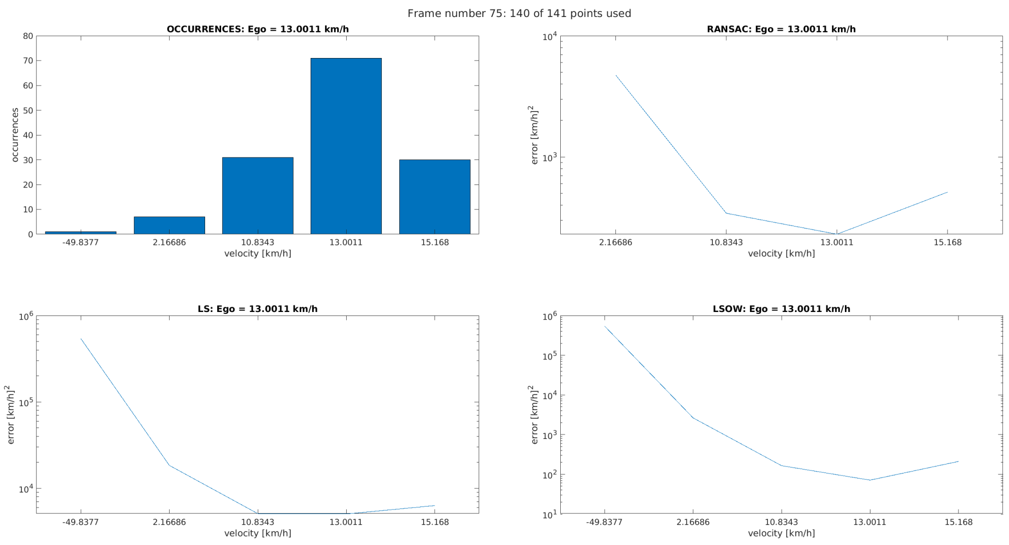


Figure 4.69: Algorithms graphical voting outcome during frame 75

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-49.8377	1	-	5.7325	5.7325
2.16686	7	3.6756	4.2657	3.4206
10.8343	31	2.535	3.7079	2.2165
13.0011	71	2.3619	3.7039	1.8526
15.168	30	2.7091	3.801	2.3239

Table 4.30: Algorithms numerical voting outcome during frame 75

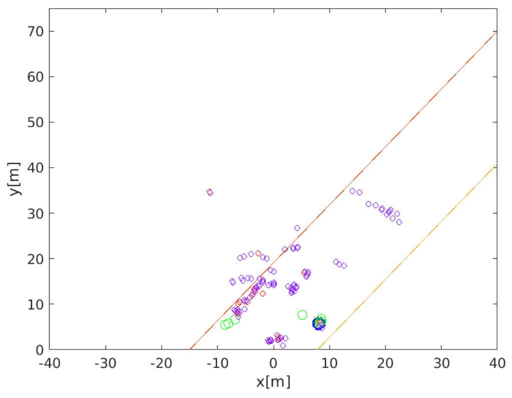


Figure 4.70: Input target cloud to the algorithms during frame 75

4.7.3 Frame 76

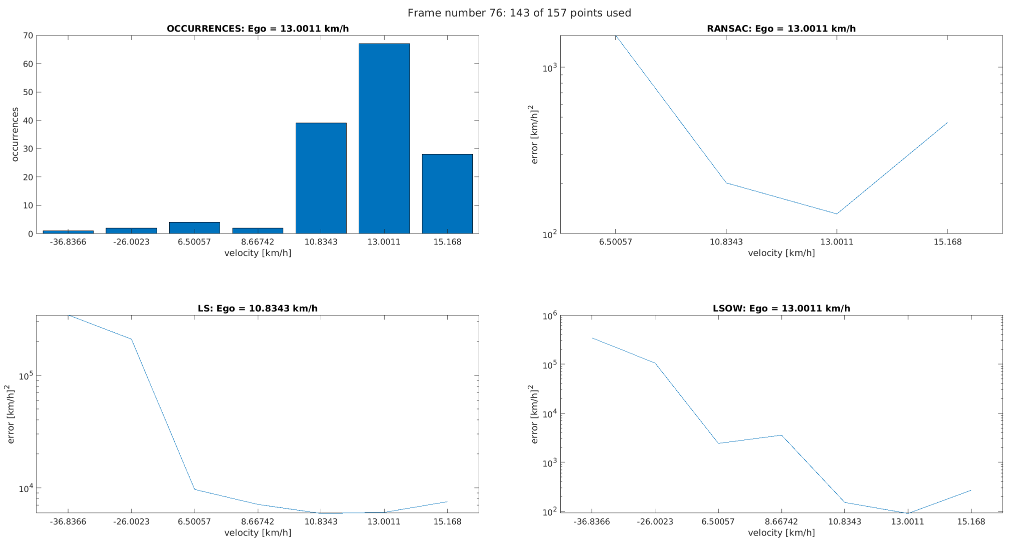


Figure 4.71: Algorithms graphical voting outcome during frame 76

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-36.8366	1	-	5.5349	5.5349
-26.0023	2	-	5.3203	5.0192
6.50057	4	3.1915	3.9853	3.3833
8.66742	2	-	3.8524	3.5513
10.8343	39	2.3051	3.7717	2.1806
13.0011	67	2.1188	3.7816	1.9555
15.168	28	2.6673	3.8766	2.4294

Table 4.31: Algorithms numerical voting outcome during frame 76

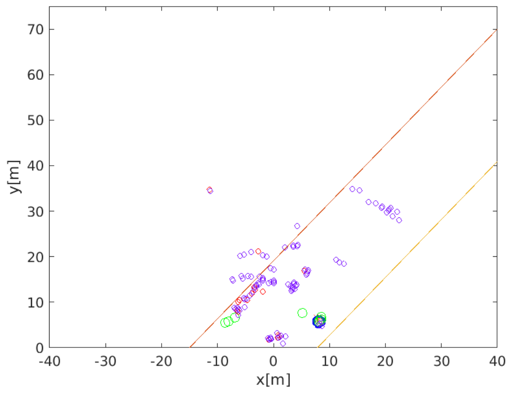


Figure 4.72: Input target cloud to the algorithms during frame 76

4.8 Dynamic host, closing target

For this acquisition the *Host* car was driven on a straight line along the *Host*'s left side of the road, keeping the velocity as constant as possible. The *Host* car was driven in the roadway scenery at about $10\,\mathrm{km\,h^{-1}}$. The incoming *Target* car was driven from a distance lesser than the *Maximum Range* due to the roadway length. The *Target* car approached at an higher velocity than the *Host* one in order to exit the *Radar* field of view during the overtake. Thus three snapshots are taken for three situations:

- when the *Radar* started a track on the *Target* which started moving from its start position inside the field of view

- when the *Target* is at a medium distance (with respect to the one occurred when the track started)

- when the *Radar* built the last tracks before releasing them because the *Target* is out of sight after the overtake of the *Host*

The *Target* car used in this test has been driven towards the *Host* car from a standstill start to about $30\,\mathrm{km\,h^{-1}}$.

4.8.1 Dynamic host, closing target: Track Start

The allocation of a track for a closing target can be seen in the frames proposed in the figures 4.74, 4.76 and 4.78. The estimation of the "*LS*" algorithm is the one which is affected by an error during all the proposed frames, as can be seen in figure 4.73, 4.75, 4.77 and their respective tables 4.32, 4.33 and 4.34.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended when no *Ego Velocity* estimation error occurs. The desired behavior is that the *Tracking* unit correctly starts a track on a moving *Target* when the *Host* is moving, but only tracks the ones which do not belong to the scenery. Thus the presence of a track only on a true *Target* demonstrates that it works properly.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity.

4.8.1.1 Frame 156

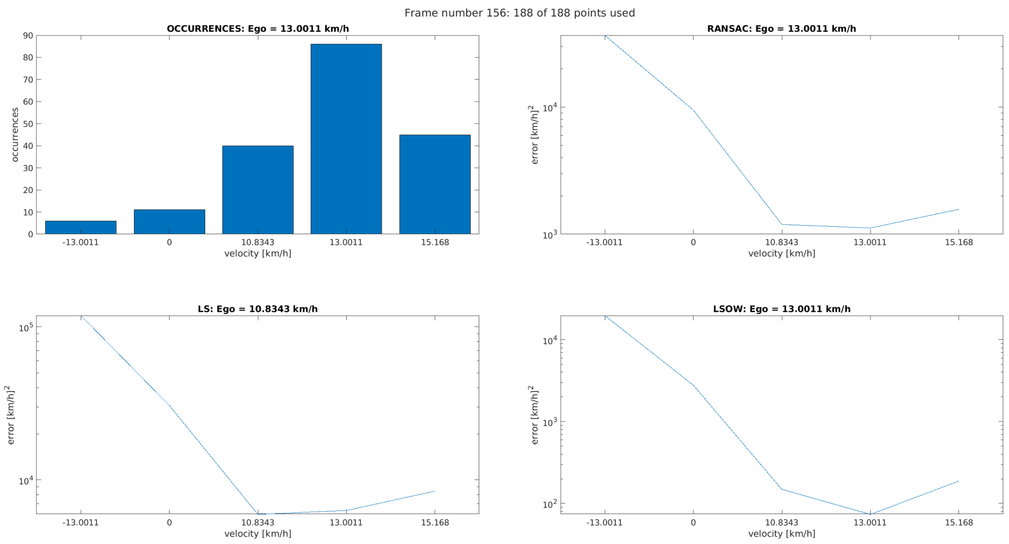


Figure 4.73: Algorithms graphical voting outcome during frame 156

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-13.0011	6	4.565	5.0735	4.2953
0	11	3.9757	4.4857	3.4443
10.8343	40	3.0765	3.7744	2.1724
13.0011	86	3.0482	3.8004	1.8659
15.168	45	3.1954	3.9267	2.2735

