POLITECNICO DI TORINO

Dipartimento di Ingegneria Meccanica ed Aerospaziale

Tesi di Laurea Magistrale

Development of a localization system based on ArUco markers for a small space platforms test bench

Relatori

Prof. Sabrina Corpino

Ing. Fabrizio Stesina

Candidato

Luca Patrioli

Matr. 246972

Aprile 2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

AC	Alternating Current
AKE	Absolute Knowledge Error
ArUco	Augmented Reality Universidad de Córdoba
CCC	CAST Control Centre
CPU	Central Processing Unit
CV	Computer Vision
DOF	Degrees of Freedom
FPS	Frames Per Second
FOV	Field of View
FTS	Frictionless Table System
GNC	Guidance, Navigation and Control
GSE	Ground Support Equipment
HIL	Hardware-In-the-Loop
I/O	Input/Output
LocSys	Localization System
MBS	Main Board System
MEMS	Micro Electro-Mechanical Systems
MKE	Mean Knowledge Error
MPS	Measures per second
MRS	Master Reference System
NPS	Naval Post graduate School
OBC	On-Board Computer
PnP	Perspective-n-Point
RKE	Relative Knowledge Error
RS	Reference System
SS	Solar Simulator
TO	Test Object
UWB	Ultrawide Band
WPS	Wi-Fi Positioning System

ABSTRACT

CubeSats are an increasingly common reality, both in the academic and industrial fields, thanks to their possibility of carrying out a wide variety of missions, with a “low cost” and “fast delivery” approach. The verification and validation phase usually take place after the assembly phase or on the individual subsystems, losing the possible interactions between them. Therefore, arises the need of a platform capable of reproducing simultaneously as many aspects as possible of the CubeSat operating environment and capable of simulating the undeveloped subsystems through a virtual model. This facility would allow to support the development of the satellite throughout its life cycle, further reducing time and cost.

CAST (CubeSat Advanced Simulator and Testbench), a project born within the CubeSat PoliTo Team, aims to improve and boost the verification process of a CubeSat, providing an integrated environment. Thanks to a modular architecture and the in-the-loop verification approach, CAST is composed of four main elements: 1) a Control Centre, to simulate the desired modules; 2) a Main Board System, to manage other ground support equipment and interfacing to the test object; 3) a Sun Simulator, to emulate the sun radiation in different scenarios; 4) a Frictionless Table System, to simulate proximity manoeuvres, thanks to a support structure, the TowerSat.

This thesis proposes the design of a localization system for the TowerSat, the LocSys. The TowerSat is free to move in a limited environment, but it is necessary to monitor its position and orientation, both for safety reasons and, possibly, to validate the results elaborated from the test object. LocSys adopts optical cameras, as they do not interfere in any way with the test object. To increase the efficiency and the accuracy of the system, some binary fiducial markers have been adopted, called ArUco markers.

Two software have been developed adopting Python as programming language: the first implements an architecture for a single camera, in order to test the performance of a such kind of system; the second software implements a multi-camera architecture, to cover a wider area, overcoming the limits of the field of view of a single camera and increasing the LocSys accuracy. Each software implements the possibility to calibrate the optical cameras, through a properly developed calibration tool. The calibration phase is the most crucial, greatly influencing the results.

Finally, some tests were carried out to validate the system, obtaining an average error less than 1.0 [cm] for the position and less than 1.0 [deg] for the orientation.

1 INTRODUCTION

Testing a system level is one of the major expensive steps in a product life cycle, especially for those systems which incorporate embedded computing, like the space ones. In fact, increasing the levels of complexity in system hardware and software, makes the verification process more severe. Additionally, any significant changes made to an existing hardware or software products involve a regression in testing the system. Clearly, the need of accelerating and automating the system level tests is becoming increasingly evident. [1]

Nowadays, several techniques have been developed for this mandatory phase in a life cycle of a product. These techniques aim to increase the validity of the test results, trying to best simulate the final operating environment in which the system will live, and interfering as little as possible with it.

One effective modelling and simulation method is the Hardware-In-the-Loop (HIL) approach. This methodology consists in the combination of both computer simulation and hardware in a single platform. It is a hybrid architecture that simulates both software and hardware, in which the hardware part can vary from a few components to the fully integrated system. HIL simulation requires the development of a real-time simulation that models some parts of the embedded system under test and all significant interactions with its operational environment. [1] The outputs of the test object (TO) are used as inputs to the simulation, in turn the simulation generates outputs that became inputs to the embedded system. An example of this kind of facility is shown in Figure 1.

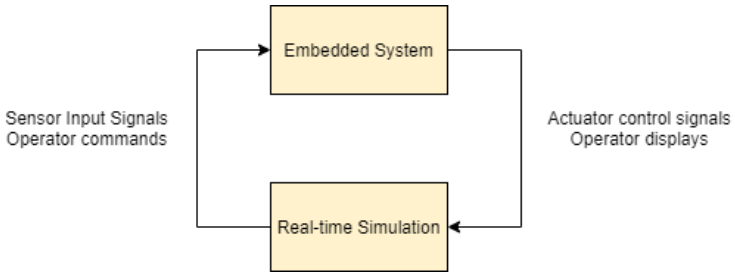


Figure 1: HIL simulation example

HIL technique is particularly useful for the verification of those systems that operate in special environments and conditions which are difficult to reproduce in a laboratory, such as the satellites in the operational orbit environment.

The art state of simulator based on HIL technology present a wide range of options, depending on the complexity level. An interesting example is the simulator of on-orbit docking between two spacecraft at the Spacecraft Robotic Laboratory of the Naval Post Graduate School (NPS) in Monterey, CA.[2] In this facility, the target and the chaser are physically reproduced, while the rest of the system is simulated in a real-time by a dedicated computer. This example perfectly shows an HIL simulation where the real hardware is just one subsystem of the whole spacecraft.

Figure 2: Three generations of spacecraft simulator at the NPS ©[2]

The reliability and validity of the test results are closely linked to the fidelity with which the operating environment is simulated and to the quantity of systems involved. Furthermore, the verification process should be performed at any moment during the entire product life cycle, to verify the feasibility, the capabilities and the performances of a system. The Model Based System Engineering (MBSE) fits exactly in this context. The verification phase is extended to the entire duration of the project through simulation sessions, where different models are used to emulate the behaviour of the systems or the environment, depending on the life cycle phase.

During the development of the project, the virtual models are replaced with the constructed hardware parts. Therefore, the need to relate the physical components to the models arises. For traditional spacecraft, tailor-made facilities are developed for the verification and validation phase. Moreover, due to the great diversity of the spacecrafts, these test benches cannot be reused without substantial changes, increasing the costs and design times of the satellite itself.

CubeSats, on the other hand, are small standardized satellites, therefore the platforms to test them could in turn be standardized, adopting a modular HIL approach. CAST was conceived in this context and the following thesis concerns one of the subsystem necessary for the monitoring and verification of the test bench.

2 THE CAST PROJECT

CAST, CubeSat Advanced Simulator and Test bench, is a project born inside the CubeSat Team of Politecnico of Torino.

The main objective of CAST is to have an integrated environment to support the development and verification of CubeSats along their life cycle, from preliminary design to operations.

CubeSats are small spacecrafts that follow a well-defined standard, the CubeSat Design Specification [3], created by California Polytechnic State University in 1999. This standardization allows a "low cost" and "fast delivery" approach, and it keeps increasing the interest for this specific platform by both universities and industries.

The "low cost" concept is due to the large quantity and variety of off-the-shelf components (COTS) which are being developed since their creation.

Traditional spacecraft developments require a complex and expensive process starting from the concept of operations to the delivery and operations phases, that might go over decades. The CubeSat low-cost and fast-delivery paradigm requisites a smarter approach for the spacecraft development, while maintaining the adequate level of reliability and safety, with the purpose of reducing time and costs and keeping the quality of the product.

For these reasons, the core concept of CAST is to develop a facility that adopts a multiV-model, compared to the more classic and often still used V-model (see Figure 3), so that the verification activities can be performed during all the project life cycle.

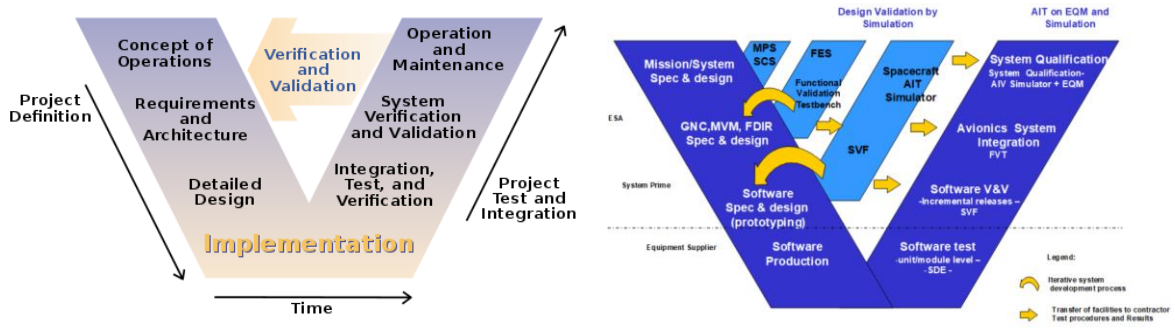


Figure 3: V-model and multiV-model comparison ©ESA

To achieve these goals CAST shall adapt his architecture to the phase and level of verification. Therefore, the simulation in-the-loop approach is implemented using

different virtual model to emulate the behaviour of a component, an equipment or an entire system, that it has not been developed yet. Hence, the CAST design is obtained considering some key concepts:

- **Reliability**: it shall have the ability to avoid and/or manage failures, that could compromise the integrity of the spacecraft system under testing
- **Autonomy**: it shall have the ability to manage expected events, reducing the operator interventions and allowing the use by non-expert operators
- **Connectivity**: it shall have the capability to connect/host elements with different types of interfaces, for both ground support equipment (GSE) or test object
- **Flexibility**: it shall have the capability to adapt his configuration depending on type of verification, thanks to its modular architecture
- **Reusability**: it shall have the capability to be adopted in different verification phases of the same project and between different projects

From these drivers, it was possible to define the high-level requirements of the CAST project, as shown in Table 1.

Table 1: CAST High-Level Requirements

High-Level Requirements	
ID	**Description**
HL10	CAST shall be an integrated environment
HL20	CAST shall support the development of small platforms
HL30	CAST shall support the verification of small platforms
HL40	CAST shall be used in different phase during the product life cycle
HL50	CAST elements shall be connected through logical and physical interfaces
HL60	CAST elements shall be adoptable between different projects
HL70	CAST shall cost less than 100K €
HL80	CubeSat Team PoliTo shall be the main user of CAST
HL90	Project shall be completed in the fourth quarter of 2020
HL100	CAST shall be defined by a modular design
HL110	Safety shall be considered TBD in relation to operators
HL120	Safety shall be considered TBD in relation to the physical system
HL130	Reliability shall be expressed according to redundancies and cryptic functions
HL140	CAST shall continue the simulation, despite a failure, without performance drops
HL150	CAST shall continue the simulation, despite two failures, with performance drops

HL160	CAST shall stop the simulation when the third failure occurs; human control shall be required
HL170	CAST shall maximize number of interfaces
HL180	Autonomy shall be defined by the simulator, during the development, by rapid prototyping and by auto generation of the code
HL190	Autonomy shall be defined during verification activities by reducing intervention of Operators
HL200	CAST shall be used also by not-experts

2.1 CAST MAIN ELEMENTS

The modular architecture and flexibility concepts concern the availability to implement new ground support equipment, without major changes to the entire facility. This design philosophy allows CAST to be extended and improved, trying to simulate, more and more accurately, the operative environment of the satellite.

Currently, CAST shall be composed by four main elements, connected as shown in Figure 4:

- CAST Control Centre (CCC)
- Main Board System (MBS)
- Solar Simulator (SS)
- Frictionless Table System (FTS)

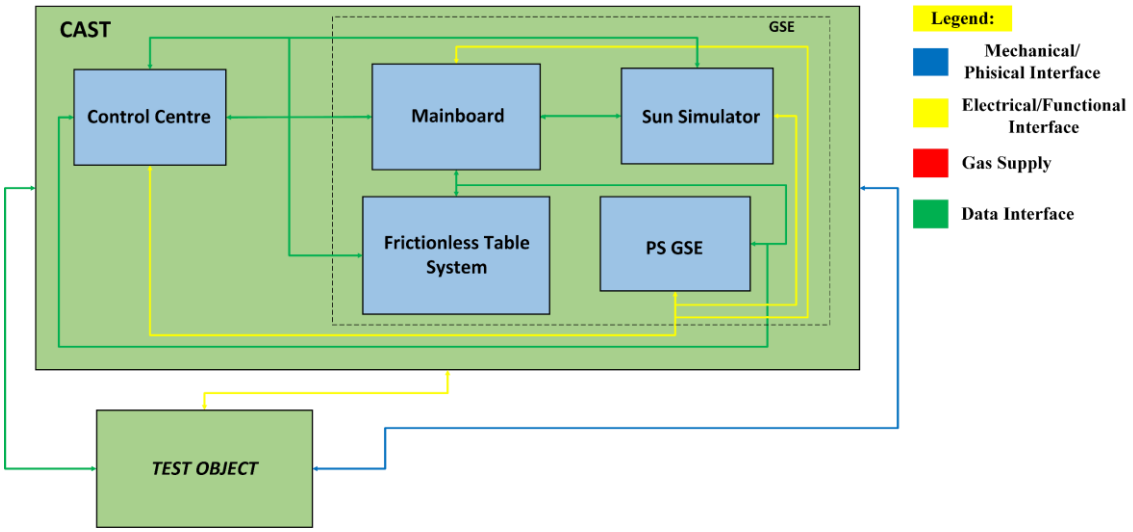


Figure 4: CAST functional architecture

PS GSE is the Power Supply Ground Support Equipment to power up the other CAST elements.

The CCC aims at managing the entire facility during its operational phases, coordinating the verification and simulation processes. It is the interface between the facility and a non-expert operator. Through the Control Centre, it shall be available to select the virtual models that simulate the missing hardware, according to the HIL approach. This functionality is the key one in order to be able to use CAST at any stage of the product life cycle. In fact, different levels of virtual model details can be implemented, with respect to project phases and type of missions.

The MBS is what make CAST an innovative system. This ground support equipment represents the link between who manage the correct functioning of the system (CCC), the other GSE and the test object. It shall be composed of different interfaces types, in furtherance of supporting a large variety of TO, without having to modify the facility. The system communicates with all the CAST elements, and in case of anomalous operations that were not managed by the CCC, implements some safety measures. Therefore, it should represent a major security tool.

The Solar Simulator, as the name suggests, has the purpose to simulate the radiation emitted by the Sun, in different flight conditions. It shall vary the light intensity by acting directly on the power emitted by the lamp, or by moving relative to the test object. Through this GSE it shall be possible to test an entire Electric Power System (EPS) of a spacecraft, or only the solar panels, according to the design phase and the other elements involved in the verification process.

The last main element of CAST will be described more in detail in the next section, considering it is closely linked to the topic of this thesis.

2.2 FRICTIONLESS TABLE SYSTEM

The Frictionless Table System shall allow the simulation of proximity manoeuvres, such as rendezvous and docking, performing movements with 3 degrees of freedom (DOF). The movements allowed are two translational on the table and one rotational around the perpendicular direction of the table surface. To test this type of in-orbit operations in a laboratory, it is necessary to reproduce the absence of friction in space. For these reasons, the system shall implement a technology already adopted for this kind of verification in the space field, namely planar air bearing.

The concept is to reproduce the absence of friction by injecting a very high-pressure gas between the main structure of the system and a specially designed surface (the table). The structure can then float over the table, minimizing the friction. Indeed, the friction is minimized, but is not eliminated. The pressurized gas can interact with the micro roughness of the supporting surface, for the simple principle of action and reaction (Newton's third law of motion). Hence, these interactions create disturbances that shall be compensated, both translationally and rotationally. It shall be autonomous both in terms of energy and management, implementing an onboard computer for decision making and appropriate corrections evaluation.

The FTS shall therefore perform many functions just for its self-operation., described in the subsystem level functional tree shown in Figure 5.

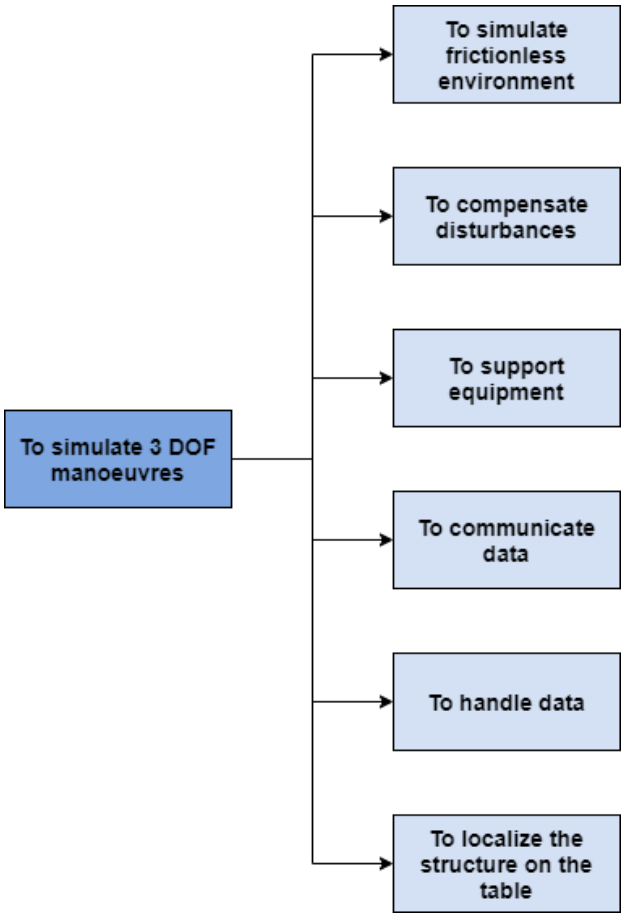


Figure 5: FTS Functional Tree

2.2.1 FTS Main Systems

Therefore, the FTS is composed by two main elements: the table and the TowerSat. The table shall be made of a material with the lowest roughness level, e.g. epoxy resin or granite, to minimize the interactions. An excellent example of such a surface can be found at the Spacecraft Robotics Laboratory of the Naval Postgraduate School. [4]

TowerSat is the operational structure, which implements all the subsystems necessary to create the environment without friction. Its design is based on some key concepts, which reflect those of CAST. Some of these concepts are:

- **Modularity**: capability to add or remove functionalities to the system, without having to design a new structure
- **Accessibility**: capability to facilitate maintenance (like the tank refuelling) or implementation of the test object, without having to disassemble large parts of the structure
- **Flexibility**: capability to adapt to different test objects in different phases of the project, without major changes
- **Simplicity**: capability to have a simple approach even for a non-expert operator

The Figure 6 shows a preliminary design of the TowerSat with the main elements of which is composed. The modularity key driver led to the division of the structure into floors.

In the first floor shall be the heavier components, to lower the centre of mass of the system and increase its stability, compared to the disturbances. The green elements at the bottom are the air bearing pads, which generate the thin layer of compressed gas with the table. The quantity and pressure of the gas emitted depends on the force that the pads must generate to support the weight of the structure. It is perceived the need to have a light system.

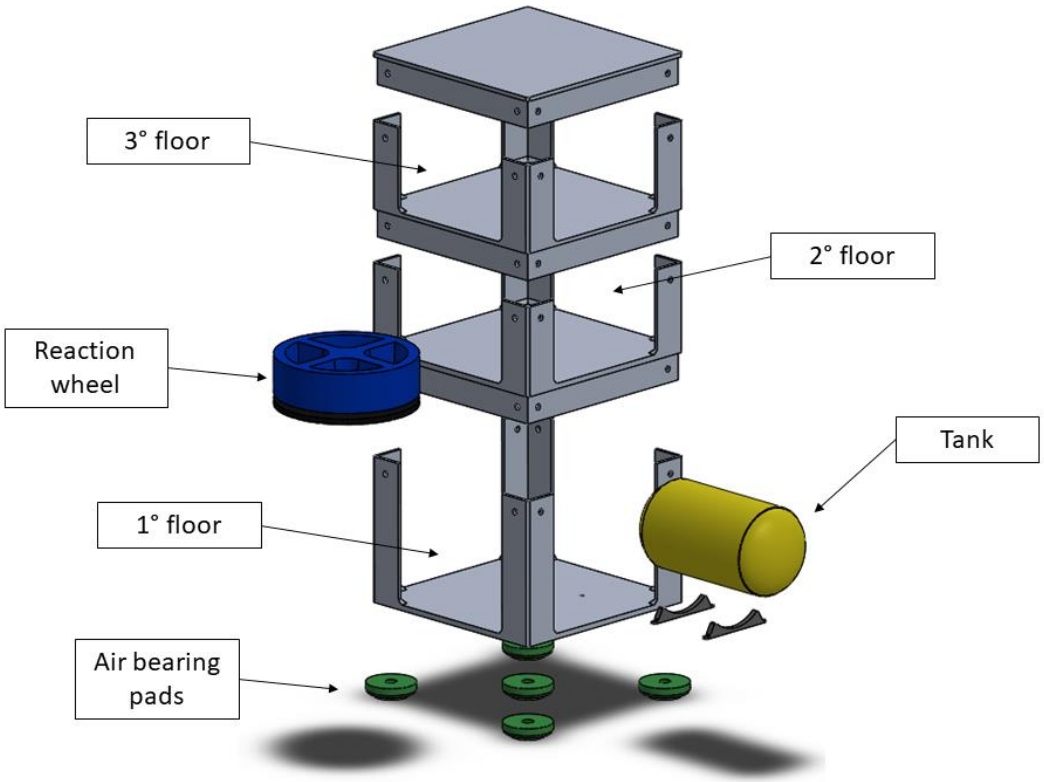


Figure 6: FTS exploded view

The pads are connected to a compressed gas tank, the yellow element, which is the heaviest element of the system and for this reason it influences the size and position of the other subsystems. The tank will need to be filled, that is why it must be easily accessible. These components allow the TowerSat to float on the table.

In addition, eight thrusters, two per side, are located on this floor, as they must be powered by compressed gas, reducing the weight of the connection pipes as much as possible. The thrusters are intended to compensate for translational disturbances and to generate the force to move the TowerSat on the table. The Figure 7 shows a possible functional configuration of the first floor of the TowerSat.

Concerning the other floors, they are still in development. The best configuration so far hypothesized sees the remaining equipment in the second floor and the test object, with its supporting interfaces, on the third floor (see Figure 8).

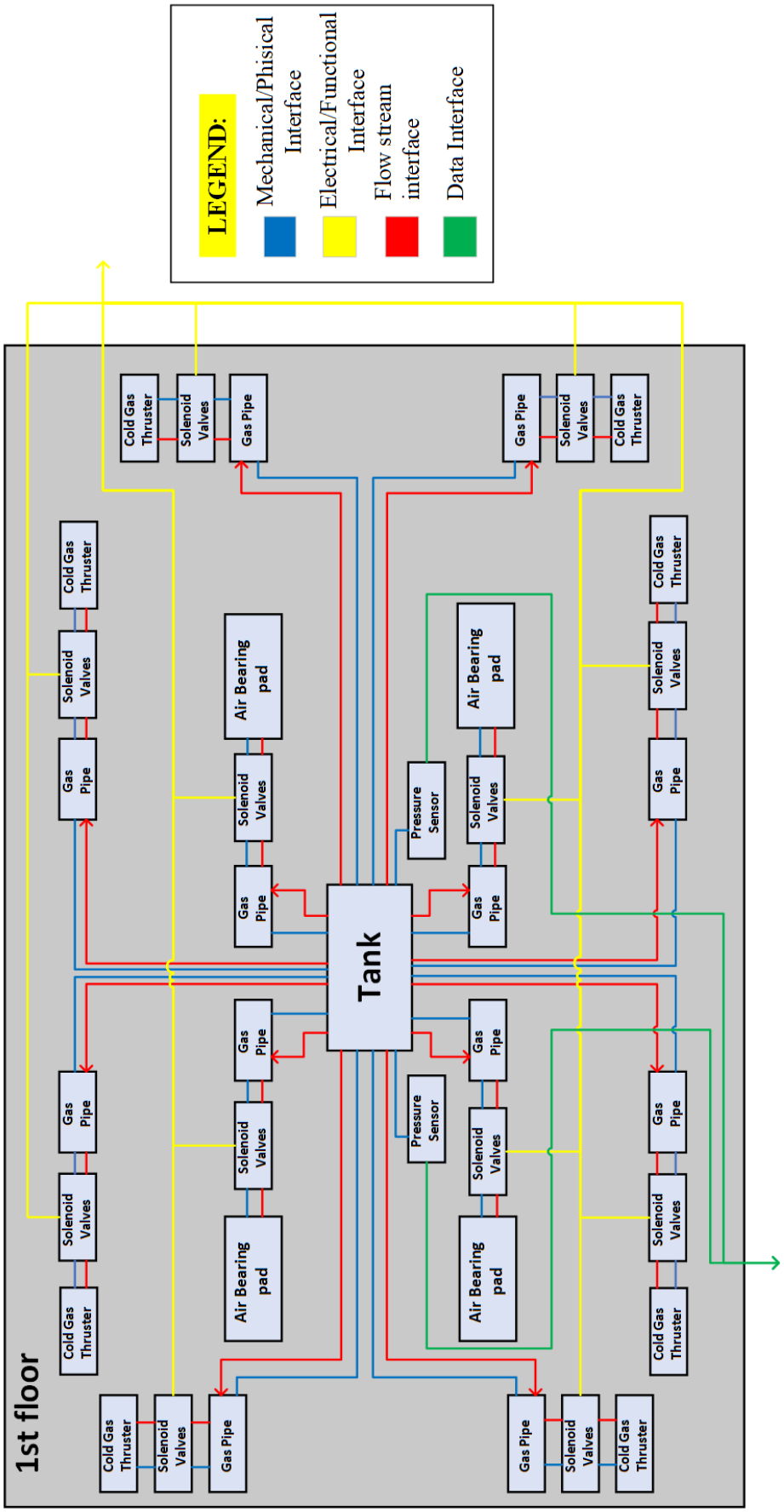


Figure 7: TowerSat 1st floor

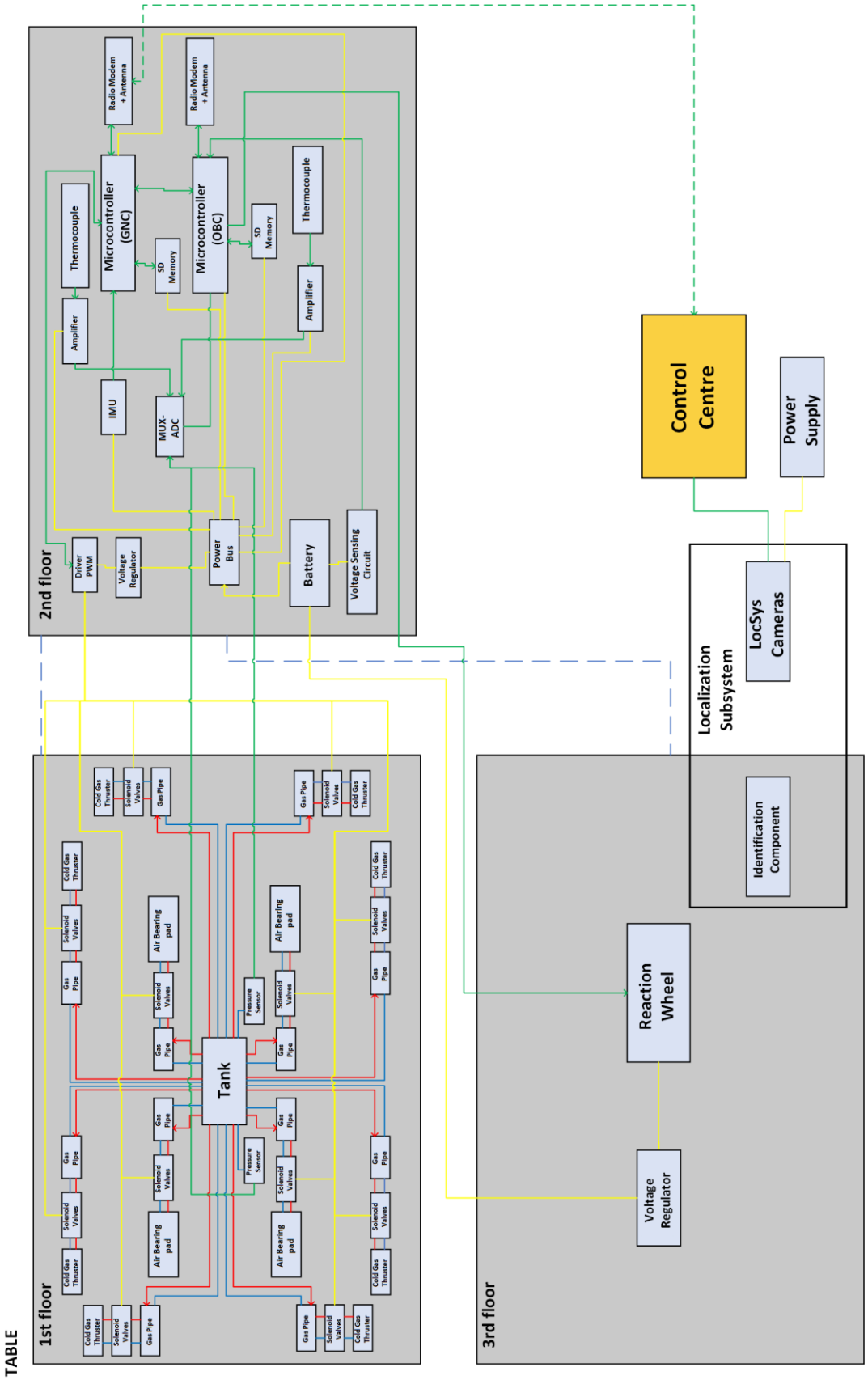


Figure 8: TowerSat functional architecture

In the second floor shall be a reaction wheel (the blue element in Figure 6) to perform corrections due to rotational disturbances and to intentionally rotate the structure. The size of the wheel depends on the total inertia of the TowerSat, so once again the minimization of dimensions and weights influences the components themselves, increasing the complexity of the project.

Together with the reaction wheel, the electronic equipment should be mounted on the side panels of the structure, always considering the accessibility desired by the system. The electronic equipment consists of two calculation units, the GNC and the OBC, analogous to a spacecraft system. The GNC includes an Inertial Measurement Unit (IMU) and a microprocessor, to process the corrections and movements to be made with the thrusters and the reaction wheel. Instead the OBC provides the necessary autonomy to the system, reducing the communication to the CAST Control Centre. It checks the correct functioning of the system, monitoring the pressures and temperatures, provided by the appropriate sensors. It also receives commands from the CCC and sends all the required data.

To add functionality to the system, the design shall allow the implementation of any other floor. In any case, the latter floor should be dedicated to implementing a test object. Therefore, it must have several mechanical and electrical supports. For this last reason, one of CAST goals is to implement the Main Board System on the TowerSat, making it an increasingly integrated environment.

Finally, the FTS needs a localization system, the LocSys. This system aims to monitor the movements of the TowerSat, evaluating its position and orientation over the time. It is the thesis topic and it will be introduced in the next section.

In conclusion, such a defined facility allows a great variety of test objects. Here are some examples, but it must be remembered that the possibility of expanding its functionality would allow other type of tests:

- Docking mechanism (main function)
- Solar panels: simulating the movement with respect to the Solar Simulator
- Optical payload for navigation
- Navigation algorithms, using the LocSys for validation
- CubeSat: whose movements are simulated by the TowerSat

2.2.2 FTS Localization System

The localization system aims to determine the position of the TowerSat with respect to a reference system, like a corner of the table. This system has a dual purpose:

- to check that the TowerSat does not hit the edges of the table, to not interfere with the test in progress, damaging or degrading it
- to check and, when required, validate the position and orientation that the test object is evaluating during the test

Especially for the second objective, the system should have high performances and must be able to cover the whole interested area of the table. Depending on the technology chosen for localization, its implementation in the FTS may vary.

The indoor positioning systems, like the LocSys should be, use different technologies, which can be grouped into two categories:

- Non-radio technologies
- Wireless technologies

The non-radio technologies can be used for positioning and can provide high accuracy with expensive equipment and installations. This category includes magnetic systems, inertial measurements and vision-based technologies (with both visual markers and features).

For the LocSys application, a magnetic system is not suitable because it can interfere with the electronic equipment, violating one of the system purposes. The inertial measurement unit based on electronic components (MEMS) can provide high accuracy, but they suffer from internal noise which result in growing position error with time. To reduce the accumulation of errors, these sensors need an eternal reference to correct them. Therefore, they could be used in combination with other localization systems.

The vision-based solutions can be adapted to many situations, thanks to the great variety of hardware available, in terms of optic camera quality. Their operation can rely on the recognition of fiducial markers or chosen features. The difference lies on the calculation time required for detection. Once the object to be located, whose geometry is known, is identified, its position can be assessed with respect to a reference system.

Concerning the wireless technologies, they include different solutions, here are some of them: Wi-Fi positioning system (WPS), Ultrawide band (UWB) and various methods for GPS indoor.

The WPS is a good solution where GPS is inadequate. It evaluates the position measuring the intensity of the received signal. The accuracy is closely related to the number of access points and signal fluctuations. In addition, it requires the implementation of devices on the TowerSat, with consequent increases in weight and consumption. It is a technology suitable for geolocation rather than for this application.

An UWB system is a low-power solution that does not interfere with other electronic devices and it could be a good technology for LocSys. In such a system, different transmitters are placed around the laboratory emulating the GPS satellites. A receiver is then placed on the moving structure and it would receive the signals. The position is evaluated using the triangulation equations. The disadvantages lie in the achievable accuracy of the order of 10 [cm]. [5]

Finally, regarding the GPS indoor approaches, the one adopted by the facility of the NPS is of particular interest. [4] In that application a laser GPS system was adopted. Each transmitter fixed on the laboratory walls emits a laser beam. A receiver implemented on the object receives the different signals, identifying the source of it. Knowing the nature of the beam and the location of the transmitters, the on-board computer can triangulate the receiver position. The system is very complex, and the on-board receiver requires a lot of power.

Remembering the objectives of the system and, in addition, the fact that it must not interfere with the simulated environment by the facility or with the test object, it was chosen to develop a system based on optical cameras. This solution allows flexible and completely customized development. Furthermore, it does not involve an addition of weight and consumption to the TowerSat. In fact, the system , can be completed outside the moving structure, except for markers, whose weight is negligible.

A system of this type allows to monitor the movements of the structure and to evaluate its orientation, using special markers.

The following thesis proposes the development of a localization system based on cameras, and aims to evaluate its performances, in terms of accuracy on position, orientation and frequency measurements in which results can be obtained.

3 LOCSYS: PROBLEM FORMULATION

As a reminder, the localization system aims to determine the position and orientation of the TowerSat with respect to a corner of the table.

Having chosen to use optical cameras to develop this system, is required to introduce a simple model that well represents the relationship between an object in three-dimensional space and its projection on the image plane of a camera. This chapter introduces the localization system from the mathematical point of view and therefore all the elements necessary for the development.

3.1 PINHOLE CAMERA MODEL

There are several models that describe how an optical camera works, which can be found in any book about computer vision. In this paragraph will be described the so-called "*pinhole camera model*", that although it is one of the simplest models, provides excellent results and it is implemented in the library used to develop the LocSys software.

In this model, a single ray of light enters the pinhole from any point in the scene. This point is then "projected" onto an image plane (also called the *projective plane*) that is always in focus, so the size of the image relative to the distant object is given by a single parameter of the camera: its *focal length* [1].

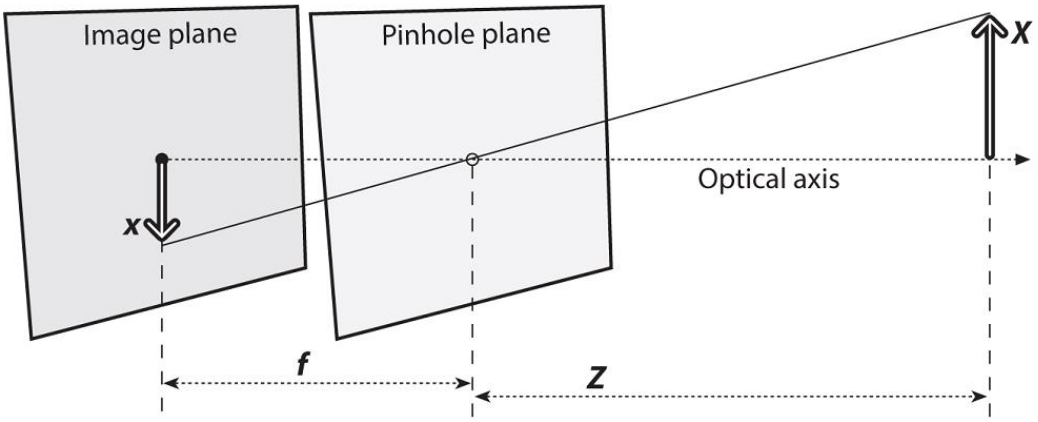


Figure 9: Pinhole camera model. ©[1]

The distance between the pinhole camera aperture and the image plane is precisely the focal length. This model is well represented in the Figure 9, where f is the focal length, Z is the distance from the camera to the object, X is the length of the object and x is the object image on the image plane. The figure shows by similar triangles the relationship between X and x. The same reasoning can be done for the other dimension of the object, which on the image plane is transformed into y. So, the *equations for the projection* of object points into image points are the following:

$$\begin{cases} x = -f \cdot \frac{X}{Z} \\ y = -f \cdot \frac{Y}{Z} \end{cases}$$

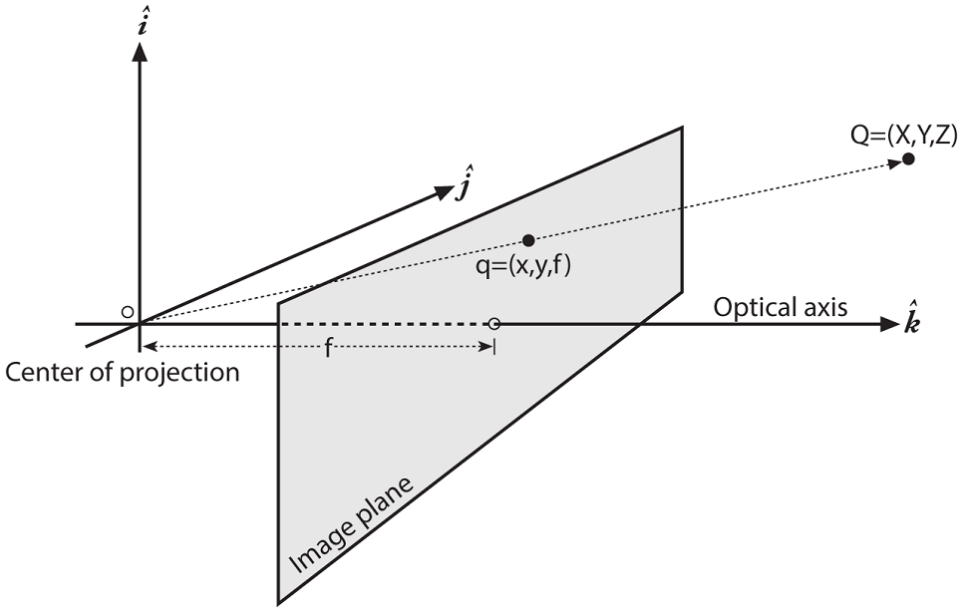


Figure 10: Rearranged pinhole camera model. ©[1]

It is now possible to rearrange the pinhole camera model in an equivalent form that is easier from the math point of view. In order to accomplish that, the pinhole and the image plane have to be swapped, so that the object now appears right side up, as it can be seen in the Figure 10. The old pinhole point is now represented by the *centre of projection*. In this way of looking at things, each ray that starts from the object is directed to the centre of projection and creates an intersection with the image plane. The point at the intersection of the image plane and the optical axis is referred to as the *principal point*. In this new form of the pinhole camera model, the image of the distant object is the same size as it was on the image plane in the previous form. This makes the similar triangles relationship $x/f = X/Z$ more evident than before. The negative sign is now gone because the object image is no longer upside down.

It is necessary to add that the centre of the imager chip is not on the optical axis, therefore two new parameters should be introduced, c_x and c_y, to model a possible displacement (with respect to the optical axis) of the centre of coordinates on the image plane. The result is that a relatively simple model in which a point $\vec{Q}$ in the physical world, whose coordinates are (X, Y, Z), is projected onto the image plane at some pixel location given by (x, y) in accordance with the following equations:

$$\begin{cases} x = f_x \cdot \dfrac{X}{Z} + c_x \\ y = f_y \cdot \dfrac{Y}{Z} + c_y \end{cases}$$

It has to be noted that two different focal lengths have been introduced; in fact, the individual pixels on a typical low-cost imager are rectangular rather than square, as the one that has been adopted during the LocSys development. Furthermore, the focal length f_x (the same goes for f_y) is actually the product of the physical focal length (usually in millimetres units) of the lens, adopted by the camera, and the size s_x of the individual imager elements (usually in pixels per millimetre units), which means that f_x is in the required units of pixels. It is important to keep in mind that it is not possible to measured directly via any camera calibration process the imager elements' size and neither is the physical focal length, so it is possible to derive only the combinations of them, without actually dismantling the camera and measuring its components directly.