Table 4.32: Algorithms numerical voting outcome during frame 156

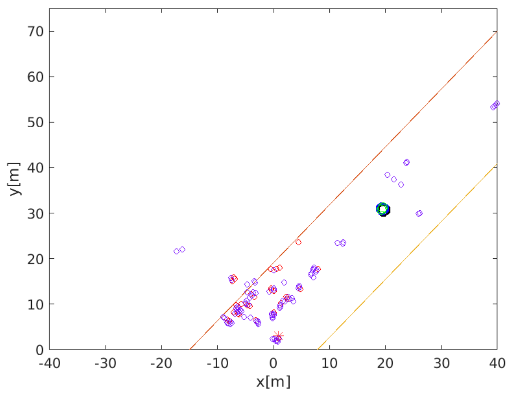


Figure 4.74: Input target cloud to the algorithms during frame 156

4.8.1.2 Frame 157

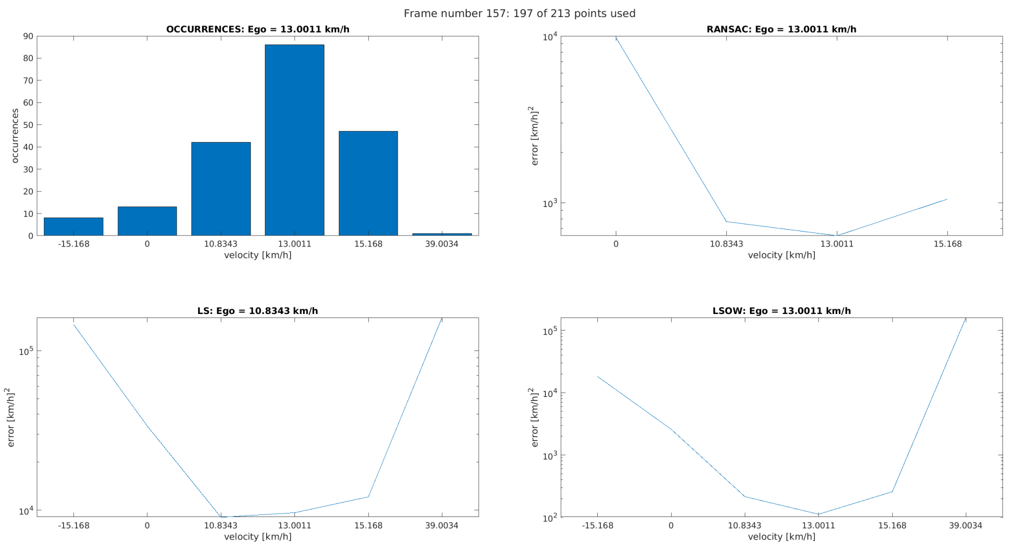


Figure 4.75: Algorithms graphical voting outcome during frame 157

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-15.168	8	-	5.1638	4.2607
0	13	3.9895	4.5269	3.4129
10.8343	42	2.8865	3.955	2.3317
13.0011	86	2.802	3.984	2.0496
15.168	47	3.0219	4.0833	2.4112
39.0034	1	-	5.208	5.208

Table 4.33: Algorithms numerical voting outcome during frame 157

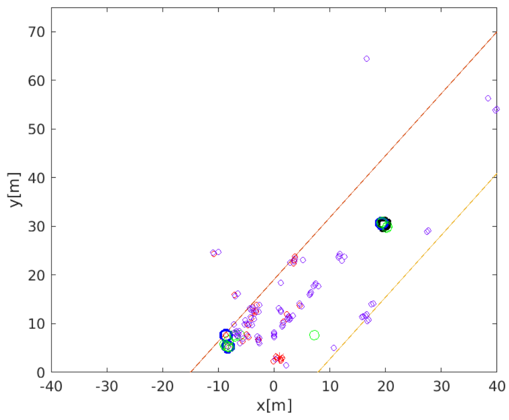


Figure 4.76: Input target cloud to the algorithms during frame 157

4.8.1.3 Frame 158

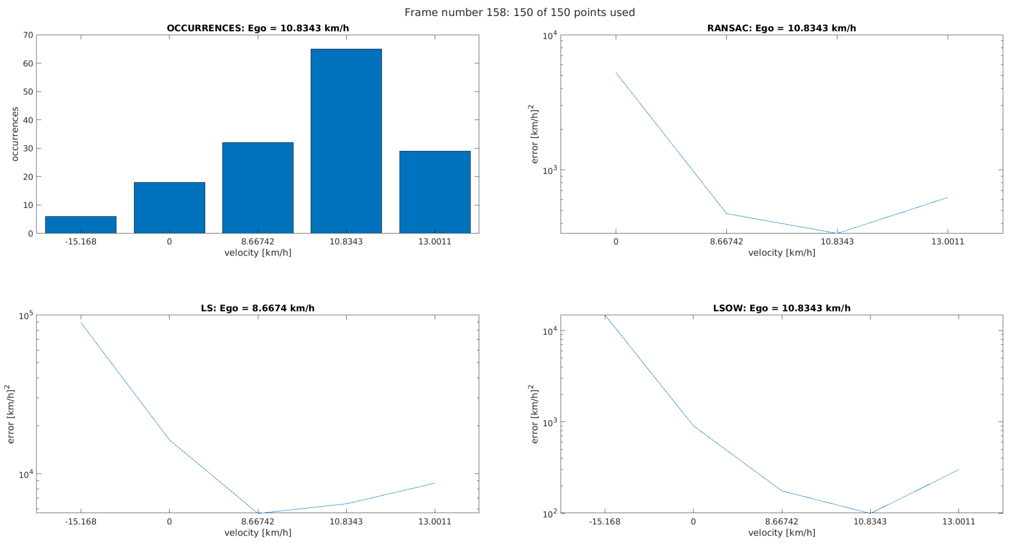


Figure 4.77: Algorithms graphical voting outcome during frame 158

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-15.168	6	-	4.9508	4.1726
0	18	3.7197	4.2126	2.9573
8.66742	32	2.676	3.749	2.2439
10.8343	65	2.529	3.81	1.997
13.0011	29	2.7955	3.94	2.4776

Table 4.34: Algorithms numerical voting outcome during frame 158

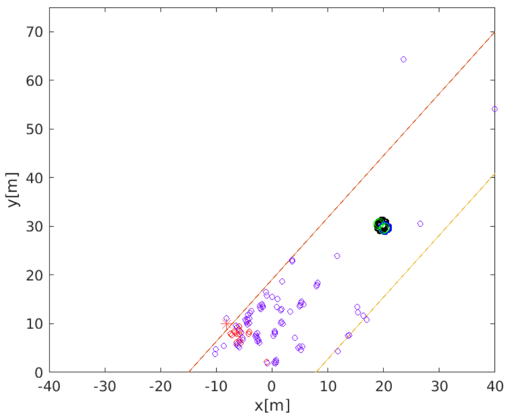


Figure 4.78: Input target cloud to the algorithms during frame 158

4.8.2 Dynamic host, closing target: Track until midway

The snapshot frames proposed in the figures 4.80, 4.82 and 4.84, present the track for the target which started in the "Dynamic host, closing target: Track Start" section. In this case it is presented when the targets has reached the midway between the position from which the track started and to which the track is released. For this acquired snapshot, all the algorithm implementations conform to estimating the same *Host* velocity during each frame.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended. The desired behavior is that the *Tracking* unit correctly keep the track on a moving *Target* when the *Host* is moving, but only tracks the ones which do not belong to the scenery. Thus the presence of a track only on a true *Target* demonstrates that it works properly.

4.8.2.1 Frame 184

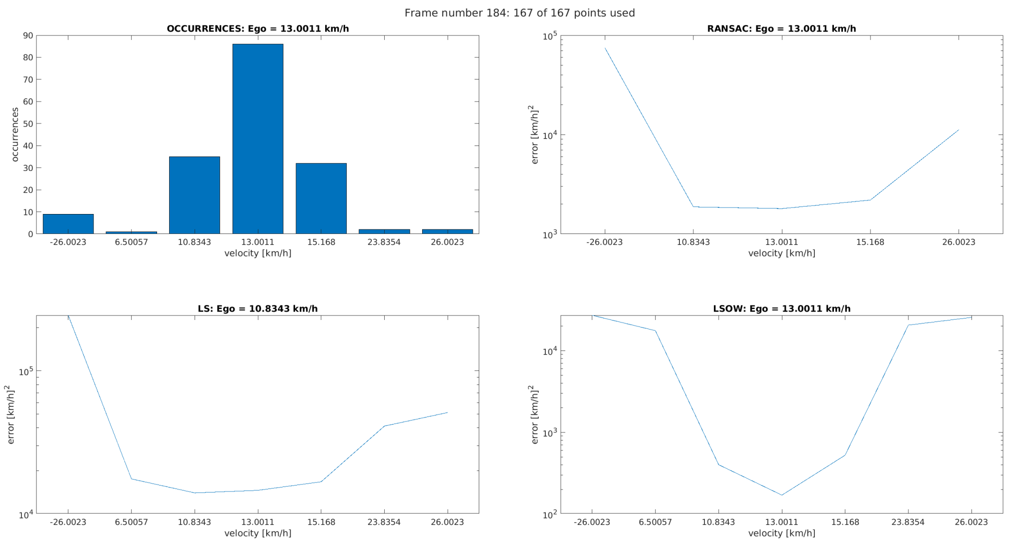


Figure 4.79: Algorithms graphical voting outcome during frame 184

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	9	4.875	5.3874	4.4331
6.50057	1	-	4.2446	4.2446
10.8343	35	3.2726	4.1472	2.6031
13.0011	86	3.2549	4.165	2.2305
15.168	32	3.341	4.2247	2.7195
23.8354	2	-	4.6136	4.3126
26.0023	2	4.0495	4.7082	4.4072