The relation that maps a set of points $\overrightarrow{Q_i}$ in the physical world with coordinates (X_i, Y_i, Z_i) to the points projected on the image with coordinates (x_i, y_i) is called *projective transform*. For this kind of transform, it is convenient to use the well-known *homogeneous coordinates*. These coordinates associated with a point in a projective space of dimension n are typically expressed as an $(n+1)$-dimensional vector, with the additional restriction that any two points whose values are proportional are, in fact, equivalent points. In this application, the projective space is the image plane, therefore the image coordinates that has two dimensions are now represented by a three-dimensional vector $\vec{q} = (q_1, q_2, q_3)$. Recalling the additional restriction, the pixel coordinates can be recovered by dividing through by q_3. The homogeneous coordinates allow to rearrange the parameters that define our camera in a single 3×3 matrix (M), which is commonly called *camera intrinsic matrix*. [1] The projection of the object points in the physical world into the camera is now summarized by the following simple form:

$$\vec{q} = M \cdot \vec{Q}$$

where:

$$\vec{q} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \qquad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \qquad \vec{Q} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Multiplying this out, it easy to find that $w = Z$ and so, the pixel coordinate can be found dividing by w (or Z).

3.1.1 Camera Intrinsic Parameters

The intrinsic parameters of the camera are those parameters closely related to its physical nature. Some of them have already been introduced by the pinhole model, such as the focal lengths (f_x and f_y) and the optic centre (c_x, c_y) which compose the camera intrinsic matrix M. However, other parameters related to lens distortions must be introduced.

In theory, it is possible to define a lens without any kind of distortion, but, in practice, no lens is perfect mainly due to manufacturing reasons. There are mainly two types of lens distortions: *radial distortion* arises as a result of the shape of lens, whereas the *tangential distortion* arises from the assembly process of the camera. [1]

The lenses of real cameras often distort the location of pixels near the edges of the image plane. This phenomenon is the source of the "fisheye" effect. Figure 11 shows why this radial distortion occurs. Rays further from the centre of the lens are bent more than those closer in. Barrel distortion is particularly noticeable in cheap web cameras but less apparent in high-end cameras, where a lot of effort is put into fancy lens systems that minimize radial distortion. [6]

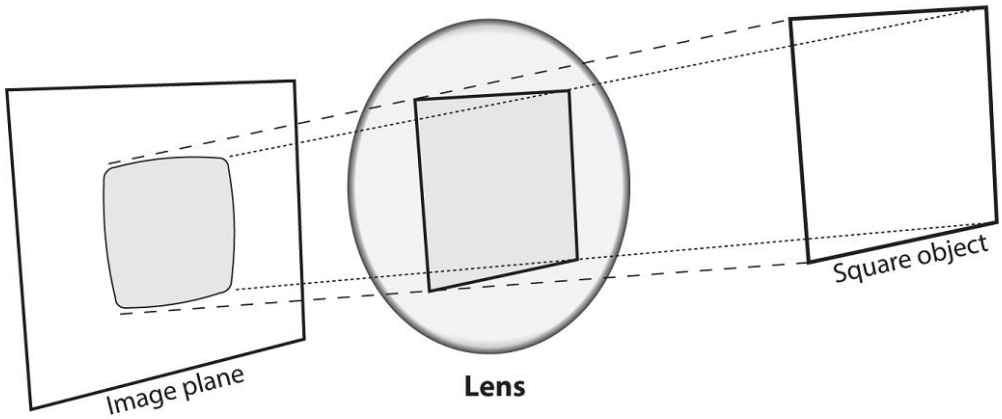


Figure 11: Radial distortion. ©[6]

The distortion is almost nil at the optical centre of the imager and increases toward the periphery. This kind of distortion can be characterized by the first few terms of a Taylor series expansion and, for cheap web cameras (like those used in this application), only the first two terms are used that are conventionally called k_1 and k_2. For highly distorted cameras such as fisheye lenses, an additional term is used, k_3.[6]

The tangential distortion is due to manufacturing defects resulting from the lens not being exactly parallel to the image plane, as is shown in the Figure 12. This distortion is minimally characterized by two additional parameters: p_1 and p_2.[6]

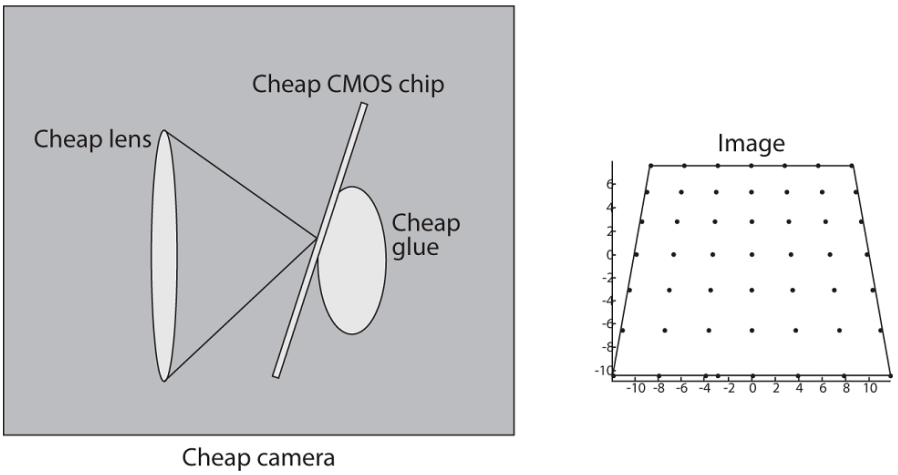


Figure 12: Tangential distortion. ©[6]

Thus, there are in total five parameters that correct lens distortion. These parameters are usually collected into a *distortion vector* and allow to transform the raw pixel coordinates $\vec{Q}$ into correct coordinates that best correlate the 3D point with its projection on the image plane $\vec{P}$.

3.1.2 Camera Extrinsic Parameters

The extrinsic parameters concern the relationship between the camera reference system and another arbitrary system. In fact, up to this point nothing has been said about the object reference system in the physical world. As the previous relationship was obtained, $\vec{Q} = (X, Y, Z)$ refers to the centre of projection. In general, points in space will be expressed in terms of a different Euclidean coordinate frame, known as the *world*

coordinate reference system.[2] The two coordinate frames are related through a rotation and a translation, as shown in the Figure 13.

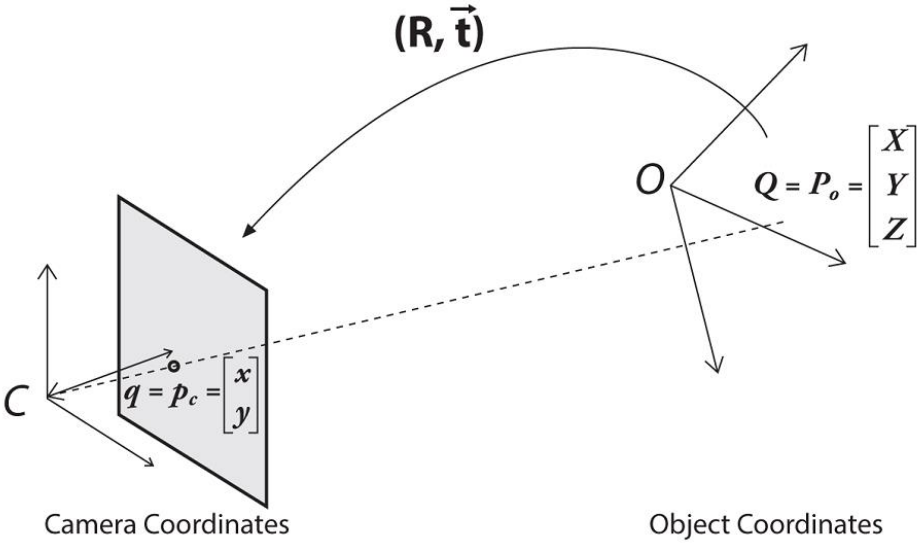


Figure 13: Camera and object coordinate frame relationship. © [1]

The translation between the two systems leads the origin of the object coordinate frame coinciding with that of the camera. It represents the offset between the two origins, and, in this application, it is the real distance from the object point to the centre of projection of the camera. The appropriate *translation vector* is simply:

$$\vec{T} = origin_{object} - origin_{camera}$$

Regarding the rotation, it allows an equivalent description of a point location in a different coordinate system. For these two systems a rotation in the three spatial dimensions is required, which can be decomposed into three different two-dimensional rotations, each around one of the x, y and z axes. In fact, a first rotation around the z-axis, then around the *new* position of the y-axis, and finally around the *new* position of the x-axis, with respectively rotation angles θ, φ and ψ, will result in a total *rotation matrix* R that is given by the product of the three matrices $R_x(\psi), R_y(\varphi)$ and $R_z(\theta)$, where:

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\psi & sin\psi \\ 0 & -sin\psi & cos\psi \end{bmatrix}$$

$$R_y(\varphi) = \begin{bmatrix} cos\varphi & 0 & -sin\varphi \\ 0 & 1 & 0 \\ sin\varphi & 0 & cos\varphi \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

thus:

$$R = R_x(\psi) \cdot R_y(\varphi) \cdot R_z(\theta)$$

The rotation matrix R is an orthogonal matrix, therefore has the property that its inverse is its transpose, hence:

$$R^T \cdot R = R \cdot R^T = I_3$$

where I is the 3×3 identity matrix.

3.2 Problem Formulation and Solution

It is now possible to relate a point in the object (or world) coordinate system $\vec{P}_O$ and his projection in the camera coordinate system $\vec{p}_C$:

$$\vec{p}_C = R \cdot (\vec{P}_O - \vec{T})$$

R and T are the extrinsic parameters of the camera and together with the camera intrinsic parameters allow correlating a defined point in three-dimensional space with its projection on the image plane. This relationship is well-defined by the following formula:

$$s \cdot \vec{p}_c = M(R \cdot \vec{P}_O + T) \qquad (3.1)$$

where a new parameter s has been introduced, which is an arbitrary scale factor derived from the introduction of the homogeneous coordinates.[6] This formula can be expressed in matrix form as follows:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \left(\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} \right)$$

It can be seen the point on the image plane $\overrightarrow{p_c}$ has been expressed as $[u \quad v \quad 1]^T$; this different notation is intended precisely to underline the difference between the raw pixel coordinates obtained from the original image $[x \quad y \quad 1]^T$ and the correct ones, by applying the corrections to the distortions previously described. Remembering that the

goal of the system is to determine the position of the object in 3D space using its projection on the image plane, it is possible to invert formula (3. 1) to get the 3D point $\vec{P}_O$ as follow (for greater clarity the points $\overrightarrow{P_O}$ and $\vec{p}_C$ are expressed in vector form):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R^{-1} \cdot M^{-1} \cdot s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} - R^{-1} \cdot T \tag{3.2}$$

Therefore, the formula (3.2) represents a system of three equations in four unknown variables. In fact, the camera intrinsic matrix *M*, the rotation matrix *R*, the translation vector *T* and the 2D point projection $\vec{p}_C$ are information obtainable with methods that will be described in the following paragraphs. Regarding the scale factor *s*, it is not possible to determine it, because, as mentioned above, it is an arbitrary factor.

For this reason, it is impossible to solve this system and to determine the position of an object in the three-dimensional space using only its projection on the image plane and a single camera. In order to solve this system, it is necessary to use at least two cameras or to introduce an assumption related to one of the three coordinates.

The object that the LocSys will have to determine is the TowerSat, which will move on a plane (the frictionless table), hence the *Z* coordinate will be fixed in the world reference system, that for this application will be a corner of the table. With this assumption and renaming the *Z* as Z_{cost}, it is possible to solve the system (3.2). The third equation in the Z_{cost} coordinate allows to evaluate the scale factor *s* and then solve the other two equations to find the *X* and *Y* coordinates of the object.

In order to accomplish this task, all the other ingredients should be determined. The following paragraph will describe how to obtain the camera intrinsic parameters (*camera calibration*), the camera extrinsic parameters (*pose estimation*) and an easy way to detect the object projection onto the image plane (through *fiducial markers*).

3.3 FIDUCIAL MARKER: ARUCO

The image of the object is captured by the camera and projects onto the image plane. Since this system is built and used in a known environment, it would be impractical to have to recognize a characteristic point of the TowerSat. In addition, LocSys should be able to provide TowerSat position and orientation at least 5 times per second.

For these reasons, it has been chosen to adopt some fiducial markers. A fiducial marker is a pattern placed in the field of view (FOV) of the camera, mounted onto the object and it is more easily detectable than the object itself, thanks to its nature and the application of some filters on the image.

For the LocSys purposes, the ArUco markers are the best solution.[8] ArUco is an Open Source library for detecting squared fiducial markers in images and it is already implemented in the OpenCV library. Additionally, if the camera is calibrated, that is the intrinsic parameters are known, it is possible to estimate the camera pose, that means to evaluate its position and orientation with respect to the markers.

These markers are comprised by an external black border and an inner region that encodes a binary pattern, as it's shown in Figure 14.[9]

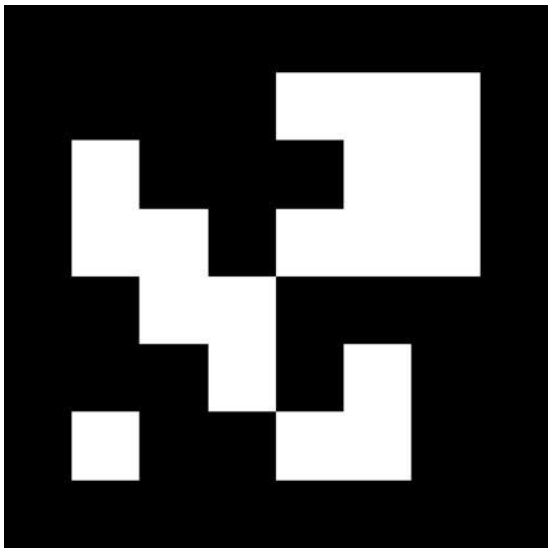


Figure 14: Example of ArUco marker

There are several types of ArUco markers, each of them belonging to a *dictionary*. The dictionaries differ in the number of markers they contain and the number of inner squares that encode the binary pattern. For example, the ArUco in Figure 14 has a binary pattern of 6x6 inner squares.

The advantages of these markers, in addition to their fast and robust detection [8], is that each of them provides a 4-point vector (representing the corners in the image) and an unique ID, which differentiates them according to their pattern and the dictionary to which they belong. Moreover, the pixel coordinates of the corners, contained in the 4-point vector, are sorted clockwise starting from the up-right corner. This fact gives to a marker its relative reference system, as is shown in Figure 15.

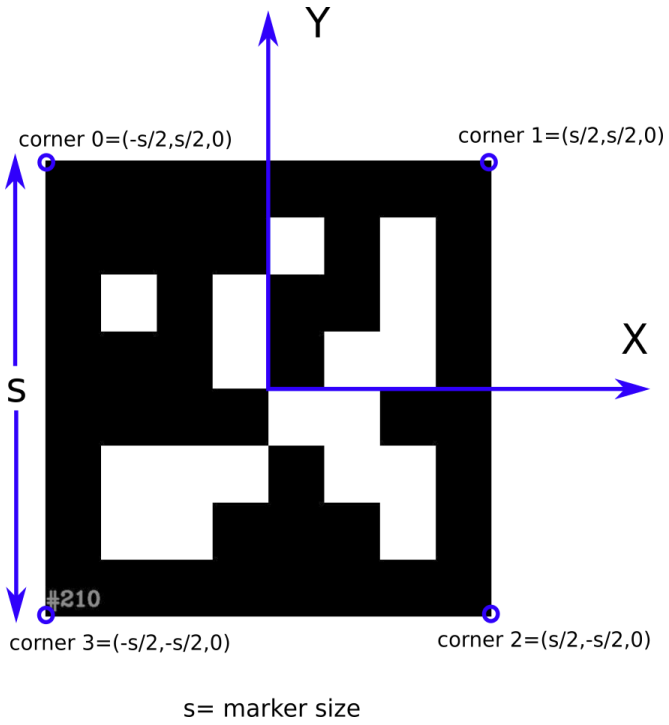


Figure 15: ArUco relative coordinate system. ©[9]

These considerations enable to parse an image to easily recover one (or more) point related to the 3D object.

3.4 CAMERA CALIBRATION

The camera calibration is a process aimed at estimating those parameters necessary to correlate the points of the real world with the image points captured through the camera. These parameters are already been introduced such as intrinsic parameters.

Once the camera is calibrated, it no longer needs to be calibrated again. The matrix M does not depend on the scene viewed. So, it can be re-used if the focal length is fixed. For this reason, zoom lens are not suggested.

Often, the accuracy of the camera calibration is strongly related to the total performance of the system related to the camera itself.

There are several calibration methods, which implement algorithms with the aim of solving the problem of minimizing a nonlinear error function to get accurate estimates for parameters. However, each of these methods is based on parsing different views of the same calibration pattern, in order to correlate some known points of the calibration pattern with their projections on the image. Using different views, usually at least 10 views are suggested [6], allows to evaluate the camera intrinsic matrix e the distortion vector. The OpenCV library implements two different algorithms that evaluate intrinsic parameters and optimize them.[10] Those algorithms are based on the Zhang calibration method [11] and the camera model proposed by Jean-Yves Bouguet [12]. Since these methods are already implemented and well described by the above references, it is worthwhile to pay more attention to the calibration patterns that can be adopted and the one used during this thesis.

Finally, one of the calibration process outputs is the *reprojection error*, that represents the quality of the calibration. The error metrics defined for estimation are done from the image points. It is usually done by computing the Euclidian distance between the projected image coordinate point, and the measured image coordinate point. [7] In the estimation process, the total error function will be minimized in the least squares fashion. The process is iterated where the reprojection error is computed during the iterations, and the final target is to get the minimum reprojection error. An optimal calibration quality is obtained when the reprojection error is less than 1 (pixel measurement unit).

3.4.1 Calibration Pattern

In principle, any appropriately characterized object could be used as a calibration object, but a more practical choice is a rectangular pattern on a flat surface, such as a *chessboard*, an *ArUco board* or a *ChArUco board*. For all these calibration pattern, some known points must be provided to the algorithm, so that it can find the corresponding projections.

A *chessboard* is a calibration pattern composed by alternating black and white squares. For this pattern, the object coordinates of the inner corners are known. In fact, using a Cartesian reference, that is usually centred in the lower left corner of the board, every inner corner is uniquely defined, as shown in the Figure 16. Since it is a flat surface, all the points in the reference system on the object itself have $Z = 0$. Regarding the X and Y coordinates, it is possible to number them simply by counting the number of inner corners, starting from zero, or use their measured distance from the origin of the object reference system. The output of the calibration process is the intrinsic parameters, which are in the pixel measurement unit, as explained in 3.1 paragraph, therefore the two formulations provide the same result.

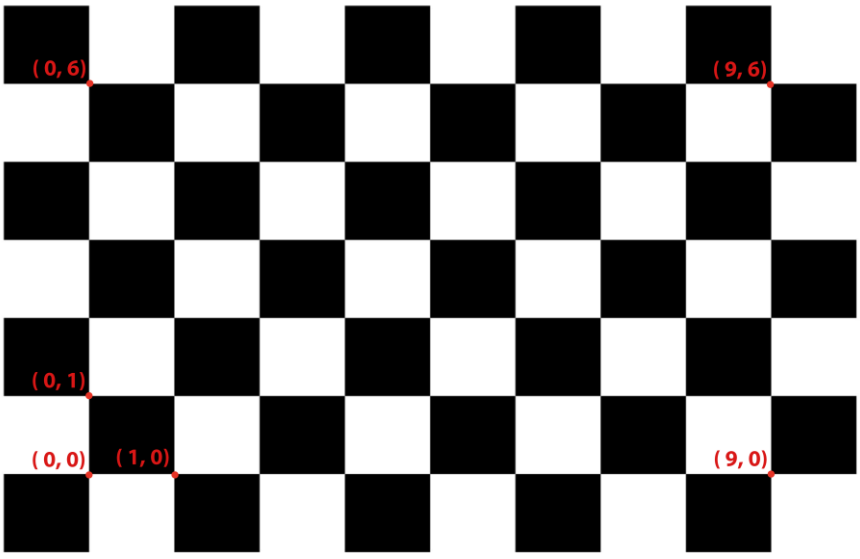


Figure 16: Chessboard calibration pattern

The use of a pattern of alternating black and white squares ensures that there is no bias toward one side or the other in measurement. Also, the resulting grid corners lend

themselves naturally to a subpixel refinement, increasing the accuracy of their detection.[6] A chessboard provides a number of known points given by:

$$num_{points} = (num_{raw} - 1) * (num_{column} - 1)$$

Furthermore, for the image to be properly used during the calibration process, all points must be visible and detectable. For this reason, it is recommended to use other patterns, which, under certain conditions, can also provide more accurate results.

One solution is the *ArUco Board* (see Figure 17), a board composed of $n = num_x * num_y$ ArUco marker, where num_x is the number of marker along the X axis and vice versa for num_y. This calibration pattern has two advantages over the more classic chessboard:

- The whole board does not have to be in view to get labelled corners on which to calibrate.
- Every marker gives four known points (corners), therefore if compared with a chessboard with the same number of rows and columns, it provides 4 times the number of object points

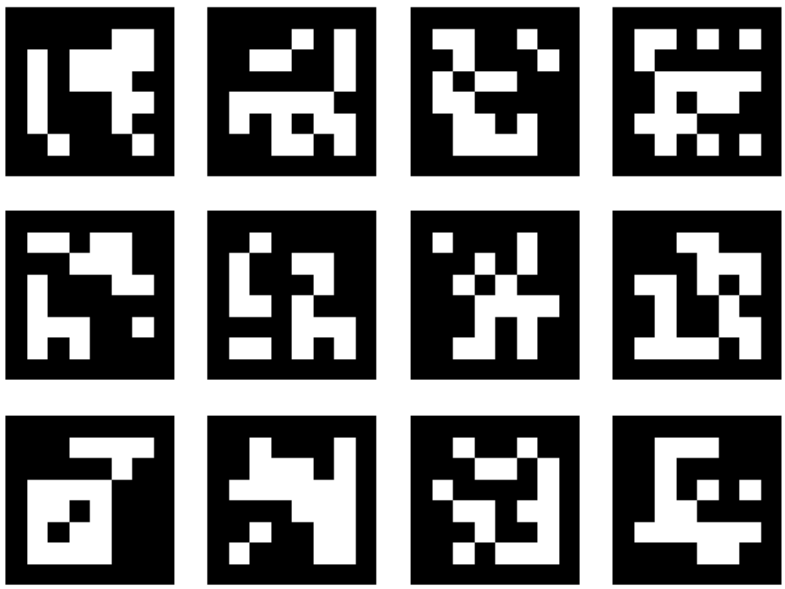


Figure 17: Example of ArUco Board

On the contrary, the corners of chessboard pattern can be refined more accurately since each corner is surrounded by two black squares.[10] However, finding a chessboard

pattern is not as versatile as finding an ArUco board: it must be completely visible and occlusions are not permitted.

Finally, the *ChArUco Board* has been created to combine the benefits carry out by the previous pattern (see Figure 18). In this board, each corner is labelled with an ArUco (2D barcode) pattern. This allows much of the calibration board to be occluded while allowing for the higher positional accuracy of corner intersections. The ArUco part is used to interpolate the position of the chessboard corners, so that it has the versatility of marker boards, since it allows occlusions or partial views. Moreover, since the interpolated corners belong to a chessboard, they are very accurate in terms of subpixel accuracy.

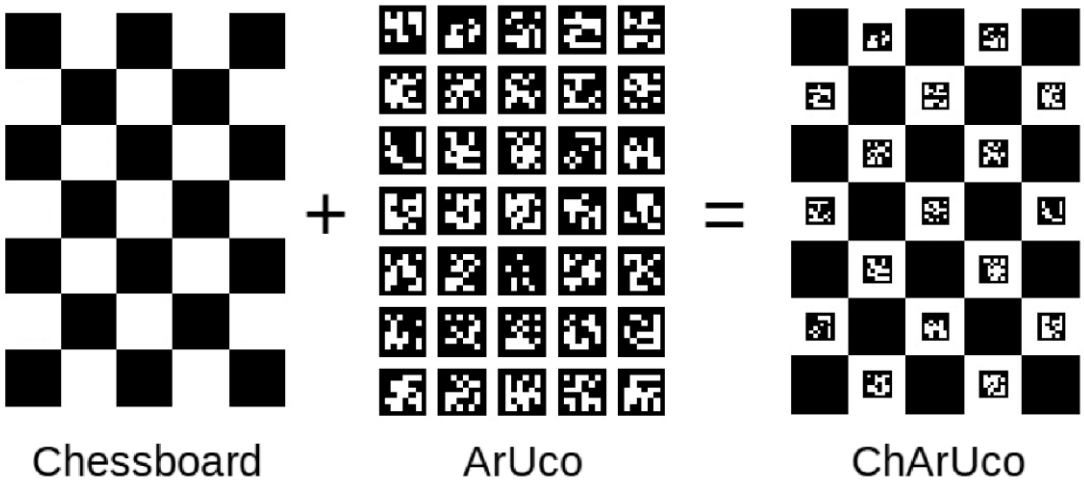


Figure 18: ChArUco Board creation concept

When high precision is necessary, such as in camera calibration, ChArUco boards are a better option than standard ArUco boards. For all these reasons, it was chosen to adopt the latter calibration tool.

3.5 CAMERA POSE ESTIMATION

To solve the equation (3.2), the translation vector T and the rotation matrix are still to be defined. These last ingredients, as mentioned before, identify the relative position and orientation between the camera coordinate frame, centred on its projection centre, and the desired reference system. This system is the same in which the coordinates (X, Y, Z), that are to be determined, are expressed.

The reference system should be centred in a corner of the table, therefore fixed in space. When the camera is also fixed in space, the relationship between the two reference systems is defined.

In literature, the problem of finding the pose of a calibrated camera given a set of n 3D points is well known and is called *Perspective-n-Point* (PnP). [6], [7], [13]

Without going into detail on how this problem is formulated, there are several methods and optimizations that solve it, depending on how many 3D-2D point correspondences are provided and how the points are positioned between them. A commonly used solution to the problem exists for $n = 3$ called P3P, and many solutions are available for the general case of $n \geq 3$, some of which, for example, are optimized if the points are coplanar.

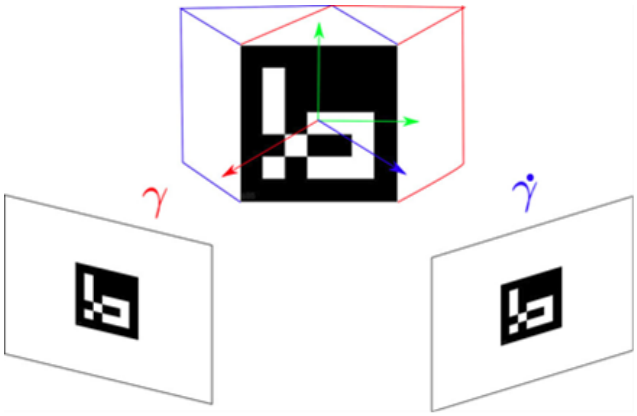


Figure 19: ArUco marker pose ambiguity. ©[9]

Since ArUco markers provide four detectable points, each marker could be used to estimate the camera pose with a planar pose estimator. It is important to remark that the estimation of the pose using only 4 coplanar points is subject to ambiguity. As shown in Figure 19, an ArUco marker could project at the same pixels on two different locations. Typically, the ambiguity can be solved if the camera is close to the marker. [9] At the

same time, as the distance between the camera and the marker increases, the marker becomes smaller, consequently the errors in the corner estimation grows and the ambiguity comes as a problem.

To overcome this problem, the external calibration can be done with a three-dimensional object, thus providing the correct direction of the axis orthogonal to the reference plane, removing the ambiguity. A non-planar calibration tool provides a better accuracy, but it requires more attention and precision on its utilization. It must be provided the exact 3D coordinates of the points that will be detected on the image (see an example in the Figure 20).

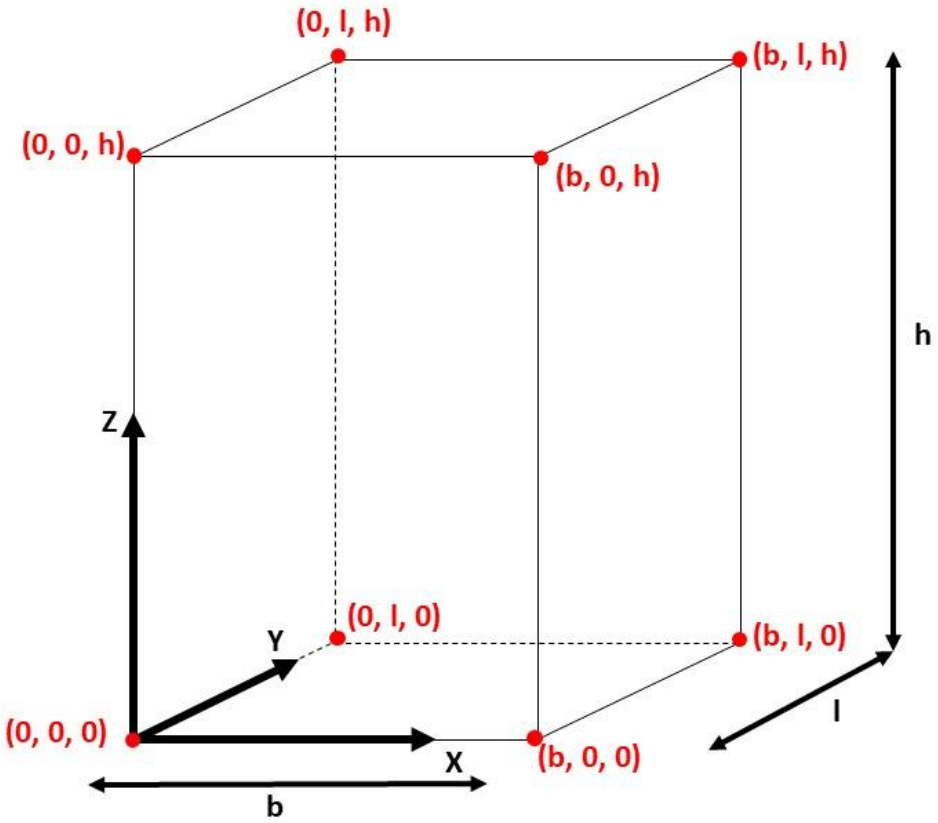


Figure 20: 3D object points example

Since the camera is fixed in space, the external calibration is to be done every time the camera intrinsic parameters change, or the camera is moved.

When the translation vector and the rotation matrix are determined, it is also possible to project a two-dimensional point in pixel coordinate into space, having coordinates centred in the same reference system adopted in the external calibration. This is the concept developed in the LocSys.

3.6 MULTIPLE CAMERA ARCHITECTURE

The previous section explained the operation of a camera, the processes to be carried out and the necessary tools. If one of the three object coordinates is known (usually the z coordinate), it is possible to determine its position with a single camera.

An architecture consisting of a single camera has obvious restrictions:

- The camera FOV is limited and consequently the operational area of the TowerSat which it can cover is limited. Increasing the distance between the TowerSat and the camera would lead to a decrease in accuracy, as it will be shown in the chapter 5. It is not even possible to tilt the plane of vision to see a larger area in a perspective way, because a distant object would suffer from major errors.
- The accuracy of the system is entrusted to a single camera, and in the presence of noise in the image the measurements would also be affected, without possibility of corrections.
- The edges of the image are more prone to distortion, as explained in section 3.1.1. Although the camera calibration and the image undistortion are carried out, some errors are still occurred. For this reason, the area covered with acceptable errors is reduced.

The first point would be enough to understand the need to use multiple cameras, to cover a larger region, without losing accuracy. There are several ways to use multiple cameras and overcome the limits just described. Two of them are explained below: the first is interesting, especially for future developments, while the second is the one that has been implemented in the LocSys.

3.6.1 Stereo Vision

The use of two cameras aimed at the same object is known as *Stereo Vision*. The operation is analogous to human sight, allowing to know the depth at which an object is located from the centre of the system composed by the cameras. With this methodology it is possible to know all the point coordinates, without to resort to assumptions. Therefore, the stereo vision allows to *triangulate* the position of a point. The mathematics that describes this type of system is called *Epipolar Geometry*.

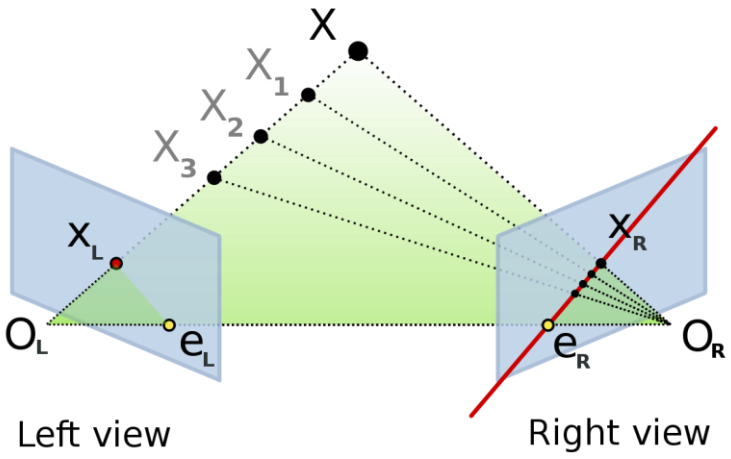


Figure 21: Epipolar Geometry

The Figure 21 shows a basic setup with two cameras taking the image of the same scene. Using only the left camera , as described above, does not allow to find the 3D point corresponding to the pixel point x_L in the image, because every point on the line $O_L X$ has the same projection on the image plane. Considering the right image, the different points on the line $O_L X$ has different projection on the image plane. These projections on the right image plane form a line (the red line in the figure above) that is called *epiline* corresponding to the point X.

As the distance of the X point from the left camera changes, its projection on the right image plane changes. Knowing the characteristics of this point, it is possible to reduce its research only on the epiline (or in its surroundings). This is called *Epipolar Constraint*.

The points O_L and O_R are the camera centres. These two points together with the X point form the *Epipolar Plane*. Furthermore, the projection of the right camera centre O_R on the left image is called *epipole* (e_L). An epipole is the point of intersection of line through the camera centres and the image planes. If one camera does not see the other, the epipole may be outside the image plane. The characteristic of the epipole is that each epiline passes through it.

The position of the epipoles is determined only by the reciprocal position of the cameras. Therefore, once the translation and rotation vector that lead to the right image plane coincide with the left one (see Figure 22), it is possible to reconstruct the 3D position of the X point.

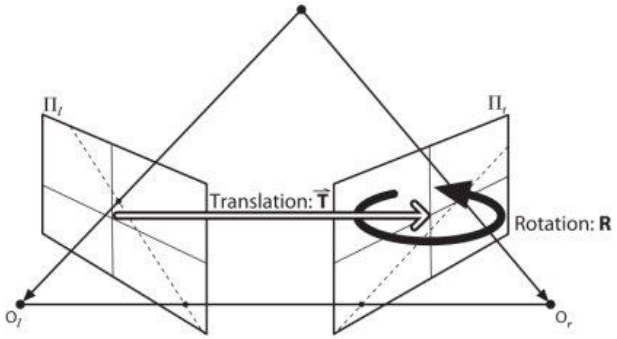


Figure 22: Left and Right image planes relationship

This geometry has some advantages over using the one camera architecture, but it has disadvantages too. These are resumed in the table below:

Advantages	Disadvantages
Depth evaluation (z coordinate)	More complex system
Higher accuracy	Reduce the operating area
	More cameras required to cover a larger area

Although it allows the determination of all three spatial coordinates and provides greater accuracy, the complexity of the system increases. The object must be seen simultaneously by two cameras at every point in the area where it can move. In fact, the field of action of a single camera is reduced, because in order to function properly the object must be seen by both (see Figure 23). The number of cameras increases.

The stereo vision is an excellent solution for future development of the Frictionless Table System, when knowledge of the Z coordinate will be required. In a first configuration of the project, the main objective is to increase the working area provided by a single camera.

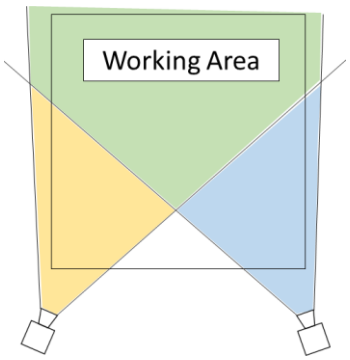


Figure 23: FOVs intersection

3.6.2 LocSys Solution

In this application case, it is necessary to determine two coordinates on the table and a rotation around the perpendicular direction to the table plane. As has been previously described, one camera and one ArUco marker are able to provide this information.

The stereo vision does not solve the restrictions given by one camera, so the idea implemented in LocSys is to use the concept adopted in the single architecture serval times.

Each camera can determine the object position with respect to the reference system (RS) adopted for that camera. If several cameras are used, each with its own reference system, and the position of the RS relative to the main one is known, it would be possible to cover a larger area.

When the marker enters a camera FOV, it evaluates its position with respect to its reference system. Note the position and, if necessary, the orientation of this RS, a translation and a rotation to the evaluated position are applied. The problem is transformed into the creation of a precise geometry, where the reference systems play the main role.

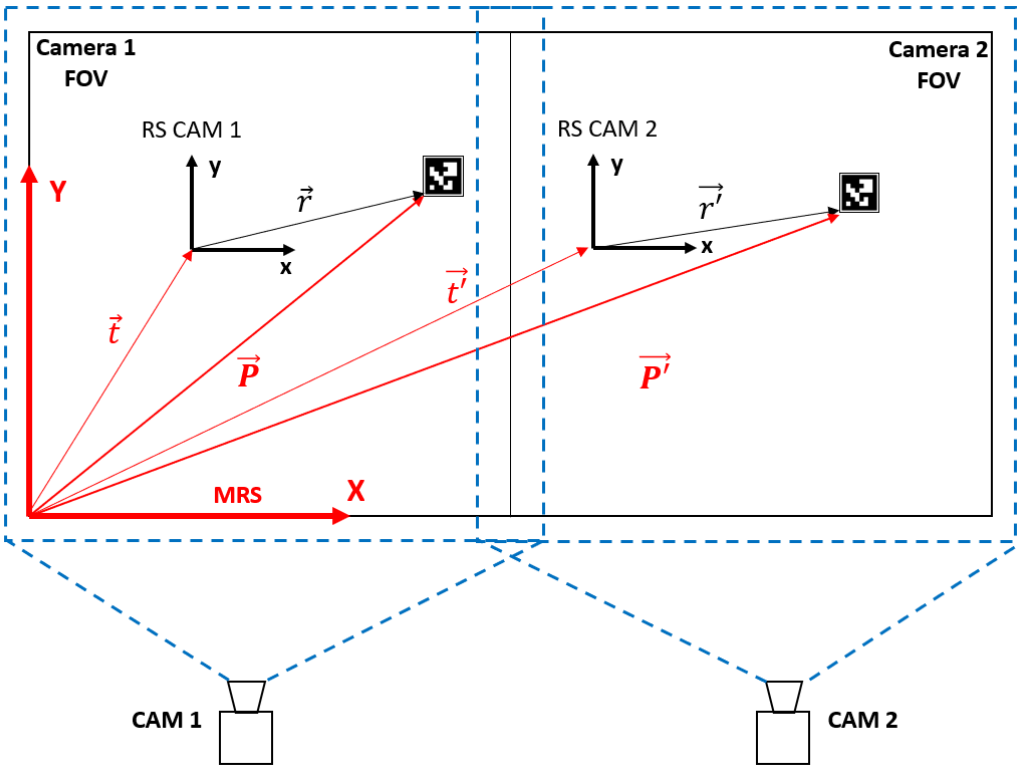


Figure 24: LocSys reference systems

The Figure 24 shows the relationships that are created between the position, evaluated by each camera with respect to its own reference system, and the main reference system (MRS). Since this is the developed architecture, it will be described in more detail in the following chapters, especially when the software is described.

To completely cover an area for which a single camera would not be enough, overlapping areas will be created, where the object can be seen by multiple cameras. Each of them would evaluate the position independently. Subsequently, not being able to give priority to one measure over another, it is necessary to average the evaluated positions, to decrease the highest error made by one of the cameras.

The limits that arise with this architecture are related to the ability to create a precise system, to synchronize the cameras and to the computing power. As the number of cameras increases, the frames to be analysed increase, reducing the number of evaluations per second. The need also arises to manage all the information concerning the cameras and the position of their reference system.

Compared to stereo vision, this solution allows fewer cameras to cover the same operating area. The overlapping area of the FOV is reduced to a minimum, while in the other case it must be maximized.

In the case of overlapping, the number of measurements increases, providing greater reliability to the system. Using more cameras could increase this area, increasing accuracy and precision.