Table 4.35: Algorithms numerical voting outcome during frame 184

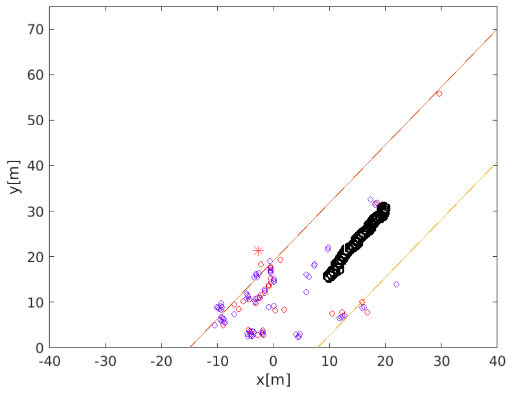


Figure 4.80: Input target cloud to the algorithms during frame 184

4.8.2.2 Frame 185

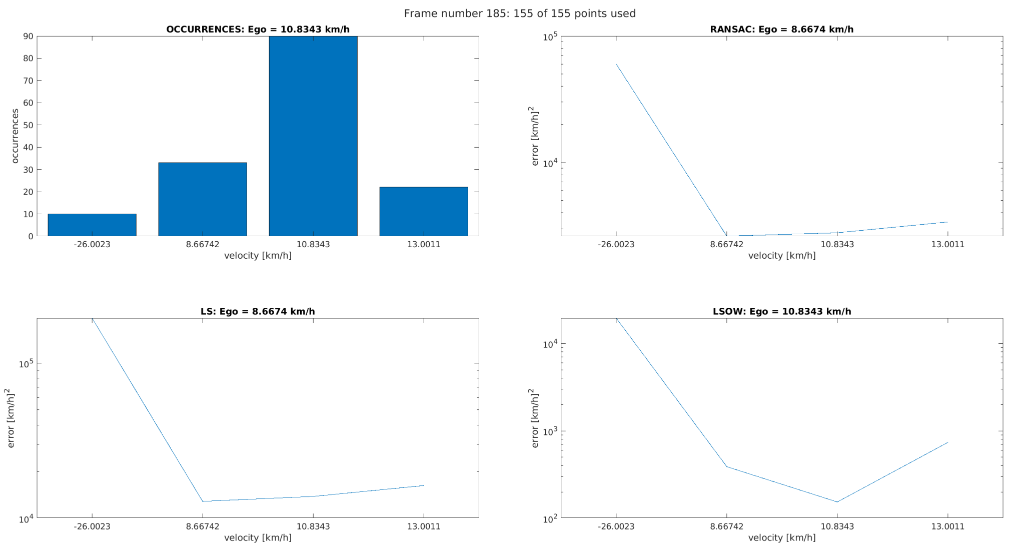


Figure 4.81: Algorithms graphical voting outcome during frame 185

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	10	4.7783	5.2906	4.2906
8.66742	33	3.4183	4.1091	2.5906
10.8343	90	3.4447	4.1407	2.1865
13.0011	22	3.5302	4.211	2.8686

Table 4.36: Algorithms numerical voting outcome during frame 185

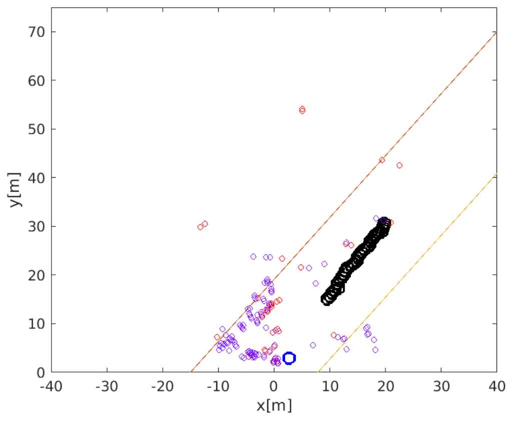


Figure 4.82: Input target cloud to the algorithms during frame 185

4.8.2.3 Frame 186

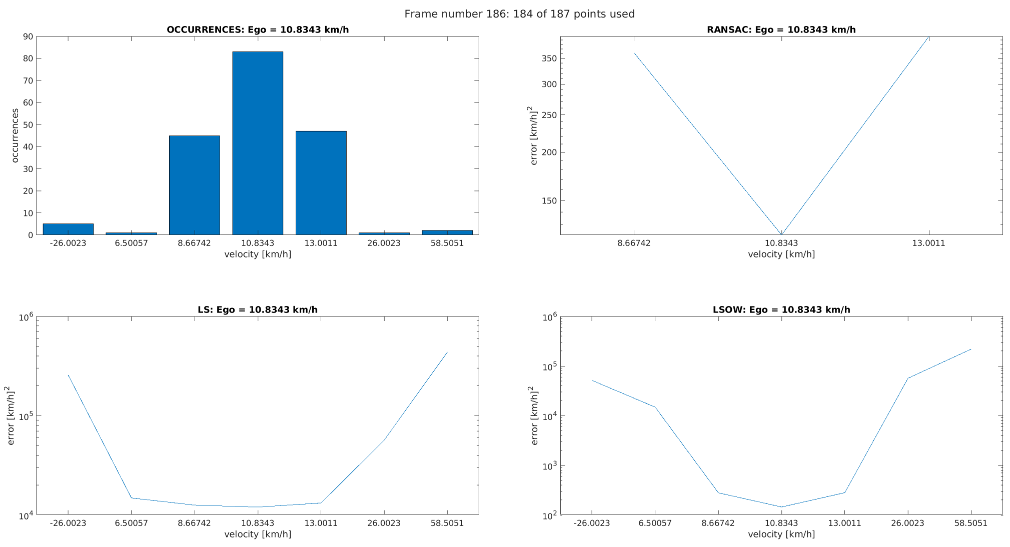


Figure 4.83: Algorithms graphical voting outcome during frame 186

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-26.0023	5	-	5.4087	4.7097
6.50057	1	-	4.1711	4.1711
8.66742	45	2.5582	4.0988	2.4456
10.8343	83	2.0866	4.0796	2.1605
13.0011	47	2.6011	4.1204	2.4483
26.0023	1	-	4.7526	4.7526
58.5051	2	-	5.6407	5.3396

Table 4.37: Algorithms numerical voting outcome during frame 186

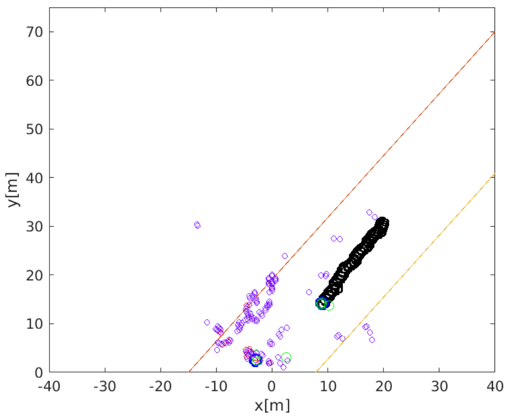


Figure 4.84: Input target cloud to the algorithms during frame 186

4.8.3 Dynamic host, closing target: Track End

The release of the target can be seen in the frames proposed in the figures 4.86, 4.88, 4.90 and their tables (4.38, 4.39, 4.40). The release occurs because the target has escaped the field of view of the *Radar*, thus the *Track* resources can be released for later assignment to new tracks. During this snapshot acquisition, the "*LS*" algorithm exhibits an estimation error during all the frames. Also the "*RANSAC*" algorithm was affected by an estimation error, but only during frames 215 and 216.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended when no *Ego Velocity* estimation error occurs. The desired behavior is that the *Tracking* unit correctly releases a track on a *Target* when the *Host* is moving. This situation occurs when the *Target* either exits the field of view or it stops, thus becoming a scenery point.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity. The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

4.8.3.1 Frame 214

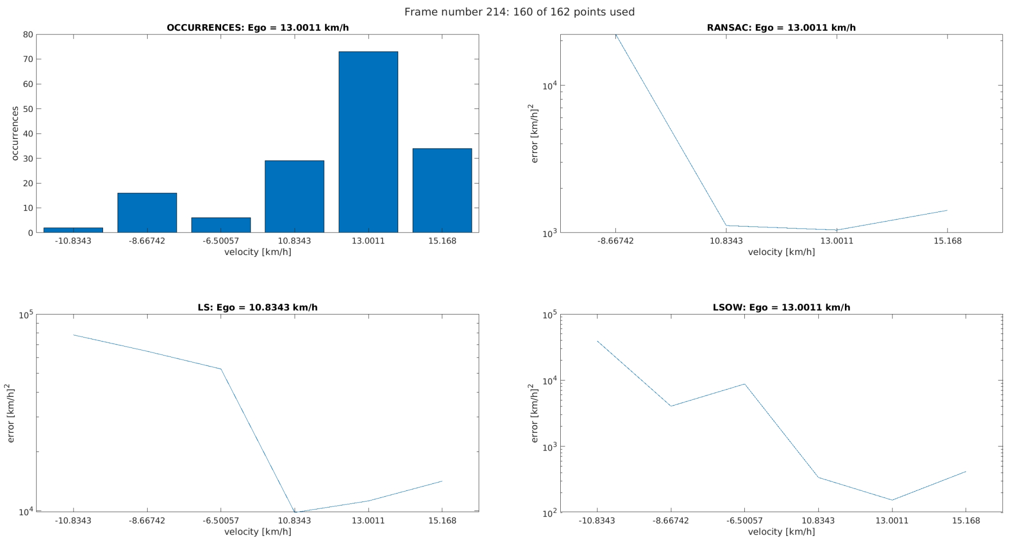


Figure 4.85: Algorithms graphical voting outcome during frame 214

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-10.8343	2	-	4.8936	4.5925
-8.66742	16	4.3439	4.8106	3.6065
-6.50057	6	-	4.7206	3.9425
10.8343	29	3.0482	3.9916	2.5292
13.0011	73	3.018	4.0502	2.1869
15.168	34	3.1517	4.1507	2.6192