The following chapters describe the software developed, necessary to determine the performance of a single camera and then the architecture with multiple cameras.

4 LOCSYS: SOFTWARE DEVELOPMENT

The goal of this thesis is to develop a baseline architecture to achieve the objectives imposed on the system. The core of this architecture is the software that manages the multiple cameras and thus evaluates the position and orientation of the TowerSat. In order to develop the LocSys software, it has been chosen to use the Python programming language, with the implementation of the OpenCV library. In this chapter, after a brief description on the choice of Python and OpenCV, the developed software will be explained. A first software has been adopted to perform performance tests with a single camera, while another is able to manage multiple cameras, as the LocSys is supposed to do.

4.1 SOFTWARE METHODOLOGY AND TOOLS

The following section describes the tools adopted for the thesis and the methodology for developing the LocSys software. The tools include the programming language, the cameras and the calibration pattern for intrinsic and extrinsic parameters.

4.1.1 Python

Python is a popular programming language, designed for readability and functionality. It has a simple syntax like the English language, that makes most codes easy to be understood even by a beginner. The Python language has a specific syntax and semantics, which allow to express computations and data manipulation by a computer with fewer lines than some other programming languages. It uses new lines to complete a command, as opposed to other programming languages, which often use semicolons or parentheses. It relies on indentation, using whitespace, to define scope (e.g. loops, functions, classes) rather than curly brackets, like other programming adopt.

Python is an interpreted language. In fact, the host CPU does not execute directly any Python program, but it needs an *interpreter* that execute it. The interpreter executes a Python code as soon as it is written. This is one of the major distinctions with the C codes, which are compiled to CPU machine code before the run time.

Another characteristic is that it is *dynamically typed*. The interpreter performs the type checking at run-time, as opposed to at compile-time. The programmer does not have to declare variable types himself. There are other dynamically typed languages, like JavaScript, Ruby and Objective-C.

Python is also *strongly typed*, meaning that it will raise a run-time type error when the programmers has violated the Python syntax rule. For all these reasons, the prototyping can be very quick.

The last property is that Python is an object-oriented programming language. It is based on *object*, which contain *data* (often known as *attributes* or *properties*) and code (often known as *methods*).

The *class* object was widely used in LocSys. A class is an extensible template for creating object, providing initial member variables and implementing member functions. Once the class is created, it allows to create an *instance* of the class. Every instance of the class owns the attributes and the methods of the class itself. To better understand this programming-concept, will follow a short example.

Let's think about a class called *Car* (usually a class name begins with a capital letter). The class Car has as initial *attributes* the model of a car (*car_model*) and the percentage of tank filling (*tank_percentage)*. This class has also two *methods* (functions) called *drive* and *refill*. When *drive* is called, it removes a 10% of tank filling, while *refill* assigns a 100% of tank filling. Thus, the Car class has two initial attributes and two methods.

```
# Class creation
class Car(object):

    def __init__(self, car_model, tank_percentage):
        self.car_model = car_model
        self.tank_percentage = tank_percentage

    def drive(self):
        self.tank_percentage = self.tank_percentage - 10

    def refill(self):
        self.tank_percentage = 100


# Cars instances
car_1 = Car("Ford Mustang", 80)
car_2 = Car("Chevrolet Camaro", 50)

print()     # Print an empty line
print("Your first car, {}, has a {} %  of full tank".format(car_1.car_model, car_1.tank_percentage))
print("Your second car, {}, has a {} %  of full tank\n".format(car_2.car_model, car_2.tank_percentage))

# Drive both the cars
car_1.drive()
car_2.drive()

print("After the guide, your first car, {}, has a {} %  of full tank".format(car_1.car_model, car_1.tank_percentage))
print("After the guide, your second car, {}, has a {} %  of full tank\n".format(car_2.car_model, car_2.tank_percentage))

# Now refill only the second car
car_2.refill()
print("After the refill, your second car, {}, has a {} %  of full tank".format(car_2.car_model, car_2.tank_percentage))
```

Figure 25: Example of class creation

Let's create two different instances for two different cars. The first car is a Ford Mustang with an initial 80% of tank filling. The second car is a Chevrolet Camaro with an initial 50% of tank filling. Now, both the cars can be driven recalling the method *drive*. After this command, the Ford Mustang has a 70% of tank filling, while the Chevrolet Camaro has a 40%. Calling the *refill* method to the Chevrolet, it refills its tank to the maximum value of 100%. The Figure 25 shows the Car class creation and the implementation of two instances. The Figure 26 shows the terminal output of the code in the previous figure.

```
Your first car, Ford Mustang, has a 80 %  of full tank
Your second car, Chevrolet Camaro, has a 50 %  of full tank

After the guide, your first car, Ford Mustang, has a 70 %  of full tank
After the guide, your second car, Chevrolet Camaro, has a 40 %  of full tank

After the refill, your second car, Chevrolet Camaro, has a 100 %  of full tank
```

Figure 26: Example of class execution

So, once a class has been created, it is possible to instantiate multiple objects that share the same "nature". It allows to recall its methods and attributes, without having to declare variables and, in principle, without knowing how the class is made internally. This is the concept adopted in many libraries implemented in Python (like OpenCV) and in other programming languages.

A class object also makes a code easier to read. It can be created and modified in one python (.py) file and then instantiated in another file, where its methods and attributes are exploited. This allows to modify, for example, the main program file, without interfere to the class and, vice versa, to modify the class without interfere to the main. The thesis adopts this methodology for the development of the LocSys software.

4.1.2 Python Multithreading Approach

To develop a multi-camera architecture, it is necessary to introduce the concept of multithreading, as the cameras should work synchronously.

In Python this is called concurrency and it is the occurrence of two or more events at the same time or, in terms of programming language, it is the overlapping of two tasks in execution. The performance of software systems can be improved because it can

concurrently deal with the requests rather than waiting for a previous one to be completed.

A *thread* is a small unit of execution that can be performed in an operating system. It is not itself a program, but it runs within a program, hence, threads are not independent from one other. Each thread shares code section, data section, etc. with other threads.

Therefore, multithreading is the ability of the CPU to manage and control the use of operating systems be executing multiple threads concurrently. The main concept of the multithreading philosophy is to achieve parallelism by dividing a process into multiple threads.[14] For this thesis, the cameras should get the frames at the same time and then these frames should be analysed. With multiple cameras, if frames were taken in sequence, synchrony would be lost. Taking them in parallel, the synchrony should be maintained, always considering the limits of the CPU.

Typically, a thread can exist in five different states (see Figure 27):

- **New Thread**: a new thread begins its life cycle in the new state. At this stage it is not started yet, and it has not been allocated any resources.
- **Runnable**: the thread is started, and it is waiting to run. It has all the resources.
- **Running**: the thread executes its tasks. At this stage, the thread can go to either the dead state or to the non-running/waiting state.
- **Non-running/waiting:** the thread is paused, and it is waiting for the response of some I/O request.
- **Dead:** the thread enters the terminated state when it completes task.

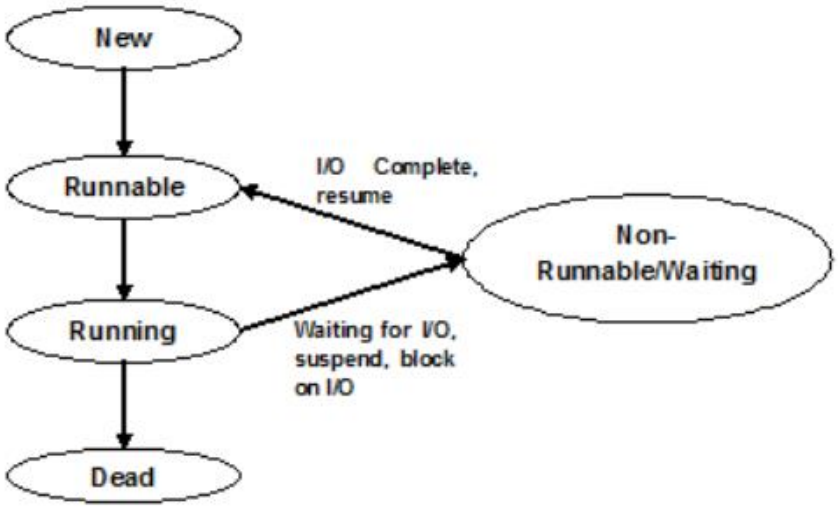


Figure 27: Thread life cycle. ©[14]

A process is defined as an entity, which represents the basic unit of the code implemented in the system. In other words, the programmer writes his program and when he executes it, it becomes a process that performs all the tasks in the program.

For this thesis, the multithreading approach was used to synchronize the camera frame acquisition. The main program will be executed by the main thread, instead each camera video dataflow will be handled by an independent thread.

4.1.3 OpenCV

OpenCV is an open source cross-platform programming library aimed at computer vision available from http://opencv.org.[10] This library was fundamental for the development of the software, because it implements many functions useful to handle cameras and to manage and parse images. All these functions can be easily found on its online documentation.

It also includes functions able to intrinsic calibrate the cameras, giving as input some parameters. One of these is *calibrateCamera* uses several views of a calibration pattern.

OpenCV provides the function *solvePnP*, that, as mentioned in section 3.5, allows to evaluate the pose of the camera with respect to a calibration tool. This function uses the 3D-2D point correspondences and different methods for solving the PnP problem.

An extension of this library includes the module that allows the recognition of the ArUco markers. This module provides the function to calibrate a camera using a ChArUco board. The function takes as input the interpolated points of the board corners and the information of the board used. The difficulty in using this function lies in extrapolating this data from the frames and proving it correctly.

4.1.4 Software Requirements

The development of LocSys is made with Python 3.7.5. During the installation of Python, available from https://www.python.org/downloads/, the *pip* is also provided. Pip is a package manager system used to install and manage software packages written in Python. The pip is used through the command-line interface, which makes installing packages as easy as issuing a command:

```
pip install some-package-name
```

In order to make the LocSys software work, the following modules should be installed:

- **NumPy**: the fundamental package for scientific computing. It helps managing N-dimensional array object

```
pip install numpy
```

- **OpenCV**: Open source Computer Vision and machine learning library. The extended version implements many modules, like the mandatory ArUco module

```
pip install opencv-contrib-python
```

- **TQDM**: show a smart progress in loops

```
pip install tqdm
```

The other modules used are already implemented in Python, they are *built-in* modules.

4.1.5 Software Directory Structure

To better understand the software, the Figure 28 shows the directory organisation in which the software, the external *.txt* files for the configurations, the results and the data of the cameras are contained.

All the Python files are contained in the main directory, where there are two folders: *Data* and *Results*. Inside the *Data* folder, the configuration files are listed. These are *.json* files and it is possible to open them with any text editor. There are also two other sub folders, *Cameras* and *Markers*, in which, respectively, are saved the intrinsic and extrinsic parameters of a specific camera, and the ArUco markers. It is possible to generate the ArUco with the *marker_generator.py* program, placed in the main directory, and save them in order to be printed and used for the tests. All the results produced by the main program are saved in a *.json* file inside the *Results* folder.

In the following sections, the content of each program will be explained, starting from the classes and then defining the main software.

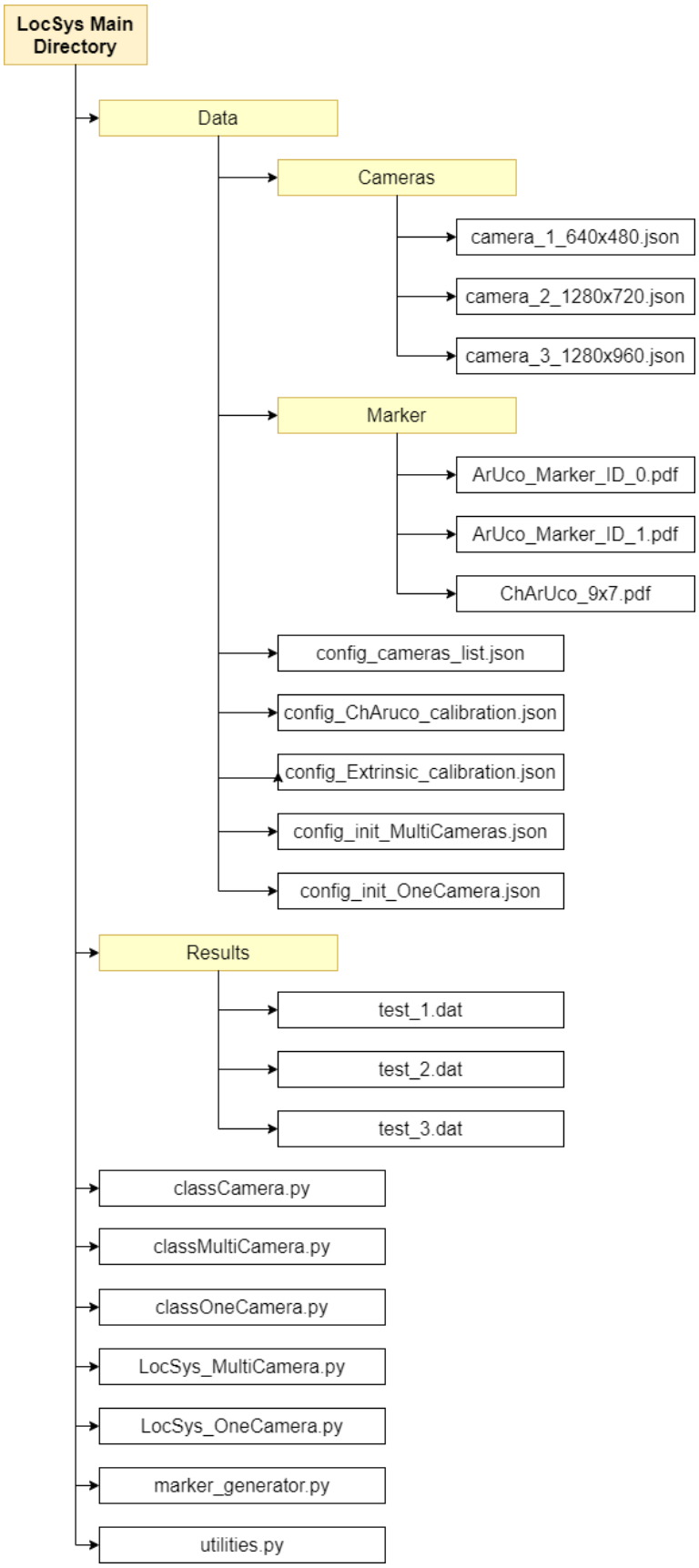


Figure 28: LocSys Directory Structure

4.2 CAMERA CLASS

The *classCamera.py* is the first program to be described, because is the core of LocSys software. It is widely used in both mono and multi-cameras architecture. This program is a class object. It initializes and manage a camera with all the benefits of using a class (see section 4.1.1). The *classCamera* is composed by the methods shown in Figure 29.

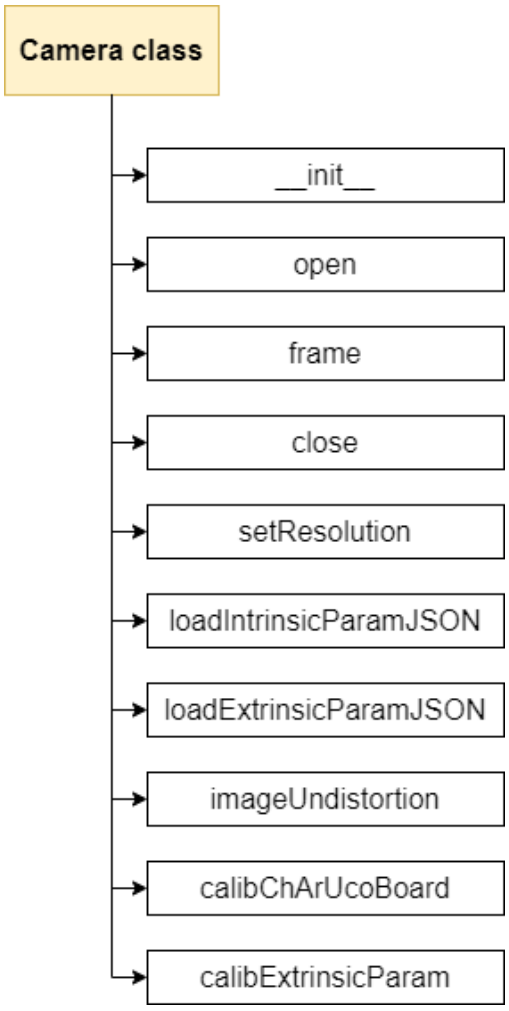


Figure 29: Camera class methods

4.2.1 Basic methods

When declaring this class, it must be initialized the camera reference ID and the name to be attributed to it, and this is performed by the *__init__* method. The camera ID is

the USB port in which the camera is plugged in. For further information and future developments, see [10]. In this, the camera brand was used as camera name.

The *open* method allows to open the camera data flow, providing a desired resolution. It would warn the user if something went wrong during any steps.

The *frame* method provides a single frame of the video data flow. This is simpler than OpenCV built-in function.

The *close* method simply closes the video data flow of the camera.

The *setResolution* method allows to change the camera resolution, providing the desired image width, height or both. The function will evaluate the proper camera resolution if the camera itself does not support the desired one. Finally, it saves the frame width and height as attribute of the camera class.

The *imageUndistortion* method provides the undistorted image given the original one. It uses the camera parameters that are saved as camera attributes.

4.2.2 *loadIntrinsicParamJSON* and *loadExtrinsicParamJSON* methods

These two methods allow to load the respectively intrinsic and extrinsic camera parameters from an external *.json* file. The external file name must be in the form '*cameraName*' _ '*width*' x '*height*' *.json* (e.g. *microsoft1_1280x720.json*). The parameters are then saved as attributes of the camera class.

4.2.3 *calibChArUcoBoard* method

This method implements one of the two most crucial steps for the correct functioning of the LocSys, that is the calibration of intrinsic parameters (see 3.4). The logical flow implemented in the function is shown in Figure 30:

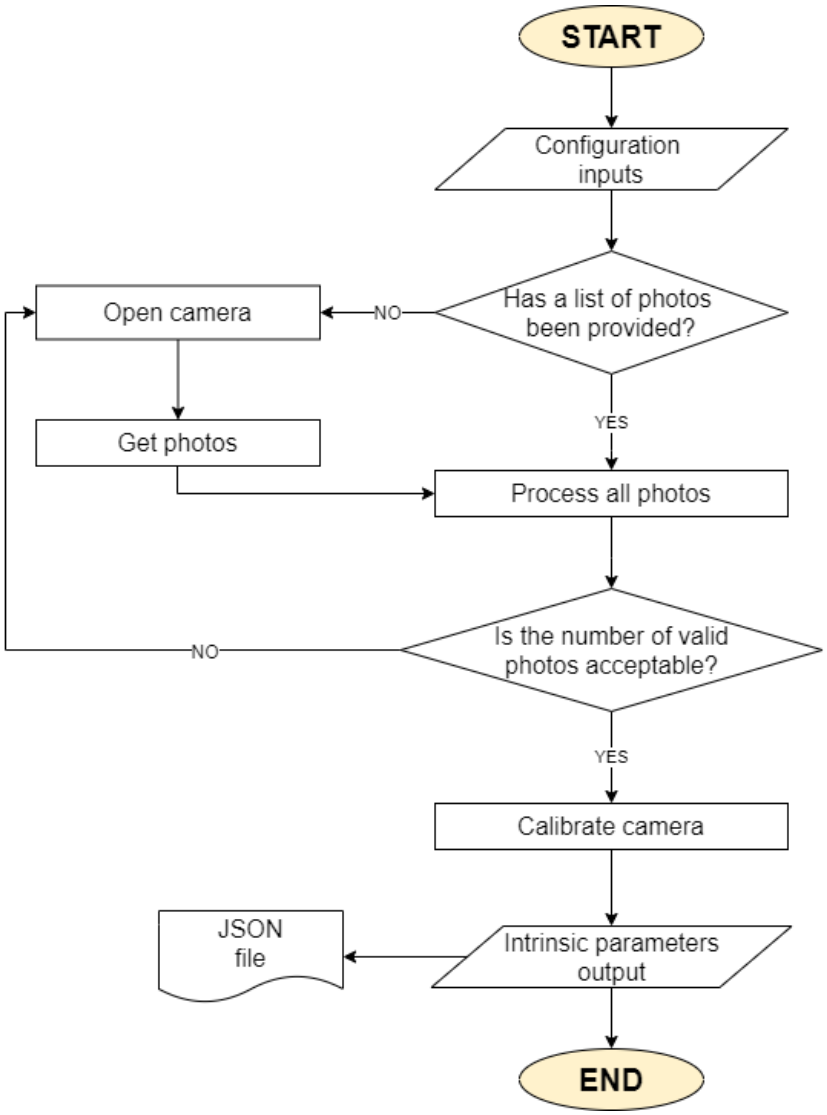


Figure 30: calibChArUcoBoard method flowchart

First, it takes as input the configuration settings that are adopt for calibration. These are contained in the *config_ChArUco_calibration.json* file (Figure 31).

```
{
    "squaresX": 9,
    "squaresY": 7,
    "squareLength": 3.9,
    "markerLength": 2.4,
    "min_valid_frame": 10
}
```

Figure 31: ChArUco calibration configuration

They are:

- The **calibration pattern geometry**, understood as the number of rows, of columns, the **length of a square side** in centimetres, the **length of an ArUco marker** in centimetres (see Figure 32)
- The **minimum number frames** that are considered valid to be used for the camera calibration process

For the correct operation of the software, the variable name between the quotation marks must not be changed, but only the values placed after the colon. In addition, it is advisable to print the pattern in the largest possible size and make the surface opaque.