Table 4.38: Algorithms numerical voting outcome during frame 214

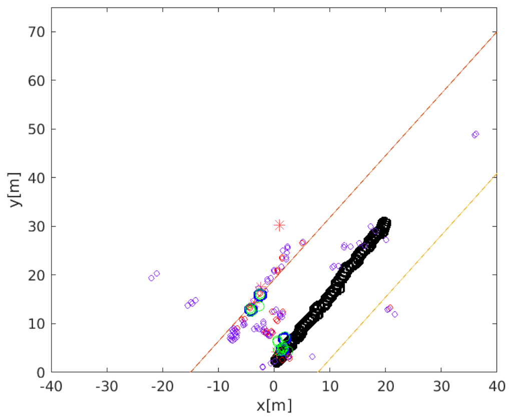


Figure 4.86: Input target cloud to the algorithms during frame 214

4.8.3.2 Frame 215

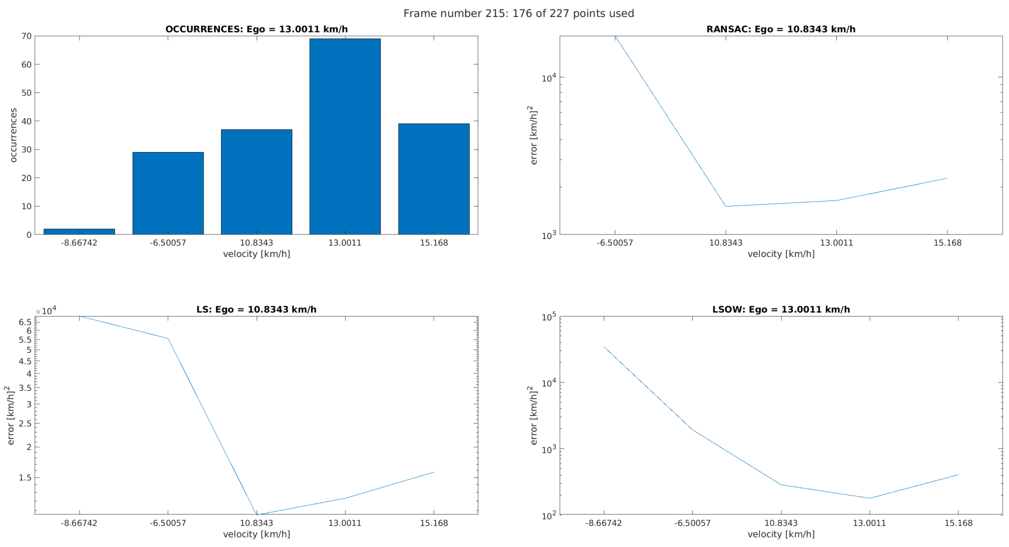


Figure 4.87: Algorithms graphical voting outcome during frame 215

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-8.66742	2	-	4.8373	4.5363
-6.50057	29	4.2634	4.7457	3.2833
10.8343	37	3.1795	4.0225	2.4543
13.0011	69	3.217	4.0908	2.2519
15.168	39	3.3583	4.1979	2.6068

Table 4.39: Algorithms numerical voting outcome during frame 215

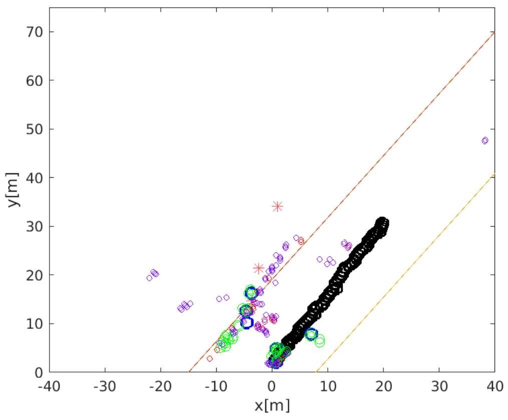


Figure 4.88: Input target cloud to the algorithms during frame 215

4.8.3.3 Frame 216

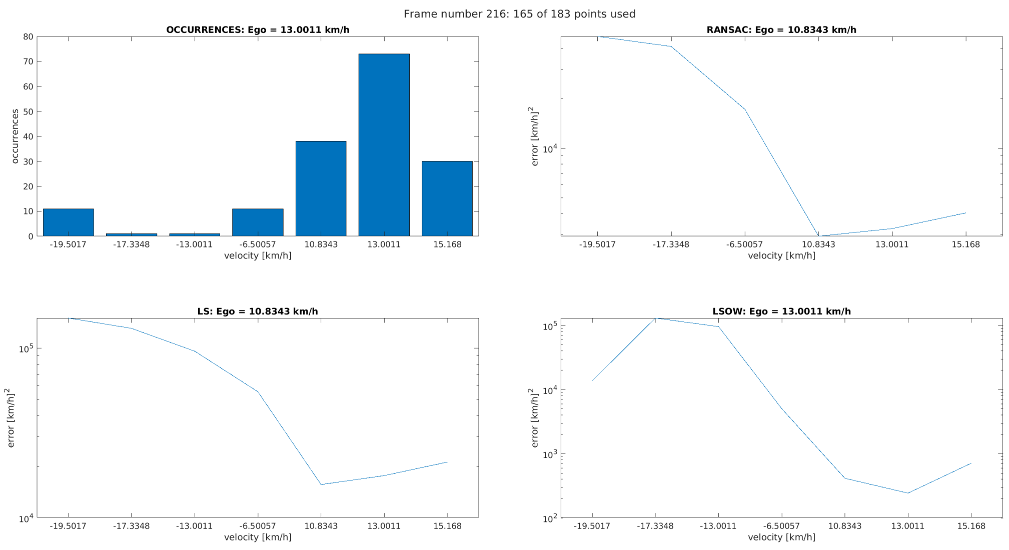


Figure 4.89: Algorithms graphical voting outcome during frame 216

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-19.5017	11	4.6778	5.1763	4.1349
-17.3348	1	4.6167	5.1152	5.1152
-13.0011	1	-	4.9809	4.9809
-6.50057	11	4.2354	4.7426	3.7012
10.8343	38	3.4655	4.1958	2.616
13.0011	73	3.5118	4.2485	2.3851
15.168	30	3.6072	4.3282	2.8511

Table 4.40: Algorithms numerical voting outcome during frame 216

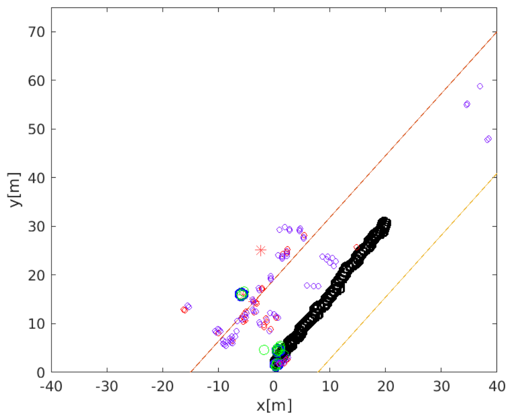


Figure 4.90: Input target cloud to the algorithms during frame 216

4.9 Dynamic host, opening target

For this acquisition the *Host* car was driven on a straight line along the *Host*'s left side of the road, keeping the velocity as constant as possible. The *Host* car was driven in the roadway scenery at about $10\,\mathrm{km\,h^{-1}}$. The incoming *Target* car was driven from a close distance, entering the *Radar* field of view from the right side of the *Host* car. The *Target* was then driven until it reached a distance of about 60 m, which approaches the *Maximum Range*. A total of three snapshots are taken for the following three situations.

- when the *Radar* started a track on the *Target* which entered the field of view
- when the *Target* is at a medium distance (with respect to the one occurred when the track started)
- when the *Radar* built the last tracks due to the *Target* reaching the wanted point

The *Target* car used in this test has been driven away from the *Host* car at about $30\,\mathrm{km\,h^{-1}}$.

4.9.1 Dynamic host, opening target: Track Start

The allocation of a track for an opening target can be seen in the frames proposed in the figures 4.92, 4.94 and 4.96. The estimation of the "*LS*" and "*RANSAC*" algorithms are affected by an error during all the first two proposed frames, as can be seen in figure 4.91, 4.93, and their respective tables 4.41, 4.42.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended when no *Ego Velocity* estimation error occurs. The desired behavior is that the *Tracking* unit correctly starts a track on a moving *Target* which enters the field of view when the *Host* is moving, but only tracks the ones which do not belong to the scenery. Thus the presence of a track only on a true *Target* demonstrates that it works properly.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity. The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

4.9.1.1 Frame 135

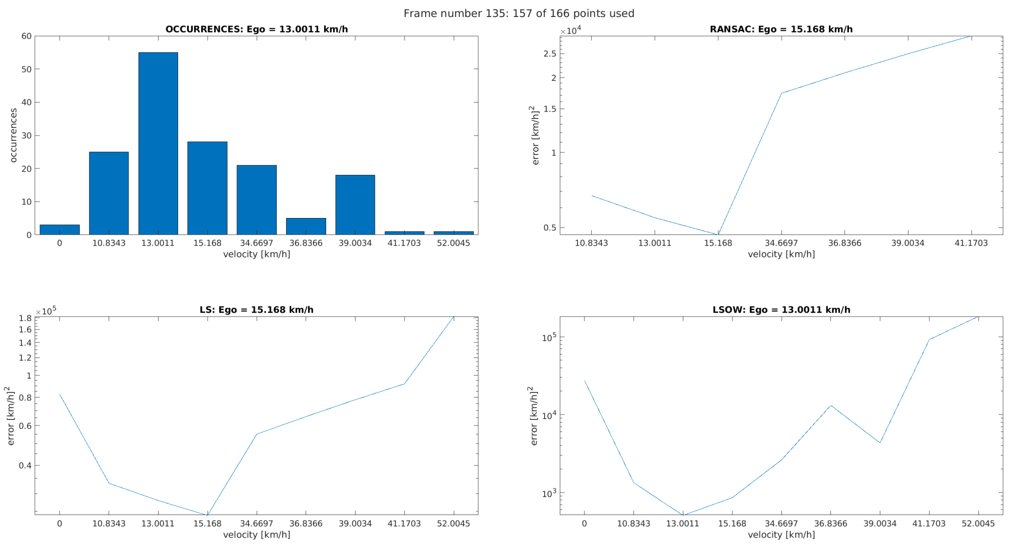


Figure 4.91: Algorithms graphical voting outcome during frame 135

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
0	3	-	4.9164	4.4392
10.8343	25	3.827	4.523	3.1251
13.0011	55	3.738	4.4463	2.7059
15.168	28	3.669	4.3804	2.9333
34.6697	21	4.2381	4.7403	3.418
36.8366	5	4.3204	4.8182	4.1193
39.0034	18	4.3973	4.8926	3.6373
41.1703	1	4.4692	4.9631	4.9631
52.0045	1	-	5.2619	5.2619