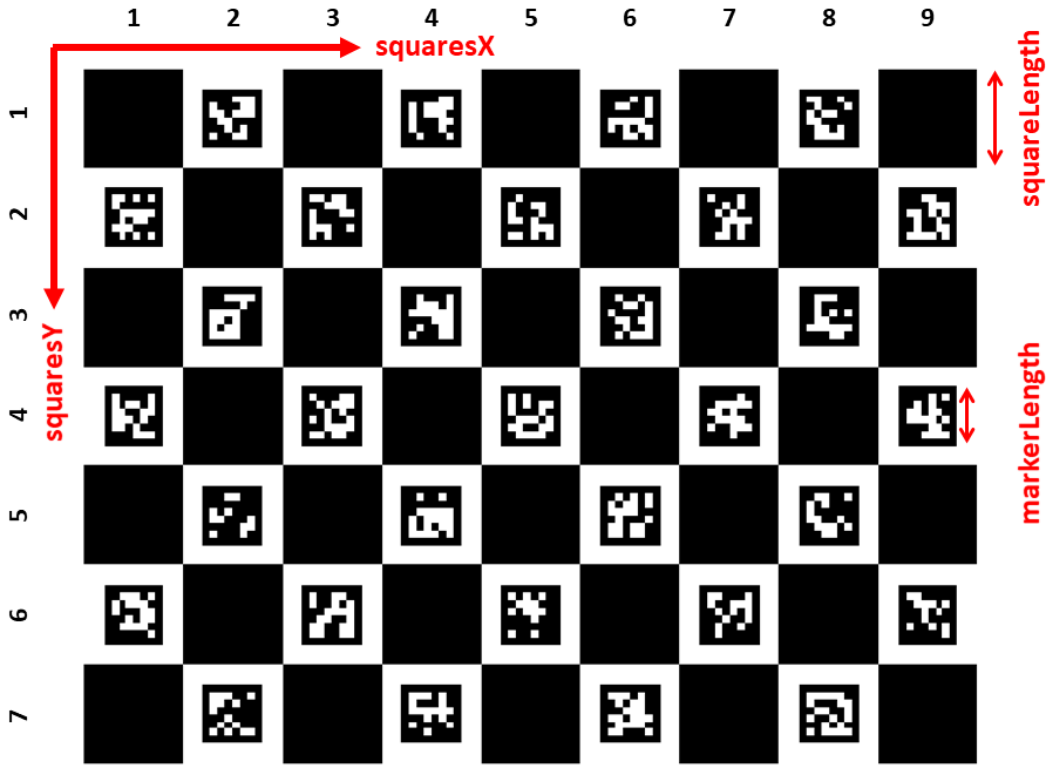


Figure 32: ChArUco board geometry

If none frames (photos) are provided as input to the method, it opens the camera video flow to take them, using the terminal interface to guide the user. After, it processes all the photos, detecting the ArUco markers and then interpolating the inner corners of the chessboard. If the number of valid frames meet the minimum requirement, it saves the results in the external camera file and as attribute of the camera class. The external file has the form explained in 4.2.2.

To perform a correct camera calibration, it is recommended to use at least 15 different photos of the calibration pattern, the ChArUco board. The photos must contain the pattern seen from different views, tilting it from different angles, from near and far the

camera. In this way, the built-in function of OpenCV can find the correct correlation between the physical points of the board with their projection on the image. The correlation is intended as the intrinsic parameters of the camera, according to what was previously said in section 3.1.

4.2.4 *calibExtrinsicParam* method

To determine the position and orientation of the test object, the camera pose with respect to a reference system must be known (the *R* matrix and the *T* vector described in section 3.1.2). If the camera remains fixed in space, this process can be done only once using this method.

In theory, any object could be used by providing the coordinates of the 3D points to be recognized. In order to improve the accuracy of this process, a 3D calibration tool with two tilted ArUco markers was created (see Figure 33).

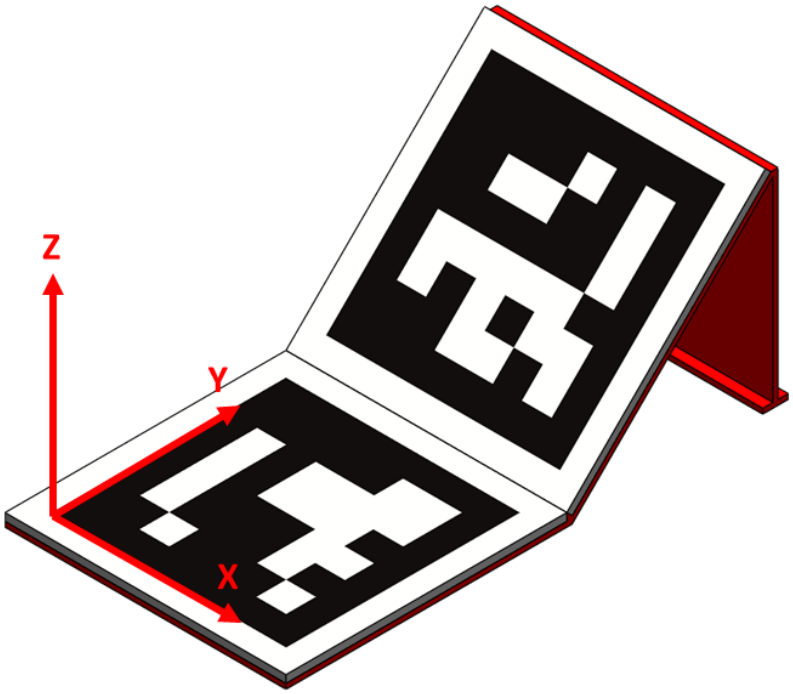


Figure 33: 3D calibration tool with its reference system

As previously said, each marker provides 4 easily identifiable points, as shown in Figure 34. Therefore, after calculating their position in the object reference frame, it is possible to find the rotation and translation to be implemented to transform the reference system of the camera into the desired one.

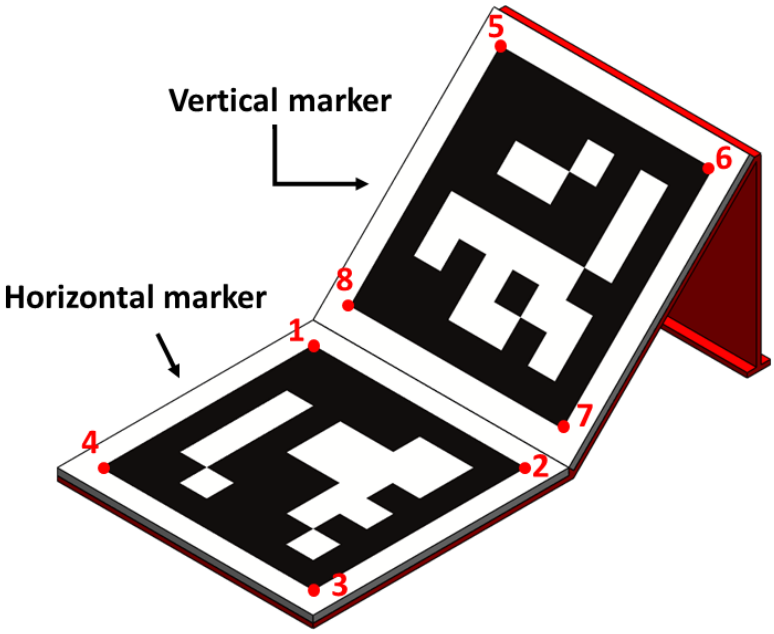


Figure 34: 3D calibration tool corners

The reference system of the calibration pattern does not have to coincide with that desired for the specific camera. In fact, due to physical limits, the reference system can be translated and rotated. This concept will be explained in next chapter, where the tests conducted will be described.

The Figure 35 shows the variables that allow to evaluate the relative position between the two ArUco markers, (d, h, φ). By providing these information and the relative position and orientation of the reference system $(x_{rel}, y_{rel}, z_{rel}, \theta)$, the object points in the desired reference system are assessed by the formulas in Table 2.

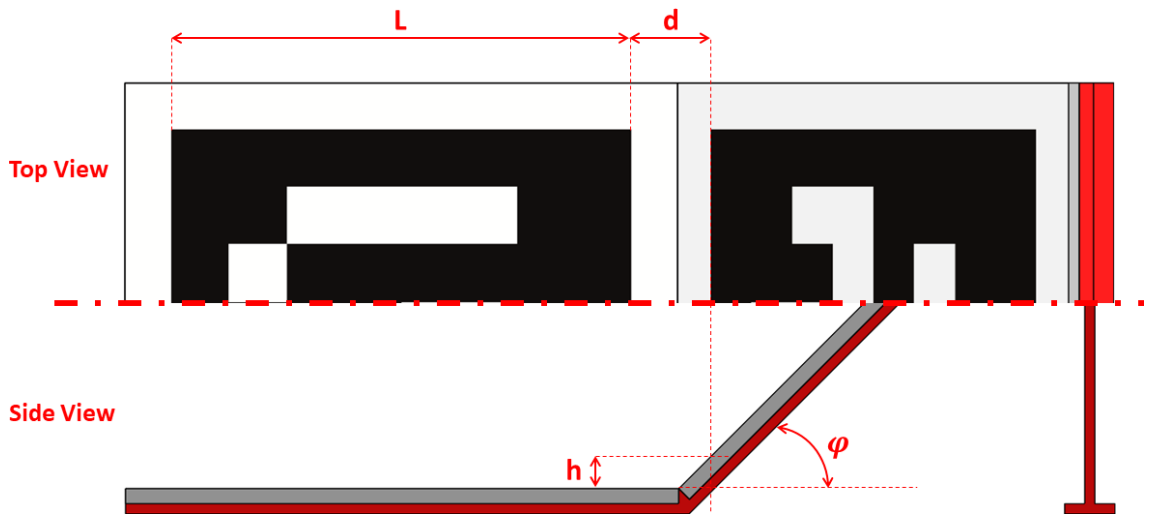


Figure 35: 3D calibration tool sizes

Table 2: Extrinsic calibration tool corners

Corner	Coordinate formula	
1	X	$L \cdot \cos\left(\frac{\pi}{2} + \theta\right) + x_{rel}$
	Y	$L \cdot \sin\left(\frac{\pi}{2} + \theta\right) + y_{rel}$
	Z	$0.0 + z_{rel}$
2	X	$L \cdot \frac{\cos\left(\frac{\pi}{4} + \theta\right)}{\cos\left(\frac{\pi}{4}\right)} + x_{rel}$
	Y	$L \cdot \frac{\sin\left(\frac{\pi}{4} + \theta\right)}{\sin\left(\frac{\pi}{4}\right)} + y_{rel}$
	Z	$0.0 + z_{rel}$
3	X	$L \cdot \cos(\theta) + x_{rel}$
	Y	$L \cdot \sin(\theta) + y_{rel}$
	Z	$0.0 + z_{rel}$
4	X	$0.0 + x_{rel}$
	Y	$0.0 + y_{rel}$
	Z	$0.0 + z_{rel}$
5	X	$(d + L\cos(\varphi)) \cdot \cos\left(\frac{\pi}{2} + \theta\right) + x_{rel}$
	Y	$(d + L\cos(\varphi)) \cdot \sin\left(\frac{\pi}{2} + \theta\right) + y_{rel}$
	Z	$h + L \cdot \sin(\varphi) + z_{rel}$
6	X	$(d + L \cdot \cos(\varphi)) \cdot \cos\left(\frac{\pi}{2} + \theta\right) + L \cdot \cos(\theta) + x_{rel}$
	Y	$(d + L \cdot \cos(\varphi)) \cdot \sin\left(\frac{\pi}{2} + \theta\right) + L \cdot \sin(\theta) + y_{rel}$
	Z	$h + L \cdot \sin(\varphi) + z_{rel}$
7	X	$d \cdot \cos\left(\frac{\pi}{2} + \theta\right) + L \cdot \cos(\theta) + x_{rel}$
	Y	$d \cdot \sin\left(\frac{\pi}{2} + \theta\right) + L \cdot \sin(\theta) + y_{rel}$
	Z	$h + z_{rel}$
8	X	$d \cdot \cos\left(\frac{\pi}{2} + \theta\right) + x_{rel}$

	Y	$d \cdot \sin\left(\frac{\pi}{2} + \theta\right) + y_{rel}$
	Z	$h + z_{rel}$

After this introduction on the calibration tool, the flowchart of the method is shown in Figure 36.

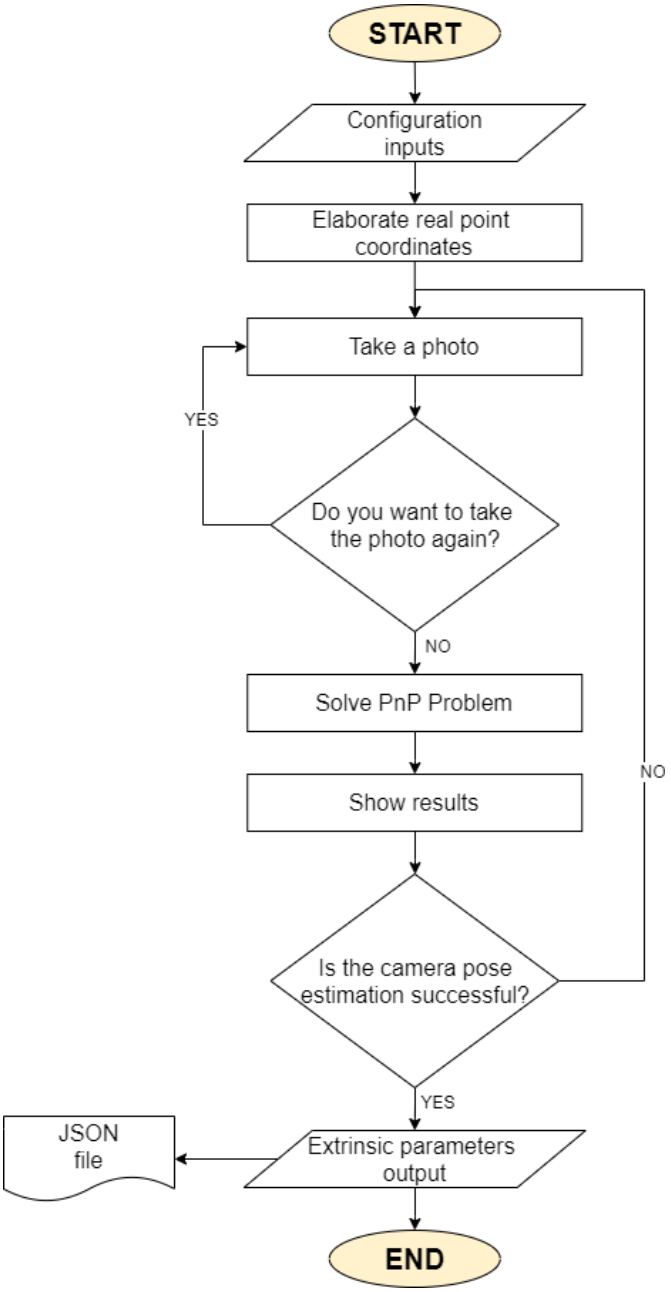


Figure 36: calibExtrinsicParam method flowchart

The structure of the function is analogous to the previous one. It takes as input from an external file the information on the calibration tool and its relative position. The first information are inside the *config_Extrinsic_calibration.json* file (see Figure 37) and they are fixed as long as the tool is the same.

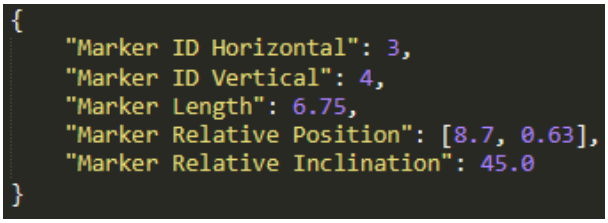


Figure 37: 3D calibration tool configuration

They are:

- The **ArUco ID** used, with reference to Figure 34.
- The **length of the marker side** in centimetres.
- The vertical **marker relative position**, with respect to the lower left corner. This information corresponds to *d* and *h* in Table 2.
- The vertical **marker relative inclination** expressed in degrees. It is the φ in Figure 35.

Its relative position with respect to the desired reference system is contained in the software general configuration file. In this way the physical tool and its applications in different cases are decoupled.

After processing the position of the ArUco marker corners of the calibration tool, it opens the camera and then allows to take some photos to be processed. In each photo the ArUco markers, whose IDs correspond to those entered in the initial configuration, are searched. Once the markers are found, it is possible to extract the pixel coordinates of the 8 corners. Then, since the real coordinates of the corners are known (evaluated at the beginning of the method with the formulas in Table 2), the PnP problem can be solved (see 3.5) to determine the camera pose with respect to the RS.

After the processing of all the photos, if the results are considered correct by the user, it saves the rotation matrix and the translation vector evaluated in the external camera file and as attribute of the camera class.

The accuracy, with which the camera poses is performed, is crucial for the results obtained. For this reason, it is advisable to calibrate the camera several times, until the values obtained can be considered truly satisfactory.

It is possible to evaluate *R* and *T* from a single frame. For greater accuracy it is advisable to use at least 5 photos, so that the program makes an average, reducing any evaluation errors.

4.3 ONECAMERA CLASS

The *classOneCamera.py* is a class object defined with to purpose to test one camera. It is not the main program to execute, but it helps to manage all the functions of the program, without interfering with the main architecture, that will be explained later. The architecture of *classOneCamera* is shown in Figure 38:

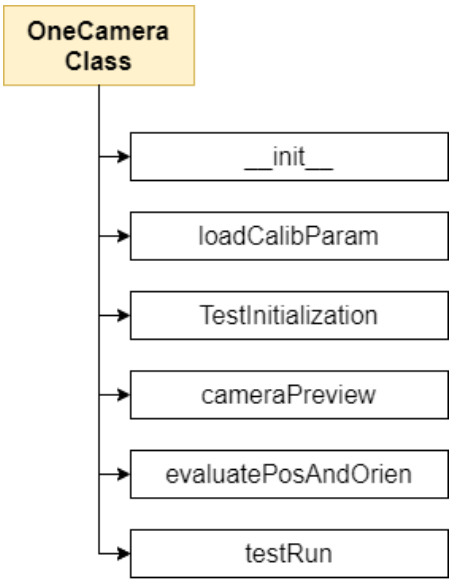


Figure 38: OneCamera class methods

4.3.1 Initialization method

The class is initialized with the *__init__* method. The use of the name *"__init__"* is attributed to the Python syntax, and for simplicity has been reported the same in this document. During the first step it loads the configuration options chosen for the test to be performed on a single camera. The concept is to manage the program setting from an external file, that is easy for a non-expert user to read. In this way, an operator shall not open the software code to edit some parameters or to active/disable some functionality

(e.g. save the test video). The options are contained in an external configuration file named *config_init_OneCamera.json*, as shown in Figure 39.


```
{
 "Camera Name": "microsoft_1",
 "Camera ID": 1,
 "Camera Width": 1280,
 "Camera Height": 720,
 "Marker Length [cm]": 5.0,
 "Z [cm]": 0.3,
 "OBJ Marker ID": 0,
 "OBJ Real Center": [2.5, 2.5],
 "True distance [cm]": 50.0,
 "True orientation [deg]": 0.0,
 "Tolerance": 1e-6,
 "Multiple tests": false,
 "Calibrate intrinsic parameters": false,
 "Calibrate extrinsic parameters": true,
 "Reference System Relative Position [cm]": [-1.5, 31.5, 0.56],
 "Reference System Relative Orientation [deg]": 0.0,
 "Resize window": true,
 "Window Name": "Acquisition",
 "Use undistortion": true,
 "Save test videos": false,
 "Seconds between results": 0.5,
 "Number of measurements": null
}
```


Figure 39: OneCamera initial configuration

The options are:

- **Camera information**: name, ID, and resolution (width and height)
- **Object marker information**: side length, ArUco ID, real centre of the object. The "Z [cm]" and "OBJ Real Centre" allow to place the marker on the object with a displacement with respect to the real point to be measured (i.e. the mass centre). LocSys evaluates the position of corner number 3 (with reference to Figure 15). The example above allows to evaluate the position of the marker centre. Having the sides 5 [cm] long, the centre is translated by 2,5 [cm] along x and along y with respect to the corner. The Z coordinate allows to consider the thickness of the real marker.
- **True distance and orientation** to compare with the ones evaluated by the software. This information is needed for the user interface created for the tests.
- **Relative position and orientation** for the extrinsic calibration tool, allowing to consider some physical limits for the process. The *z* relative coordinate comprises the thickness of the tool. In this way the reference system lies on the main surface.

- **Number of measurements** to save during the test and the **elapsed time** between measurements. If the “Number of measurements” is *null*, it saves the results until the user stops the test. If the “Seconds between results” is *null*, it saves the results of each frame that is processed. This concept will be taken up when software profiling is discussed.
- **Flags**: it is possible to set some parameters as *true* or *false*. It allows to perform multiple test, to calibrate the camera or to save a video of the test.

After loading all these parameters, they are saved as attribute of the class, to be used by the other methods. Finally, it initializes the camera class.

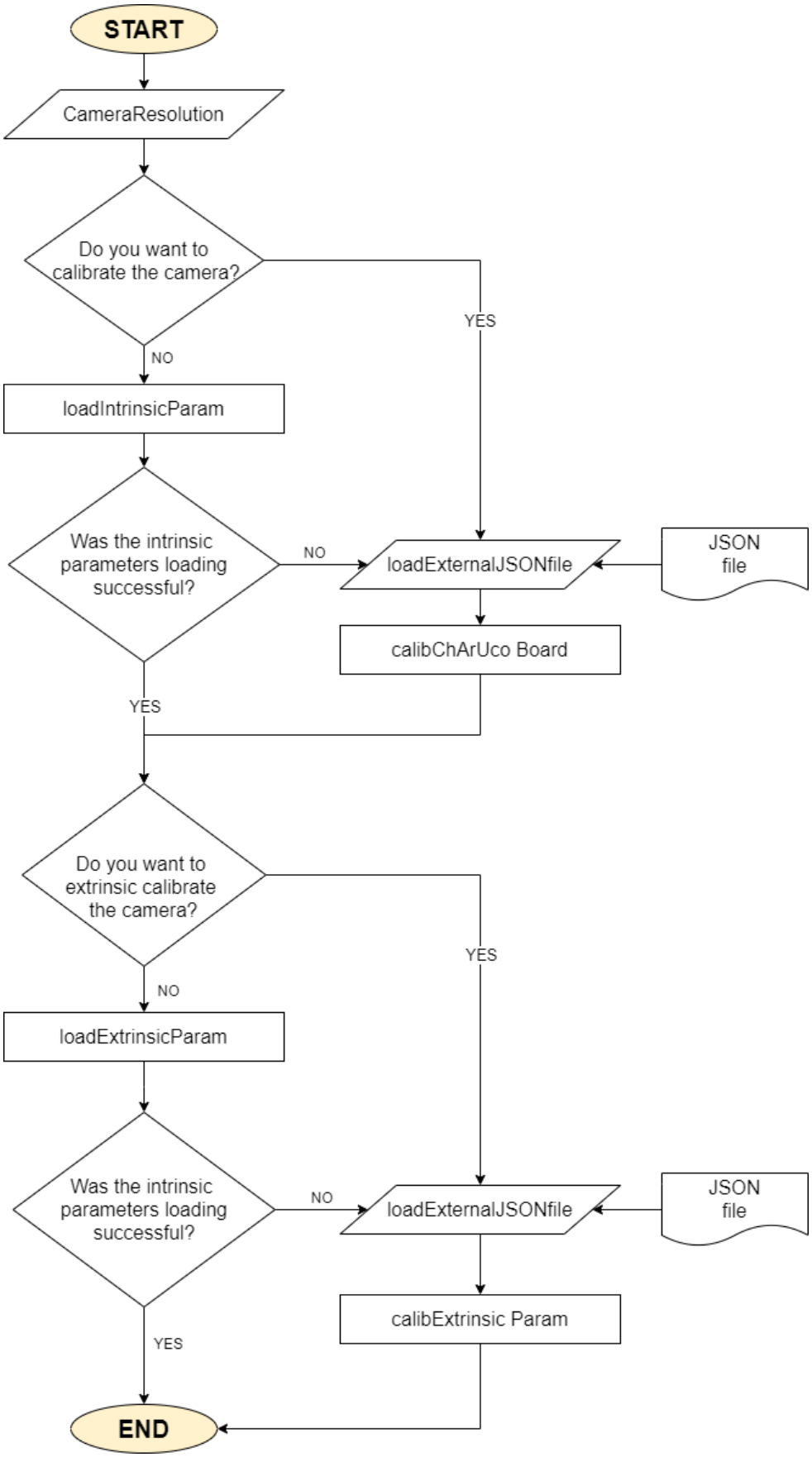


Figure 40: loadCalibParam OneCamera method flowchart

4.3.2 *loadCalibParam* method

The flowchart of the method is shown in the Figure 40. The function takes as input the resolution chosen for the test. If the user chooses not to calibrate the camera, it loads the intrinsic parameters from the external file relating to the camera, using the previously descripted camera class method (see 4.2.2).

If the parameters are not found or the user chooses to calibrate the camera, it loads the calibration options from the external file named *config_ChArUco_calibration.json*, and it calls the camera class method for intrinsic calibration (see 4.2.3). The same procedure is then repeated for the camera pose.

4.3.3 *testInitialization* and *cameraPreview* methods

The first method creates some useful variable to use during the test. It firstly askes to the user the name of the test that is going to be performed, then it initializes the output results file and finally opens the camera video flow. The output results file has the form *'test name' .dat* (e.g. *test_1.dat* or *test_2m_15deg.dat*) and the Figure 41 shows the information that it contains.

```
Test date and start time: 2020-01-30 10:58:53
Camera: microsoft_1  with a resolution of (w*h): 1280 x 720
True Distance: 50.0 [cm]    MarkerLenght: 5.0
--- Camera properties ---
Exposure: -6.0
Brightness: 110.0
Contrast: 10.0
--- Camera Relative Position [cm] ---
X: -24.867  Y: 14.879   Z: 72.072
--- Camera Relative Orientation [deg] ---
X: 176.538  Y: 1.890    Z: 0.587

Num D[cm]   AKE[mm] Th[°]   AKE[°]  time[s] Rel-X   Rel-Y   Rel-Z
1   50.48   4.80    31.57   1.57    0.501   39.87   30.96   0.30
2   50.46   4.56    31.64   1.64    0.525   39.86   30.94   0.30
```

Figure 41: Output .dat file example

The *cameraPreview* method checks if the camera is opened correctly and then shows on the screen what the camera sees. In this way, it is possible to check before the beginning of the test the exact pointing of the camera and the reference system adopted, which is drawn on the screen.

4.3.4 *evaluatePosAndOrien* method

The *evaluatePosAndOrien* is the main method of this class, as it is the function that takes a frame as input and evaluates both position and orientation of the marker and therefore of the object. The composition of this method is simple and linear, therefore a flowchart will not be reported.

The first step is the image undistortion using the camera class method. After removing the distortion, the program search for any ArUco marker in the image. If an ArUco marker is found whose ID corresponds to that of the object chosen from the initial configuration, then the code proceeds with the evaluations, otherwise it ends.

The marker bottom left corner (see 3.3) is projected into 3D space using the formula (3.2). To check if the points is projected correctly, the only condition that is known a priori is used. If the z coordinate of the 3D point is less than the true one plus a given tolerance, the projection is correct, and then the orientation is evaluated. The true z coordinate and the tolerance are taken from the initial configuration.

The orientation is evaluated using the corner number 2 and 3, using the following formulas:

$$dx = x_{corner2} - x_{corner3}$$

$$dy = y_{corner2} - y_{corner3}$$

$$\theta = arctg\left(\frac{dy}{dx}\right)$$

The orientation value is therefore between $-\pi < \theta < \pi$.

Next, the relative position of the centre (x_{rel}, y_{rel}) of the object with respect to the marker corner (x_{corner}, y_{corner}) is added, taking into account the orientation θ as follow:

$$x_{OBJ} = x_{corner} + x_{rel} \cdot \frac{\cos\left(\frac{\pi}{4} + \theta\right)}{\cos\left(\frac{\pi}{4}\right)}$$

$$y_{OBJ} = y_{corner} + y_{rel} \cdot \frac{\sin\left(\frac{\pi}{4} + \theta\right)}{\sin\left(\frac{\pi}{4}\right)}$$

The results are then saved, considering the time elapsed between the last measurement and the maximum measurements desired.

4.3.5 *testRun* method

The *testRun* method is the method that make the software work. Its flowchart is shown is Figure 42.

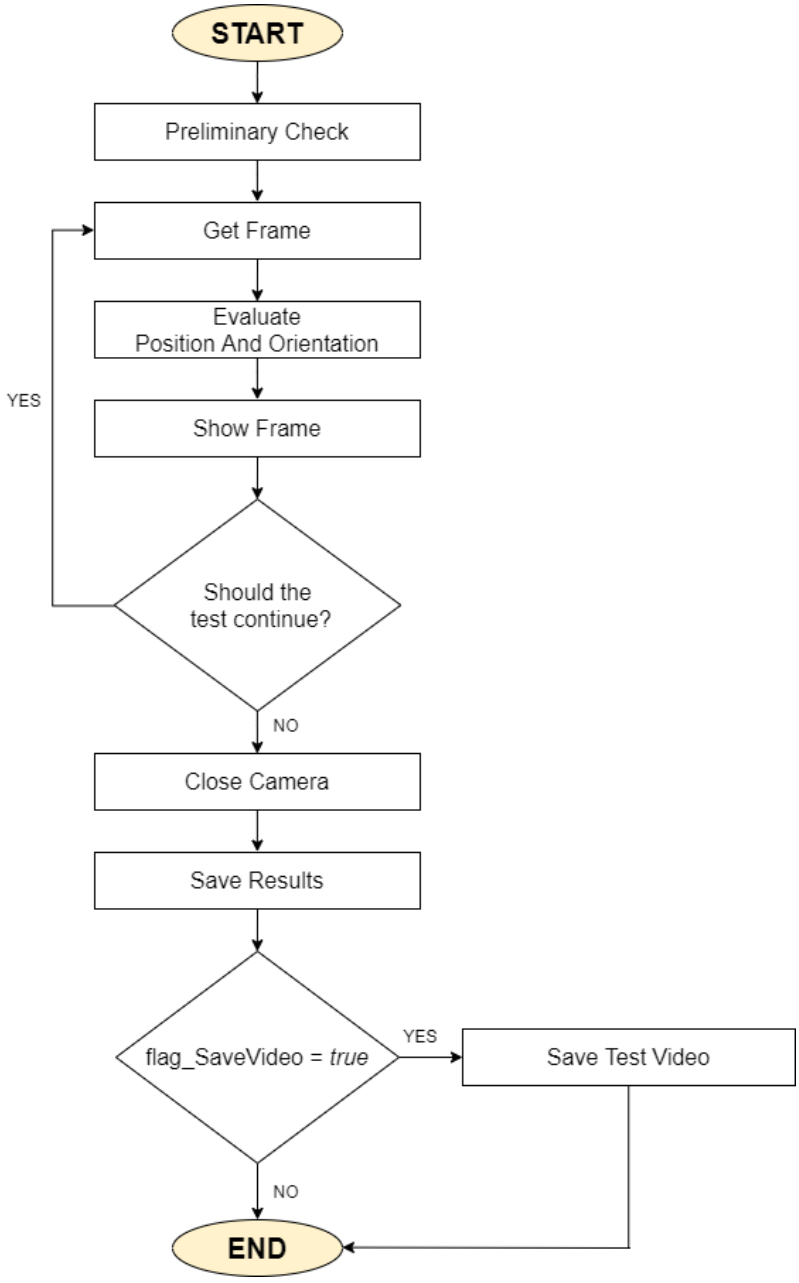


Figure 42: testRun method flowchart

When the method is called in the main program, it starts checking if the camera is properly opened, otherwise it opens it. Next, it starts a loop in which at every step it takes a frame from the video data flow, and it evaluates the position and the orientation of the

marker according to the method described in section 4.3.4. If the number of measurements correctly taken is greater than the maximum chosen in the initial configuration file or the user chooses to stop the test, the method closes the camera and saves the results in the external *.dat* file. Finally, if in the initial configuration the "Save test videos" flag is set to *true*, it saves a video of the test carried out, otherwise the method ends.

Now, all the methods that allow the main program to work have been presented, therefore in the next section the logical flow of the *LocSys_OneCamera.py* program for a single camera will be described.

4.4 *LocSys_OneCamera* Main Program

The *LocSys_OneCamera.py* program is the main software to be run, in order to test the performances of the localization system using one single camera. The flowchart of the software is shown in Figure 43.

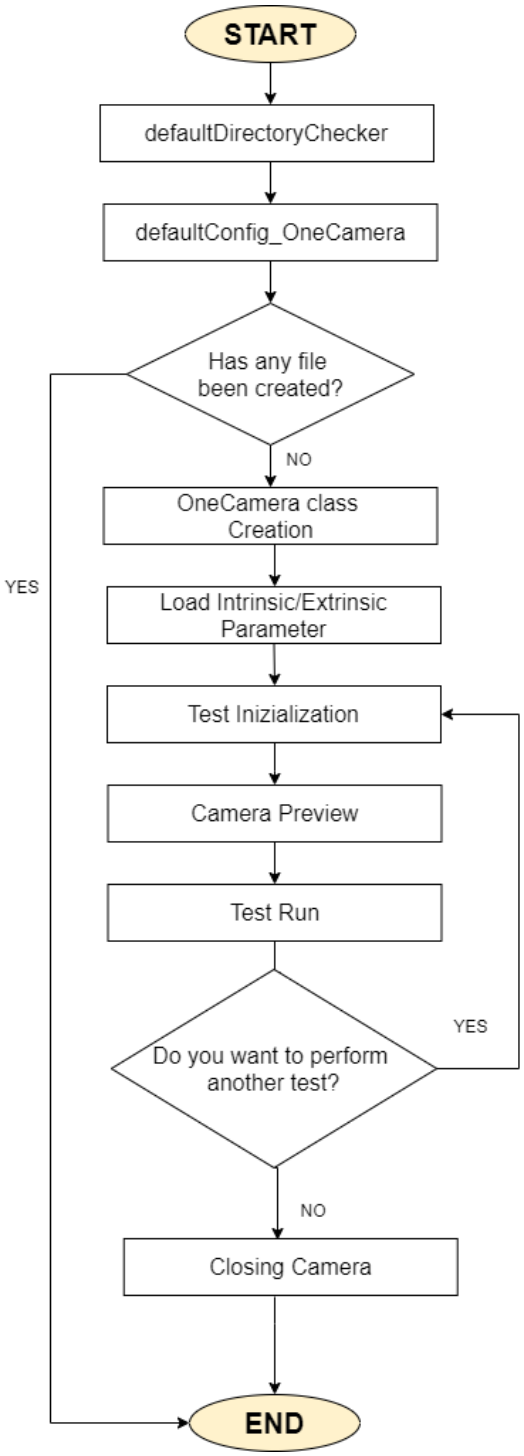


Figure 43: LocSys_OneCamera flowchart

When the program is launched, it firstly checks the existence of the folders necessary for the correct execution of the program. In case they were deleted by mistake, it creates them. The same process is done for the configuration files, including those for calibrations. The created files will have default settings which must be changed before performing a new test.

After this first step, it creates the camera class and then load its intrinsic and extrinsic parameters if they already exist, otherwise it will execute the calibration (as said in section 4.3.2).

The main program is now ready to launch the *testInitialization* method, the *cameraPreview* method and subsequently the *testRun* method that will execute the test. When the test ended in the manner described above, if the "Multiple tests" option is set to *true*, the program asks the user if he wants to perform another test. Otherwise the camera is closed, and the program ends.

As it can be notice, the *LocSys_OneCamera* program only implements the *classOneCamera* methods. In this way, it is possible to modify some functions without going to perturb the structure of the main program.

Hence, to run the program, simply enter the desired settings in the respective files and execute the *LocSys_OneCamera.py* file.

4.5 MULTICAMERA CLASS

In section 3.6, the reasons why a multi-camera architecture is needed were introduced. Therefore, it was necessary to create a different class with respect to *classOneCamera.py* to handle more than one camera. Furthermore, it is necessary that the measurements evaluated by the cameras correspond to the same time instant. The cameras must be synchronized.

The *classMultiCamera.py* has a similar structure to the *OneCamera* class, but it implements the multithreading approach, to synchronize the camera frame acquisition.

The class purpose is to test the multi-camera architecture in order to demonstrate the feasibility of this kind of localization system, using low cost webcam with ArUco fiducial marker. It is not the main program to execute, but it helps manage all the functions, without interfering with the main software. The *classMultiCamera* methods are shown in Figure 44:

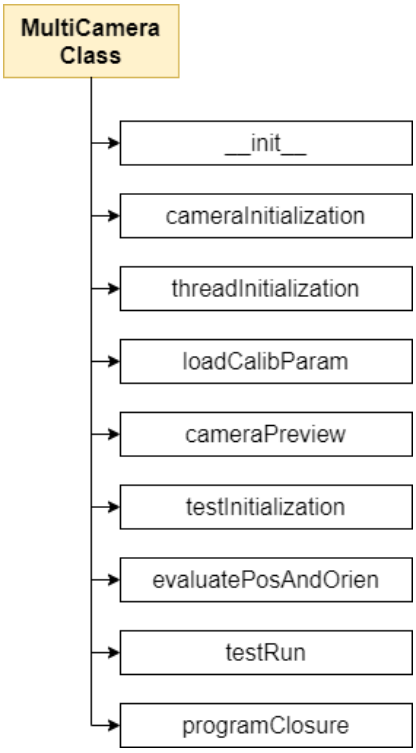


Figure 44: MultiCamera class methods

4.5.1 Initialization method

The concept of the *__init__* method is the same of the *OneCamera* class. The settings contained in *config_init_MultiCameras.json* file are reported in Figure 45:

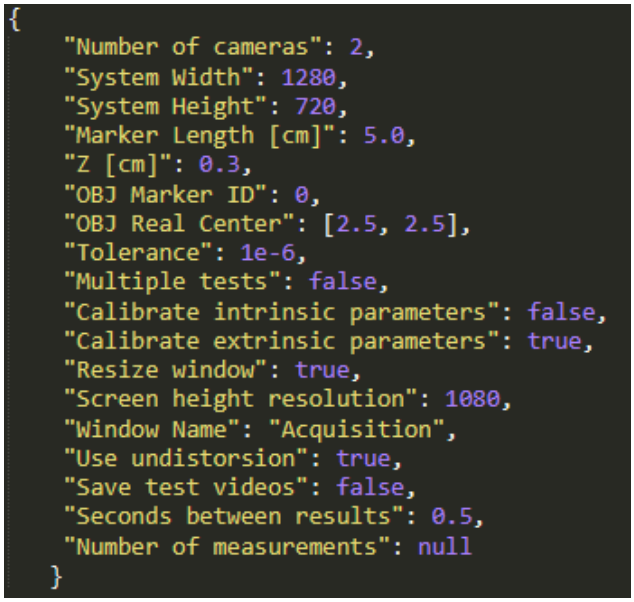


Figure 45: MultiCamera initial configuration

It is easily noted that the number of settings is less than the single camera architecture. The cameras related options are contained in another file, so to not complicate the readability of the individual files. The *MultiCamera* settings are:

- **Number of cameras**: the software is developed to manage an indefinite number of cameras. Furthermore, this number allows a correct generation of the video capture screen
- **System resolution**: set the same resolution to all the cameras. If a camera does not support that resolution, it will set a similar one (e.g. 1280x720 is like 1280x960). For the software to work properly, all cameras must have the same resolution.
- **Object marker information**: side length, ArUco ID and real centre of the object. These options are the same as *OneCamera* architecture.
- **Number of measurements** to save during the test and the **elapsed time** between measurements.
- **Flags**: allow to set some parameters as *true* or *false*.

All the settings loaded are then saved as attributes of the class.