Table 4.41: Algorithms numerical voting outcome during frame 135

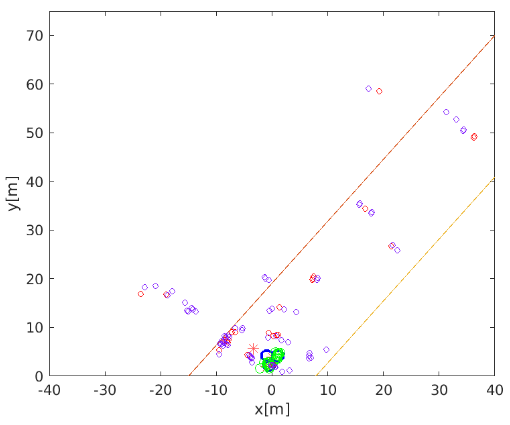


Figure 4.92: Input target cloud to the algorithms during frame 135

4.9.1.2 Frame 136

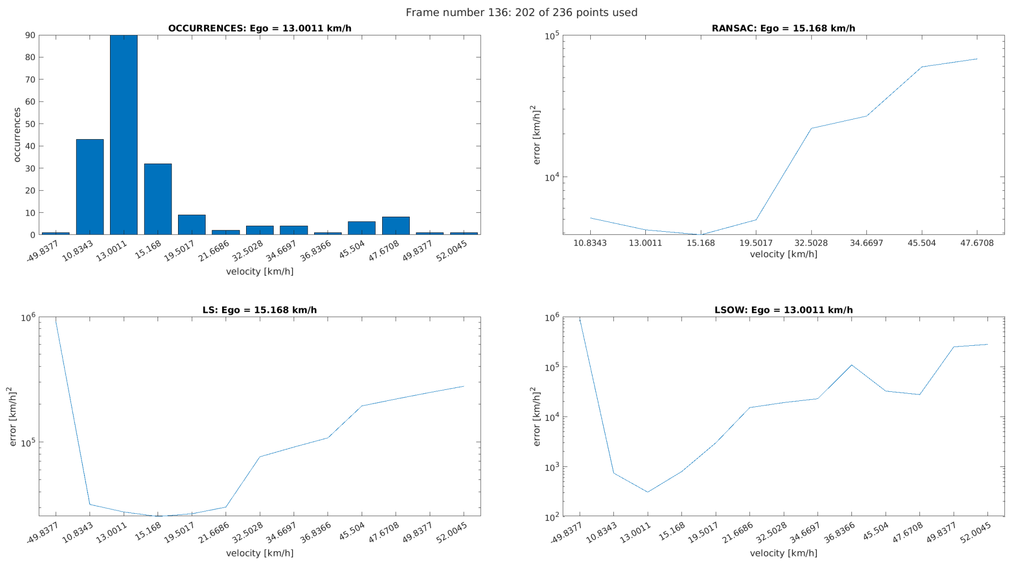


Figure 4.93: Algorithms graphical voting outcome during frame 136

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
-49.8377	1	-	5.9621	5.9621
10.8343	43	3.7083	4.5014	2.8679
13.0011	90	3.6245	4.4414	2.4872
15.168	32	3.5897	4.4054	2.9003
19.5017	9	3.6953	4.427	3.4728
21.6686	2	-	4.4803	4.1793
32.5028	4	4.3408	4.8816	4.2796
34.6697	4	4.4273	4.9591	4.3571
36.8366	1	-	5.0326	5.0326
45.504	6	4.7745	5.2875	4.5093
47.6708	8	4.831	5.3426	4.4395
49.8377	1	-	5.3949	5.3949
52.0045	1	-	5.4445	5.4445

Table 4.42: Algorithms numerical voting outcome during frame 136

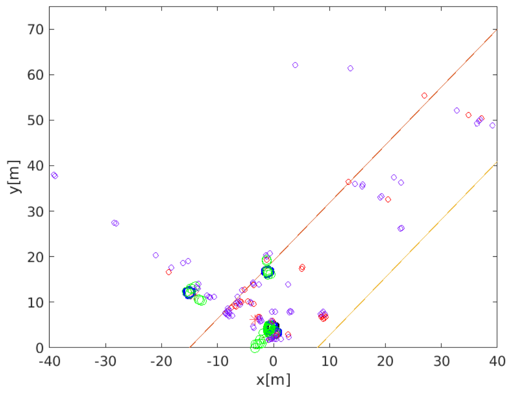


Figure 4.94: Input target cloud to the algorithms during frame 136

4.9.1.3 Frame 137

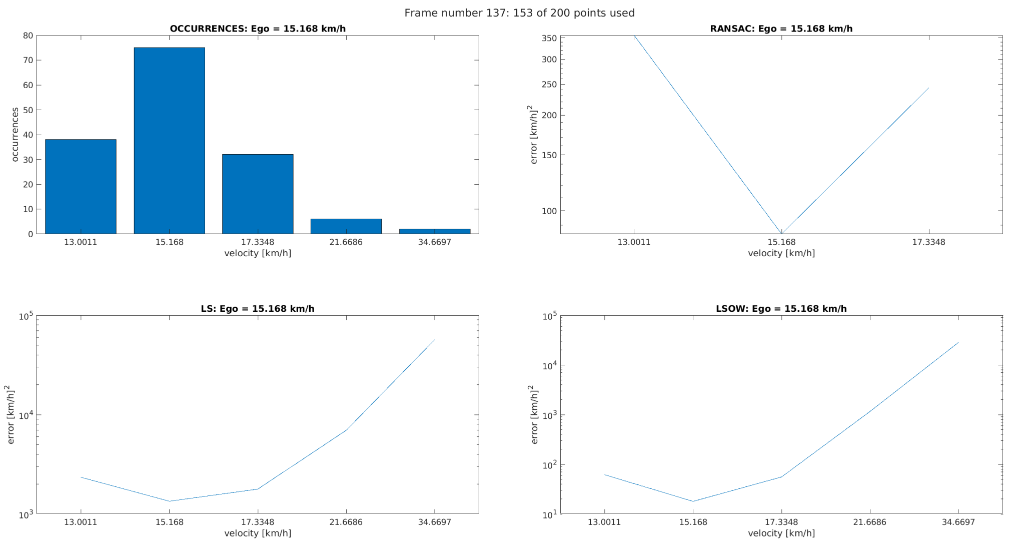


Figure 4.95: Algorithms graphical voting outcome during frame 137

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
13.0011	38	2.5525	3.3698	1.79
15.168	75	1.9269	3.128	1.253
17.3348	32	2.3877	3.2503	1.7451
21.6686	6	-	3.8428	3.0647
34.6697	2	-	4.7558	4.4548

Table 4.43: Algorithms numerical voting outcome during frame 137

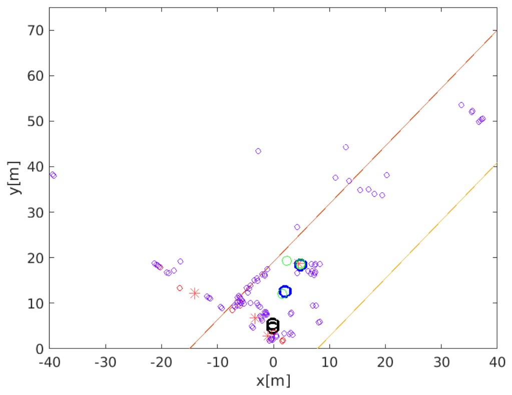


Figure 4.96: Input target cloud to the algorithms during frame 137

4.9.2 Dynamic host, opening target: Track until midway

The snapshot frames proposed in the figures 4.98, 4.100 and 4.102, present the track for the target which started in the "Dynamic host, opening target: Track Start" section. In this case it is presented when the targets has reached the midway between the position from which the track started and to which the track is released. For this acquired snapshot only the "*LS*" algorithm implementation is affected by an error during frame 165, as can be seen in figure 4.97 and its respective table 4.44. Beside that, all the algorithm implementations conform to estimating the same *Host* velocity.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended. The desired behavior is that the *Tracking* unit correctly keep the track on a moving *Target* when the *Host* is moving, but only tracks the ones which do not belong to the scenery. Thus the presence of a track only on a true *Target* demonstrates that it works properly.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity, where the impact of the *Target* is not negligible because its large velocity made the *Mean* velocity shift away from the *Mode*.

4.9.2.1 Frame 165

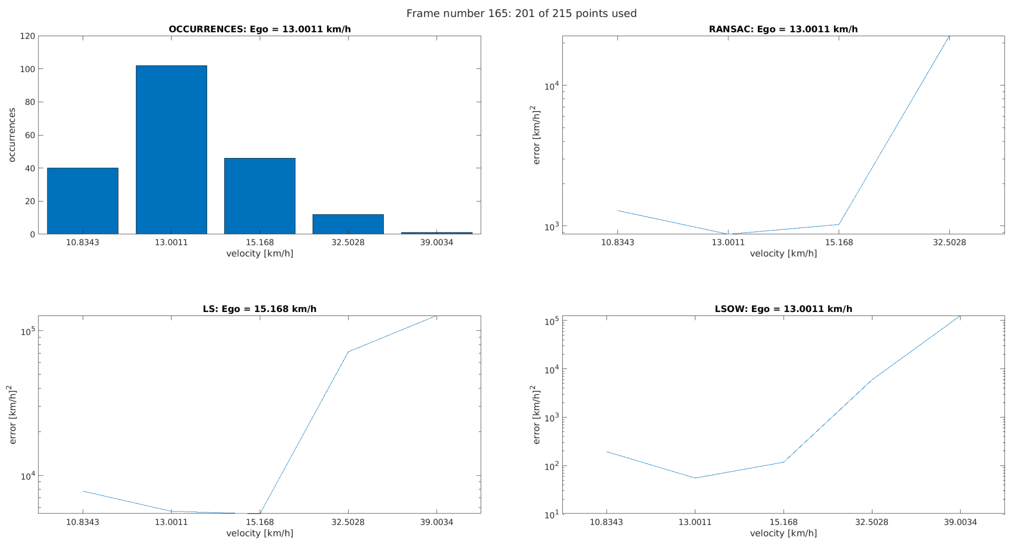


Figure 4.97: Algorithms graphical voting outcome during frame 165

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
10.8343	40	3.1094	3.8905	2.2884
13.0011	102	2.9412	3.7516	1.743
15.168	46	3.0101	3.7327	2.07
32.5028	12	4.3524	4.8539	3.7748
39.0034	1	-	5.105	5.105

Table 4.44: Algorithms numerical voting outcome during frame 165

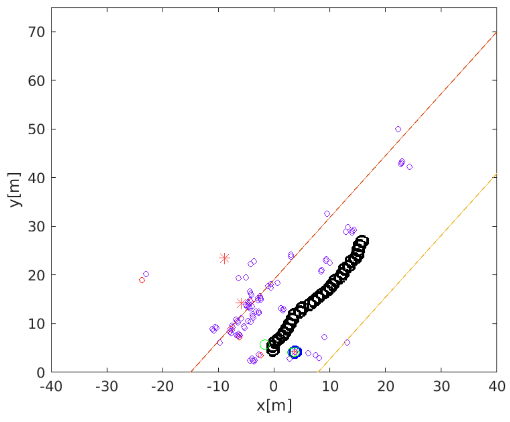


Figure 4.98: Input target cloud to the algorithms during frame 165

4.9.2.2 Frame 166

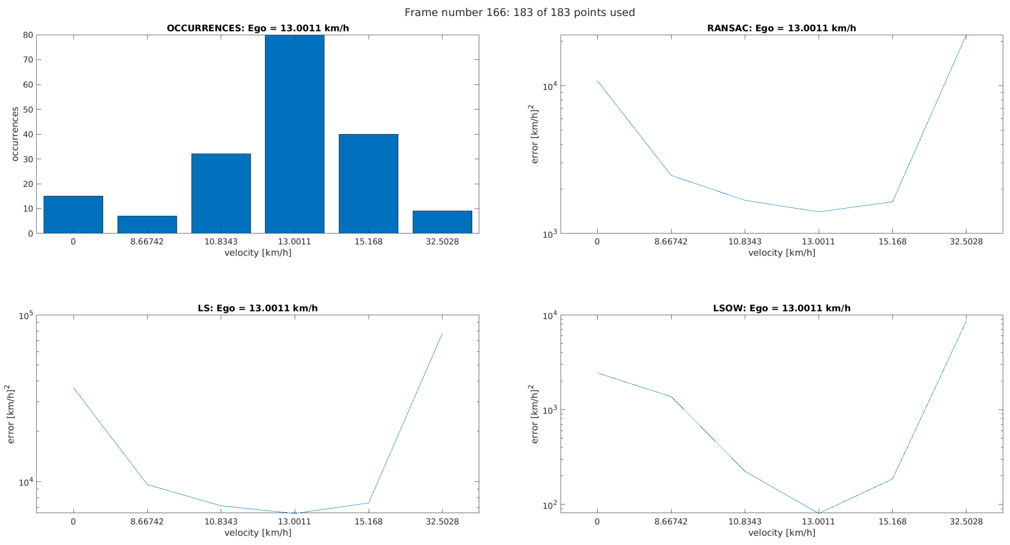


Figure 4.99: Algorithms graphical voting outcome during frame 166

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
0	15	4.0338	4.5625	3.3864
8.66742	7	3.3926	3.9815	3.1364
10.8343	32	3.2243	3.8541	2.3489
13.0011	80	3.1459	3.8081	1.905
15.168	40	3.2145	3.8709	2.2688
32.5028	9	4.3453	4.8881	3.9339

Table 4.45: Algorithms numerical voting outcome during frame 166

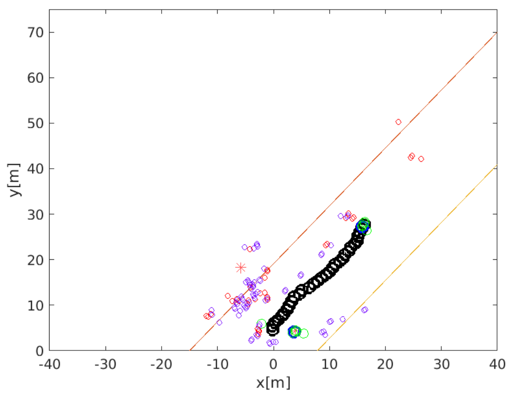


Figure 4.100: Input target cloud to the algorithms during frame 166

4.9.2.3 Frame 167

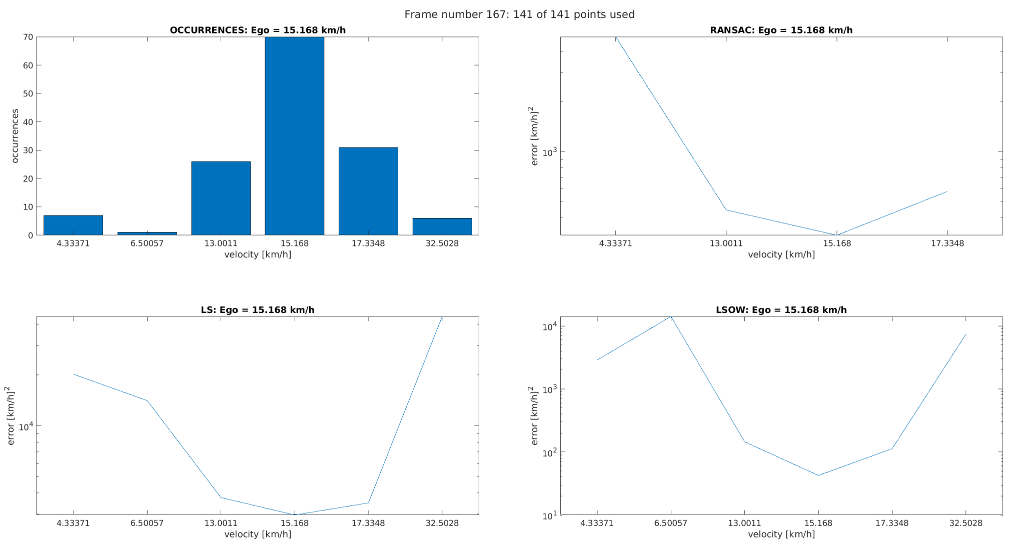


Figure 4.101: Algorithms graphical voting outcome during frame 167

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
4.33371	7	3.6916	4.3048	3.4597
6.50057	1	-	4.1488	4.1488
13.0011	26	2.6494	3.5753	2.1603
15.168	70	2.4977	3.4724	1.6273
17.3348	31	2.7616	3.5438	2.0525
32.5028	6	-	4.6463	3.8681

Table 4.46: Algorithms numerical voting outcome during frame 167

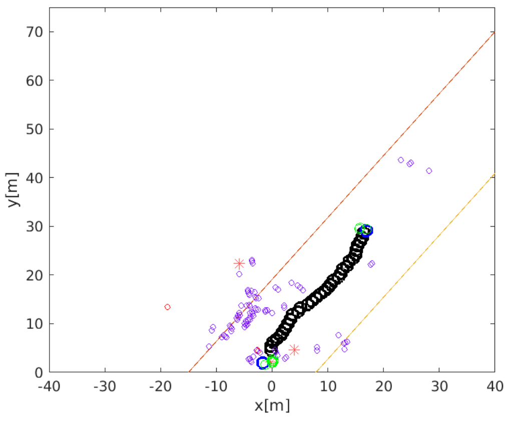


Figure 4.102: Input target cloud to the algorithms during frame 167

4.9.3 Dynamic host, opening target: Track End

The release of the target can be seen in the frames proposed in the figures 4.104, 4.106 and 4.108. The release occurs because the target has reached the point of interest, which this time was before the *Maximum Range* of the *Radar*, thus the *Track* resources can be released for later assignment to new tracks. In these frames all the implementations conform to estimating the *Host* velocity as $13\,\mathrm{km\,h^{-1}}$.

By observing this snapshot, the *Ego Estimation* algorithms allow for the *Tracking* algorithm to behave like intended when no *Ego Velocity* estimation error occurs. The desired behavior is that the *Tracking* unit correctly releases a track on a *Target* when the *Host* is moving. This situation occurs when the *Target* either exits the field of view or it stops, thus becoming a scenery point.

For what regards the *Ego Velocity* estimation error, the "*LS*" algorithm pointed to the velocity which was the closest to the targets distribution *Mean* velocity. The "*RANSAC*" implementation has subsequently chosen a subset of points from the cloud of targets whose *Mean* velocity was different than the *Mode*. Thus the "*RANSAC*" implementation voted the subset's point whose velocity was the closest to the subset's *Mean* velocity.

The proposed snapshot also shows a real issue that might affect the *Tracking* unit. As can be seen in frame 193, in figure 4.104, another track started while the *Target* was travelling to the destination. This track was caused by *multipath reflections* between the *Target* and the environment, thus resulting in a point which shows a different position and velocity than the *Target* that originated that. A positive note on that is that the *Track* only happened for a handful of frames and then released.

4.9.3.1 Frame 193

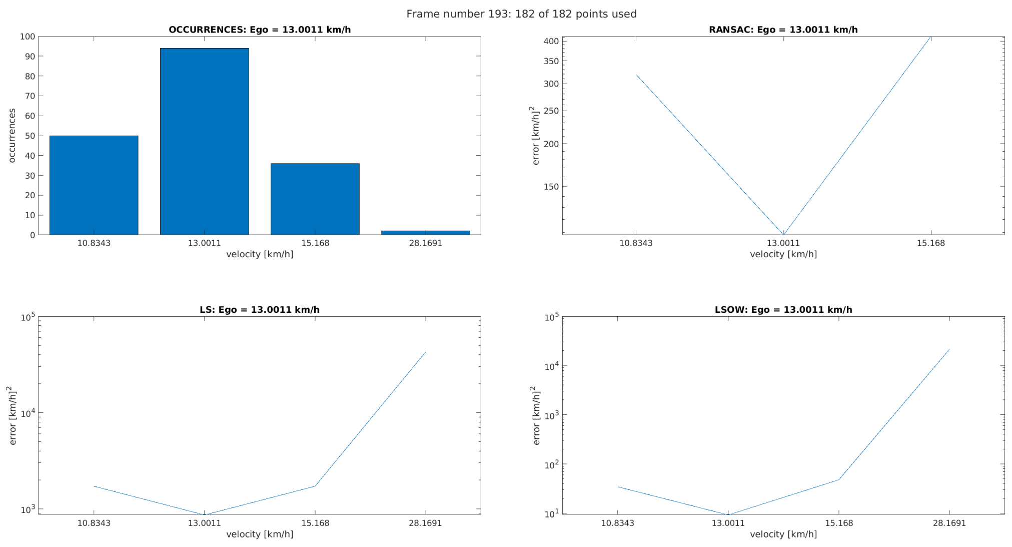


Figure 4.103: Algorithms graphical voting outcome during frame 193

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
10.8343	50	2.5042	3.2351	1.5362
13.0011	94	2.0334	2.9365	0.96335
15.168	36	2.6161	3.2351	1.6788
28.1691	2	-	4.6308	4.3298

Table 4.47: Algorithms numerical voting outcome during frame 193

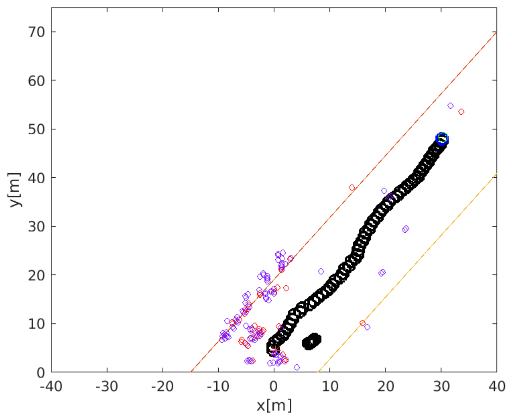


Figure 4.104: Input target cloud to the algorithms during frame 193

4.9.3.2 Frame 194

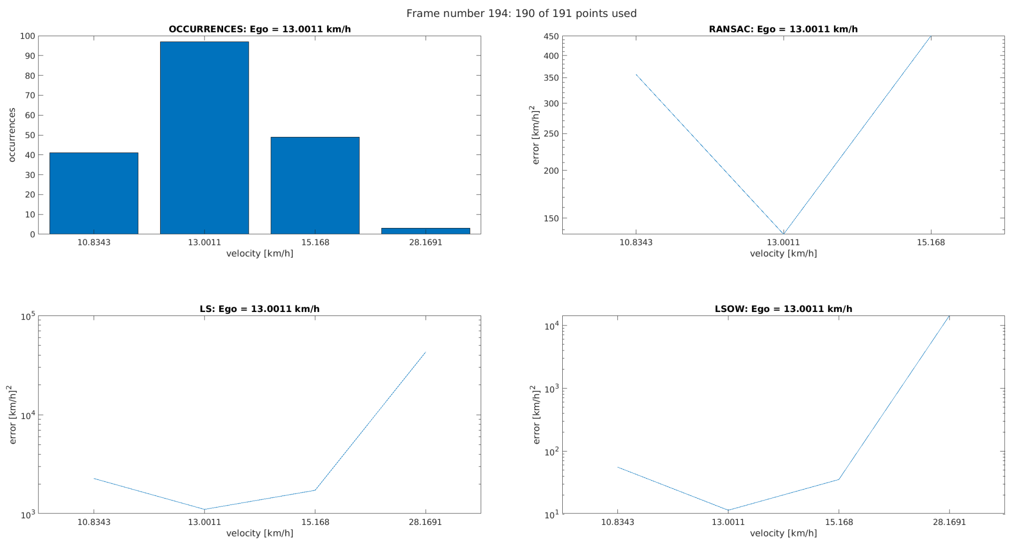


Figure 4.105: Algorithms graphical voting outcome during frame 194

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
10.8343	41	2.5525	3.3574	1.7446
13.0011	97	2.1341	3.0464	1.0596
15.168	49	2.6539	3.2387	1.5485
28.1691	3	-	4.6327	4.1555

Table 4.48: Algorithms numerical voting outcome during frame 194

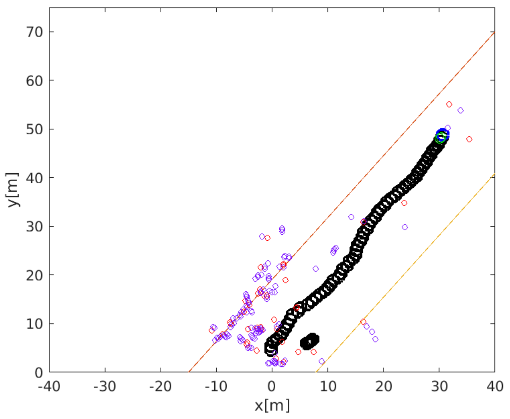


Figure 4.106: Input target cloud to the algorithms during frame 194

4.9.3.3 Frame 195

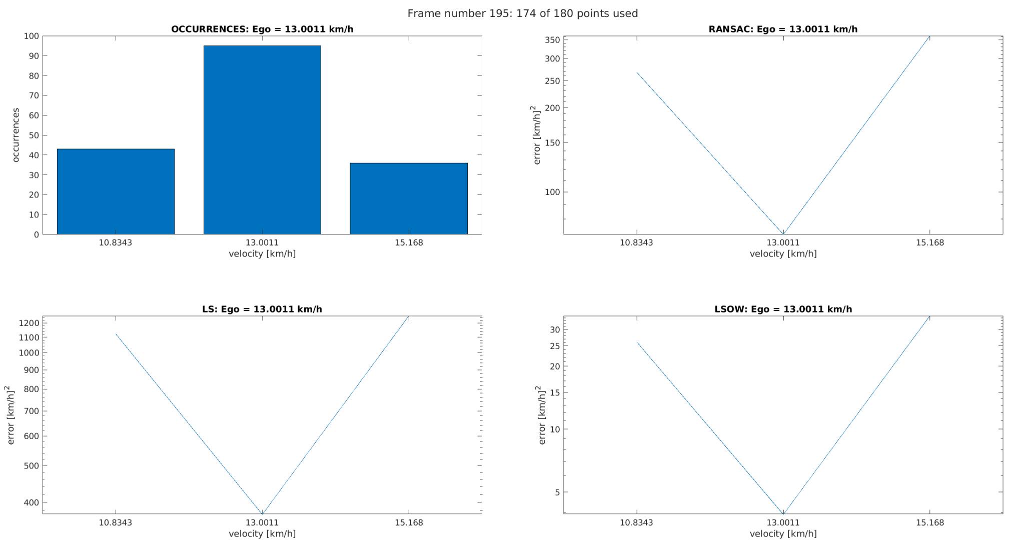


Figure 4.107: Algorithms graphical voting outcome during frame 195

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
10.8343	43	2.4275	3.0501	1.4166
13.0011	95	1.8478	2.5693	0.59156
15.168	36	2.5582	3.0982	1.5419

Table 4.49: Algorithms numerical voting outcome during frame 195

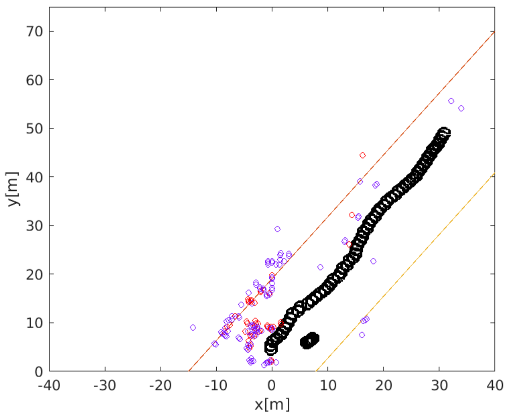


Figure 4.108: Input target cloud to the algorithms during frame 195

4.10 Further Considerations

Once the "Test Campaign" has been successfully performed, it can be observed that the "*LS*" and "*RANSAC*" implementations often have a divergent outcome with respect to the other two implementations. For what regards the other two, the "*_dop*" and "*LSOW*" implementations, they never fail to estimate a different *Ego Velocity* with each other.
This outcome derives from the fact that the targets never reach the amount of occurrences of the *Clutter* targets. If instead a large amount of opening targets are travelling at the same velocity, this situation could lead to a wrong *Ego Velocity* estimation from the "*_dop*" implementation.
In order to validate this statement, an alteration of the acquired data can be performed, since the test were not designed to include another set of moving targets.
In order to do so the frame 135, which was used for the "Dynamic host, opening target: Track Start" section, has been modified. A group of targets has been artificially introduced at $52.00\,\mathrm{km\,h^{-1}}$ in order to have the total occurrences for that group to be 56, thus being larger than the 55 occurrences of $13.00\,\mathrm{km\,h^{-1}}$. Furthermore, a velocity spread around those targets have also been introduced. After that, the *Radar* has been run with this artificially modified input set.
As can be seen in both figure 4.109 and table 4.50, now the "*_dop*" implementation now estimates the *Ego Velocity* with a wrong outcome.
The reason for the "*LSOW*" to correctly estimate the *Ego Velocity* is due to the nature of the algorithm itself that uses as estimator function the amount of error averaged by the number of occurrences. This particular estimator gives an information about a measurement point's dispersion with respect to the others.

4.10.1 Modified Frame 135

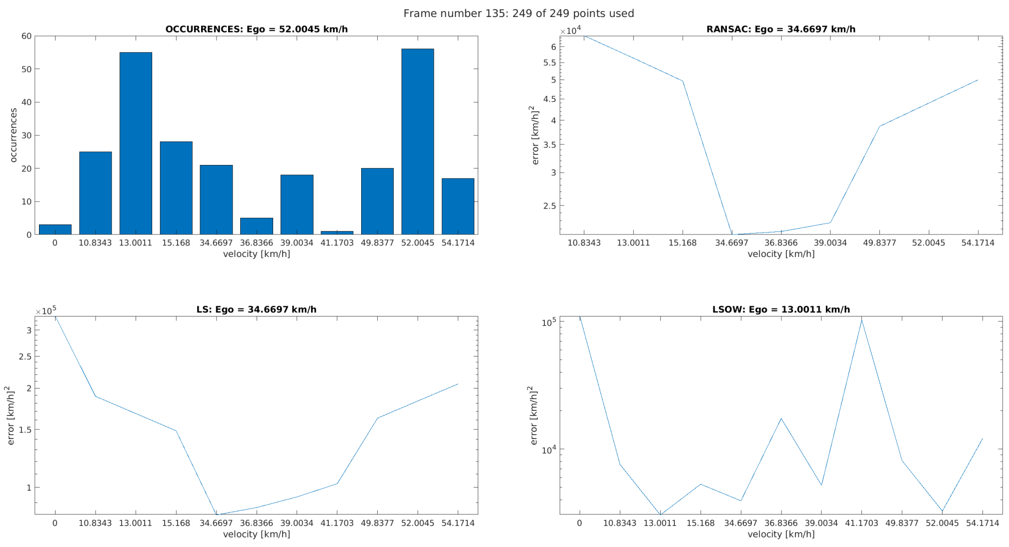


Figure 4.109: Algorithms graphical voting outcome during the modified frame 135

Velocity [km/h]	Occurrences	RANSAC Error (log)	LS Error (log)	LSOW Error (log)
0	3	-	5.5196	5.0424
10.8343	25	4.8042	5.2763	3.8783
13.0011	55	4.7508	5.2242	3.4838
15.168	28	4.696	5.1719	3.7247
34.6697	21	4.3285	4.9169	3.5947
36.8366	5	4.336	4.9392	4.2403
39.0034	18	4.357	4.9715	3.7162
41.1703	1	-	5.0115	5.0115
49.8377	20	4.5876	5.2101	3.9091
52.0045	56	4.6434	5.2623	3.5141
54.1714	17	4.699	5.3139	4.0834

Table 4.50: Algorithms numerical voting outcome during the modified frame 135

Chapter 5

Conclusions

The *Target Tracking* is one of the main functionality that a *Radar* performs during its regular operations, allowing for the detected targets to be observed and analyzed over time. This is done by collecting a given target observations in terms of the different physical dimensions that a *Radar* can measure. This collected data is then grouped into the most appropriate tracks. Tracks are software objects which identify a specific target, together with its measured information. After each collection, the tracks are updated with the new observations. The Tracking algorithm requires hardware resources during its execution, namely the CPU for the tracks' processing and the RAM memory in which the tracks are stored. In applications where *Radars* are mounted on a *Host* which is moving in an environment, all the measured points which belong to the latter move at the *Host* velocity. This condition greatly increases the demand of resources by the Tracking algorithm, which attempts to process all the moving targets for tracks to be processed. This situation leads to unnecessary hardware resources usage, which leads to *Timing* issues and ultimately in execution halt due to internal service misses. The *Ego Velocity Estimation* is proposed as a possible software algorithm which adds the capability to extract the *Host* velocity from a given *Radar*'s acquisition frame. This solution enables the possibility of using the same Tracking algorithm without the latter to be adapted for a moving *Host* scenario. Starting from this solution, a set of implementation have been discussed and presented in this thesis. Once a set of *Requirements* have been designed considering the final product's desired qualities, a first implementation have been developed, then tested and validated. Whenever the test outcome was not compliant to the requirements, the requirements are updated with the additional information gathered after the test. Thus the same process is repeated in an iterative way. The development process consist in finding a proper algorithm which can satisfy the given set of requirements and then implement it using the *C* programming language. The testing and validation process follows the development process and aims to verify that the behavior of the developed routine complies to the requirements. In this thesis the research of a suitable software routine for the *Ego Velocity Estimation* has led to four possible implementations. The routine "*egoVelocityEstim_dop*" is the one which searches the *Mode* of the targets distribution by logging the occurrences of each target velocity. Its test proved to be reliable in application scenarios where the *Host* is moving in an environments with an amount of targets' *Outliers* that do not exceed the scenery's *Inliers*. The routine "*egoVelocityEstimRANSAC*" is the one which computes the closest velocity to the *Mean* velocity of a subset of targets which are randomly extracted from the acquired set. This routine implements the solution proposed in [1]. Its test proved this routine to be mildly reliable in the same application scenario of "*egoVelocityEstim_dop*", where the unreliability comes from the *random number generator* and the statistics of the targets in a given acquired frame. This mild unreliability makes the *Radar* misinterpret which targets are really moving, thus introducing tracking errors and even *Real-Time* service missing. Thus this routine is not suitable for this particular *Radar* platform on which the CPU and memory resources are already limited by the complexity of its acquisition schema. So, the limited resources also constrains the allowed *Ego Velocity* estimation errors. The routine "*egoVelocityEstimLS*" is the one which computes the closest velocity to the *Mean* velocity of the acquired set. It is the extreme case of the "*egoVelocityEstimRANSAC*", where a single *RANSAC* attempt is performed using all the acquired targets. Its test proved this routine to be mostly unreliable in the same application scenario of "*egoVelocityEstim_dop*", where the unreliability comes from the fact that *Mode* and *Mean* overlap only for specific distributions. Thus this routine is not suitable for the moving *Host* application scenario. The routine "*egoVelocityEstimLSOW*" is the one which computes the velocity which has the least dispersion across the *Field of View*. Its test proved this routine to be reliable in the same application scenario of "*egoVelocityEstim_dop*" and outclass the latter in cases where the amount of targets are larger than the *Clutter* occurrences. This routine achieves this result due to the *Clutter* statistics, which gathers around a *Doppler* bin and is spread on all the *Range* bins. The proposed implementations are just some possible ways to implement the *Ego Velocity Estimation*, which attempts to converge to a lightweight software solution. A different approach could be the adoption of *Image Processing* and *Machine Learning* algorithms that can extract the required information using a high level approach towards the acquired data. These algorithms demands more resources than the one provided by the *AWR1843* chip, which is adopted by the *Radar* platform used for the *Testing and Validation* of the routine.

Bibliography

[1] Michael Barjenbruch Dominik Kellner, Jürgen Dickmann Jens Klappstein, and Klaus Dietmayer. «Instantaneous ego-motion estimation using Doppler radar». In: *"16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)"*. 2013, pp. 870–872. DOI: `10.1109/ITSC.2013.6728341` (cit. on pp. iii, 35, 42, 127).

[2] Merrill I. Skolnik. *Introduction to radar systems*. New York (State): McGraw-Hill, 1962 (cit. on pp. 1–5, 11, 16).

[3] «"IEEE Recommended Practice for Radar Cross-Section Test Procedures"». In: *IEEE Std 1502-2007* (2007), pp. 1–70. DOI: `10.1109/IEEESTD.2007.4301372` (cit. on p. 5).

[4] L. Nicolaescu and T. Oroian. «Radar cross section». In: *"5th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service. TELSIKS 2001. Proceedings of Papers (Cat. No.01EX517)"*. Vol. 1. 2001, 65–68 vol.1. DOI: `10.1109/TELSKS.2001.954850` (cit. on p. 6).

[5] Cesar Iovescu and Sandeep Rao. *The fundamentals of millimeter wave radar sensors*. Tech. rep. SPYY005A. MA: Texas Instruments: Texas Instruments, July 2020 (cit. on pp. 16, 18, 19).

[6] Sandeep Rao. *MIMO Radar*. Tech. rep. SWRA554A. MA: Texas Instruments: Texas Instruments, May 2017 (cit. on pp. 20–22).

[7] A. Moffet. «Minimum-redundancy linear arrays». In: *IEEE Transactions on Antennas and Propagation* 16.2 (1968), pp. 172–175. DOI: `10.1109/TAP.1968.1139138` (cit. on p. 22).

[8] Chun-Yang Chen and P. P. Vaidyanathan. «Minimum redundancy MIMO radars». In: *"2008 IEEE International Symposium on Circuits and Systems (ISCAS)"*. 2008, pp. 45–48. DOI: `10.1109/ISCAS.2008.4541350` (cit. on p. 23).

[9] Robert C. Bolles Martin A. Fischler. «"Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography"». In: *Communications of the ACM.* Vol. 24. 6. 1981, pp. 382–384. DOI: `10.1145/358669.358692` (cit. on pp. 24, 35).

[10] Msm Wikimedia Commons. *Fitted line with RANSAC; outliers have no influence on the result.* URL: `https://commons.wikimedia.org/w/index.php?curid=2071406` (cit. on p. 24).

[11] *AWR1843 Single-Chip 77- to 79-GHz FMCW Radar Sensor.* Dallas, Texas: Texas Instruments (cit. on pp. 26–29).

[12] *AWR18xx/16xx/14xx/68xx Technical Reference Manual.* Dallas, Texas: Texas Instruments (cit. on p. 30).