4.5.2 *cameraInitialization* method

Unlike the single architecture, where only one class was created to manage the camera, in this there are more the one camera. This method creates a list of class in which every single element is one specific camera. The camera information is contained in the *config_cameras_list,json* file. An example of the external file is shown in Figure 46

```
{
    "Camera 1": [2, "microsoft_1", 50.0, 50.0, 0.0, 0.0, 0.0, 0.0, 0.56],
    "Camera 2": [1, "microsoft_2", 160.0, 50.0, 0.0, 180.0, 0.0, 0.0, 0.56],
    "Camera 3": [3, "logitech", 0.0, 49.7, -12.25, 0.0, 0.0, 0.0, 0.0]
}
```

Figure 46: Cameras list configuration file example

The *.json* files do not allow the insertion of comments, therefore the order of the information to be respected is shown in Table 3 (the item numbering refers to the indexes of the vectors used in Python):

Table 3: Cameras list item description

Item number	Description	Example
0	Camera ID	2
1	Camera Name	"microsoft_1"
2	Camera RS relative position [cm] – X_{RS}	150.0
3	Camera RS relative position [cm] – Y_{RS}	0.0
4	Camera RS relative position [cm] - Z_{RS}	0.0
5	Extrinsic calibration tool relative orientation [deg] - θ	90.0
6	Extrinsic calibration tool relative position [cm] – x_{tool}	5.0
7	Extrinsic calibration tool relative position [cm] – y_{tool}	5.0

8	Extrinsic calibration tool relative position [cm] - z_{tool}	0.5

The first two item are well-explained in section 4.2.1. The goal of the system is to evaluate the position with respect to a reference system, such as a corner of the table on which the TowerSat moves. In multi-camera architecture, the operating region of the system is expanded, therefore not all the cameras have the Master Reference System (MRS) in their FOV. For this reason, in this architecture each camera has its own RS that is related with the master reference system in term of relative position (item 2,3 and 4), as shown in the left part of Figure 47.

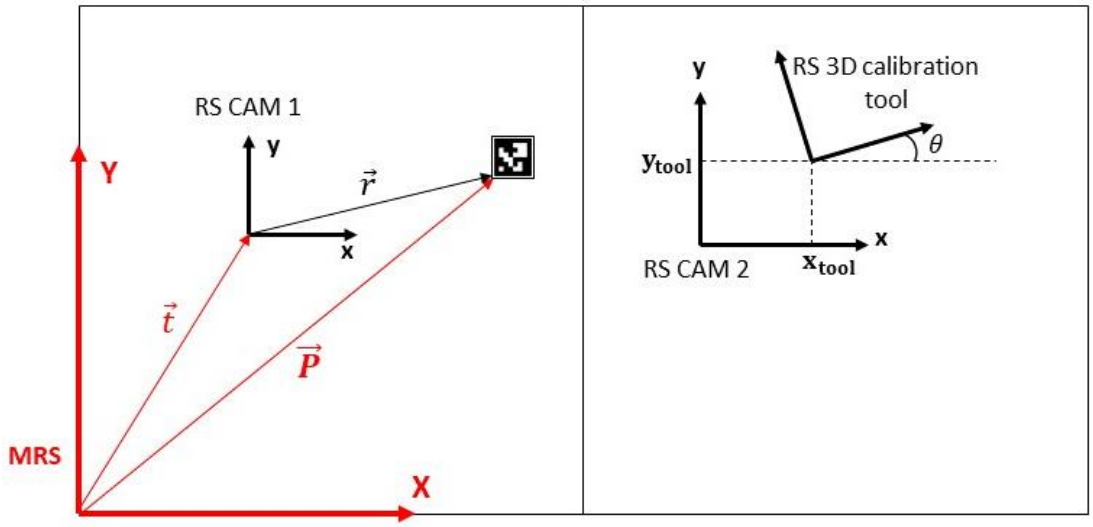


Figure 47: MultiCamera reference systems relationship

A camera allows to evaluate the position $\vec{r} = (x, y, z)$ with respect to its RS. The position of the RS in the master coordinate frame is represented by the vector $\vec{t} = (X_{RS}, Y_{RS}, Z_{RS})$. So, the position of the object in the MRS, represented by the vector $\vec{P} = (X, Y, Z)$ is:

$$\vec{P} = \vec{t} + \vec{r}$$

The items from 5 to 8 are the information related to the extrinsic calibration tool, as explained in section 4.2.4 and shown in the right part of the Figure 47. In this architecture, the need to have the tool rotated with respect to the desired reference system is clearer. Think about the case in which the camera is located on the opposite side of the laboratory, seeing the table, and therefore the calibration tool, from behind.

4.5.3 *threadInitialization* method

Concerning the cameras synchronization, the need for multiple threads has already been described. The *threadInitialization* method creates all the necessary threads, one for each camera, using the initial setting “Number of cameras”. It stores the thread object addresses in a vector, to recall them in the next steps.

In each thread, frames are continuously acquired by the camera and saved as an attribute of the thread. When a frame is requested from a thread, the last acquired is provided. In this way the cameras continue to work, saving the last frame from their data flow and the process of acquiring all the frames by all the threads is very fast because it only goes to recall the already saved attributes.

4.5.4 *loadCalibParam* method

The method has the purpose of loading the parameters of all the connected cameras. Its functioning is completely analogous to that implemented in *classOneCamera.py*. The difference lies in cycling within the vector that contains the classes of the cameras. In fact, the parameters of one camera at a time are loaded. The flowchart of the method is shown in Figure 48.

4.5.5 *testInitialization* and *cameraPreview* methods

The *testInitialization* method checks the correct creation of the threads and the capture opening of all the cameras. It also initializes the output results file, asking to the user the *“test name”*.

The *cameraPreview* allows to control the correct pointing of al the cameras and implements some elaborate steps for the management of the acquisition window, considering the number *N* of the cameras used.

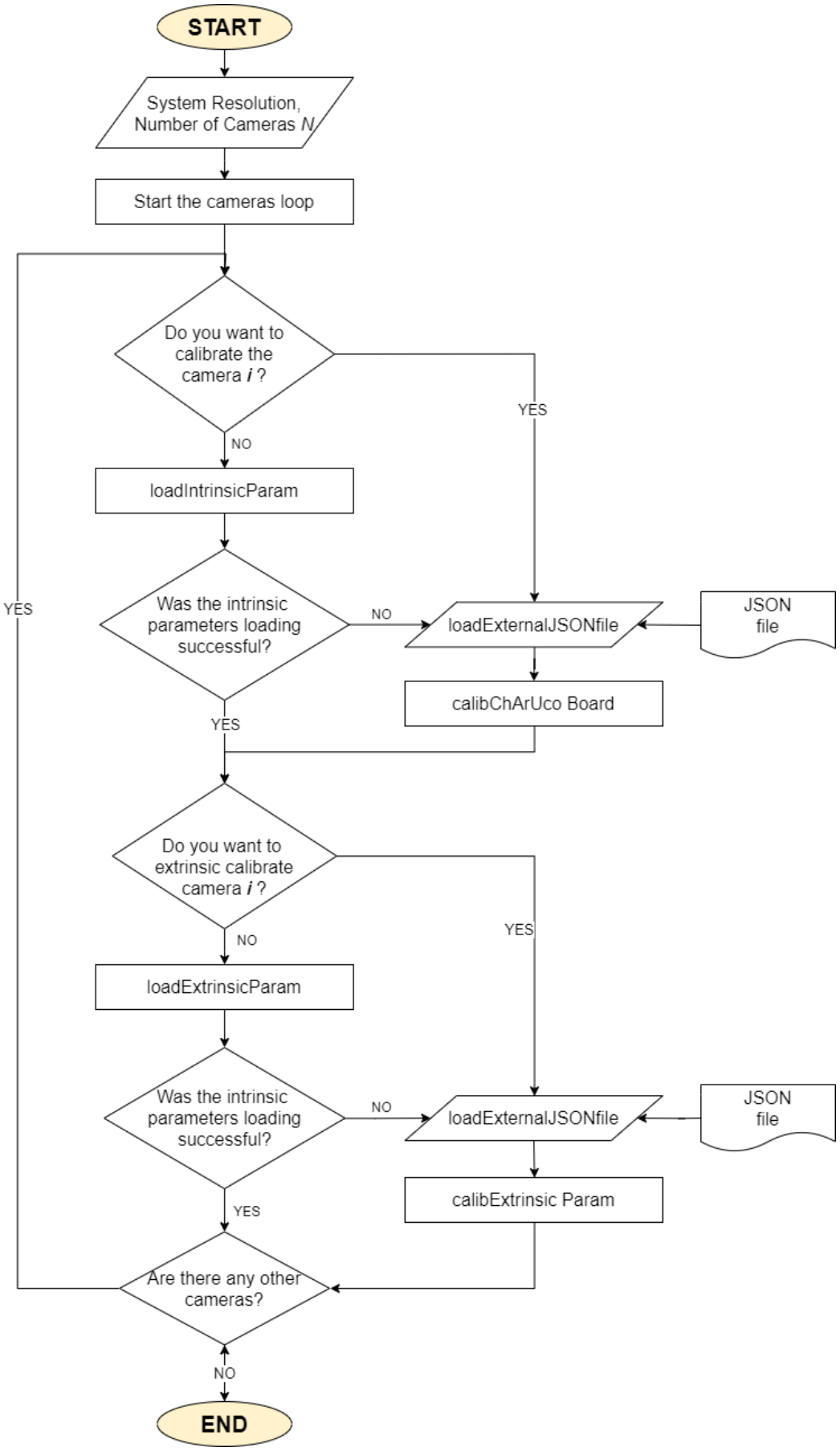


Figure 48: loadCalibParam MultiCamera method flowchart

4.5.6 *eveluatePosAndOrien* method

The method takes as input a single frame and the vector index of the reference camera from which it comes. The frame is analysed in the same procedure as that implemented in the *OneCamera* architecture. The complexity derives from the use of the indexes to refer to the camera parameters.

4.5.7 *programClosure* method

The *programClosure* function stops all the threads and closes the video flow of the cameras. It is executed only when the test is finished.

4.5.8 *testRun* method

This function implements all the other methods to make the software work. The *testRun* flowchart is reported in Figure 49.

It starts checking again the opening of cameras and if the threads are alive, to make sure the program can run properly. Then, the loop, that elaborate all the frames, starts. The first step concerns the acquisition of all the frames taken almost simultaneously. It has been repeated several times, but the synchronicity between the cameras is crucial to the reliability of the results, especially if the test object is in an overlapping area of the camera FOVs.

Subsequently, the frames are processed one at a time. This represents one limit of the multi-camera software. As the number of cameras increase, this step taker more time, decreasing the number of possible measures per second. One possible solution is to redefine the threads, so that in addition to acquiring the frames, they can process them. The difficulty of this solution is to maintain synchronism between the cameras.

Regarding the processed frames, if the object ArUco marker is found, a counter is updated. The counter will be greater than one only if the marker is viewed from multiple cameras. Then, the results are averaged, reducing the relative error.

The frames are then joined and shown on the screen, while the results obtained are saved. The test ends for the same conditions adopted in the other architecture. So, the *programClosure* is executed and the results are saved in the external *.dat* file.

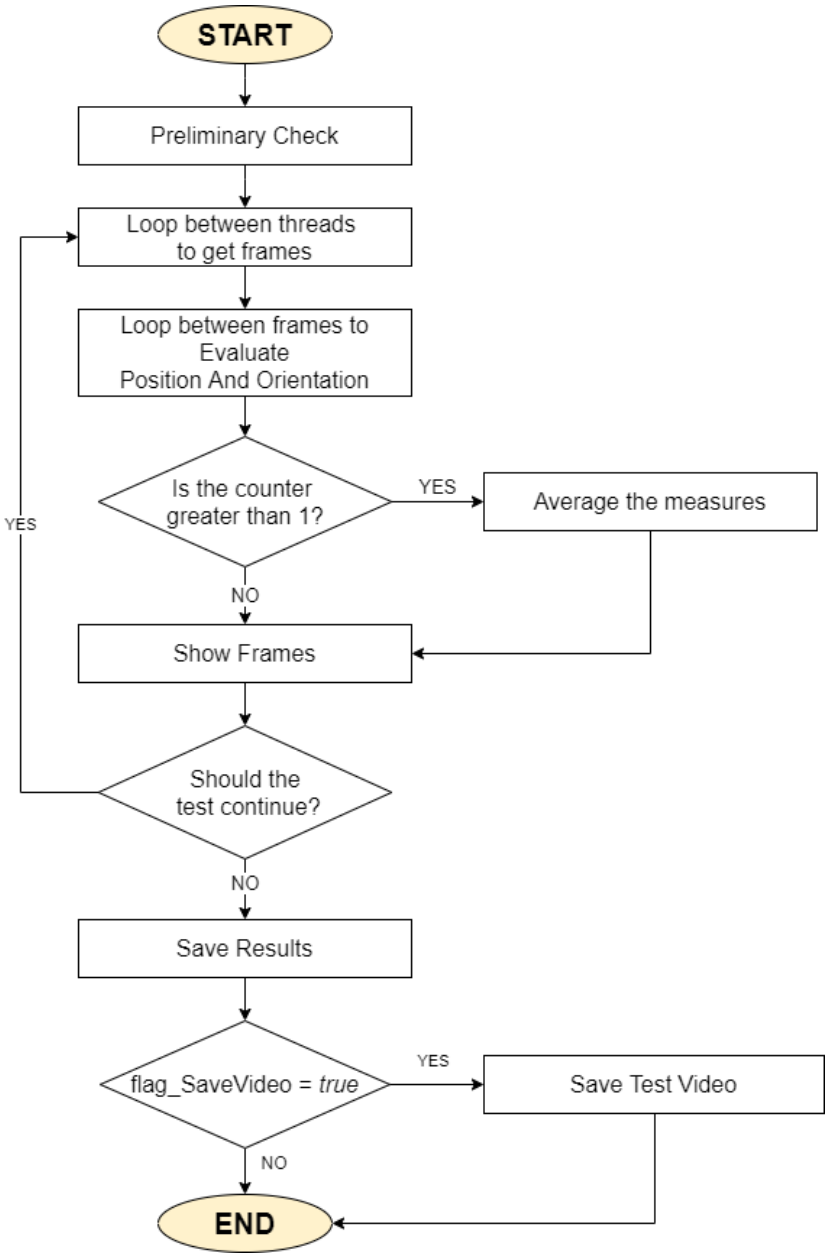


Figure 49: testRun MultiCamera method flowchart

Similarly to the one camera software, all the methods presented in this section are implemented in the *LocSys_MultiCamera.py*.

4.6 *LocSys_MultiCamera* Main Program

The *LocSys_MultiCamera.py* program is the main software to be run. It allows to test the functionalities of a multiple camera architecture, using *N* cameras. In fact, the software is not developed to handle a defined number of cameras, but it can manage as many cameras as the user want to. The flowchart of the software is shown in Figure 50.

The program firstly checks if the necessary folders and initial configuration files exist. In case it is impossible to find them, they are created, and the program is stopped. The created files are in a default version, so before continuing with the test, the settings must be changed.

Next, the *MultiCamera* class is defined and the external configurations are loaded by the initialization method of the class. After the initialization of the cameras, where *N classCamera* are created, the program loads the intrinsic and extrinsic parameters or launches the relative calibration if the configuration is set to *true*. Only after performing these steps, the *threadInitialization* method is called.

The central loop is the same as the one for *LocSys_OneCamera.py*, in fact it starts with the test initialization and it ends asking to user to perform another test. At the end of the program, the threads are killed, and the cameras are closed.

The *LocSys_MultiCamera* program implements the methods of its relative class. After analysing all the classes and methods that allow the system to function, the performance will be described in the next chapter, reporting all the tests carried out with the relative results.

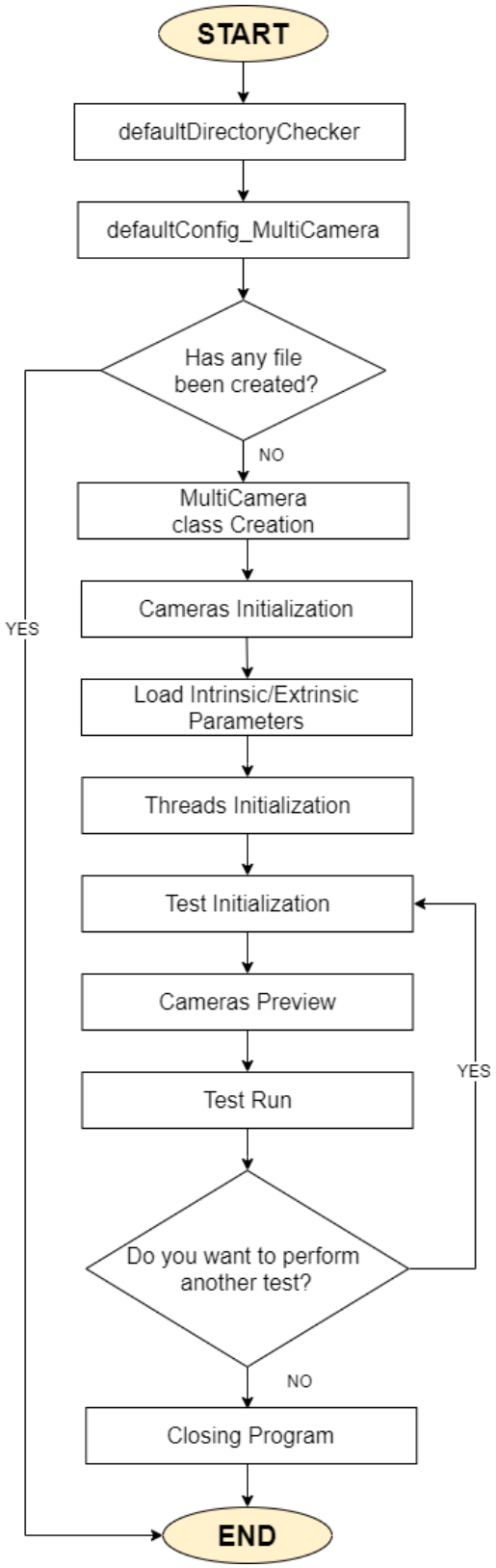


Figure 50: LocSys_MultiCamera flowchart

5 LocSys: Tests and Results

The following chapter describes the tests carried out to determine the performance of a localization system based on ArUco markers.

First, the performance of the software itself will be described, starting from debugging and code profiling. Then, it will be described the tests carried out on architecture with a single camera, used to determine the limits of the camera associated with the markers. Finally, tests on multi-camera architecture and how it should be developed will be described.

5.1 Software Performance

The localization system described so far is managed by a customized software. Before testing the system performance, the software itself had to be tested.

5.1.1 Software Debugging

The first test takes place via the debugging process and it involves locating and fixing the errors discovered in the code. The debugging occurs throughout the development of the software, searching for syntax and sematic errors that create incorrect output or do not allow the complete operation of the code.

The Figure 51 shows the usual steps of the debugging process:

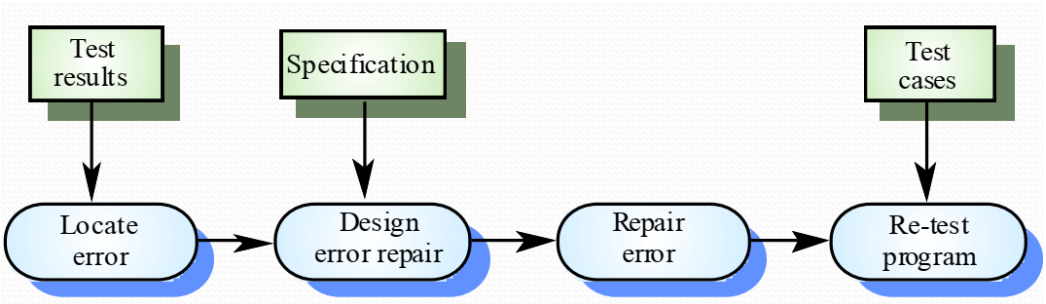


Figure 51: Debugging process

During the software debugging it is necessary to formulate some hypotheses about the program behaviour. After this step, it is possible to test these hypotheses to find the errors in the code.

The length and complexity of this process depends on the skills of the programmer, although the compiler suggests some of the errors, especially the syntax ones.

5.1.2 Software Profiling

After the physiological debugging phase, before testing the cameras architecture, the test concerning the software performances must be carried out. The software profiling means monitor the execution times of the various class methods and track the entire software path to control the presence of errors.

In Python, all these characteristics can be evaluated using its profiler. A profiler is a program that describes the run time performance of a code, using statistics and graphs. It also provides some useful tools to examine the results of a profile operation. The python profiler used is *cProfile*, that is a C extension suitable for profiling long running programs.

In this application case, it is necessary to evaluate how long the *evaluatePosAndOrien* method takes to extrapolate information from a frame. Knowing how many hundredths of a second it takes allows to understand how many measures per second can be obtained. The number of measures per second (MPS) is then compared with the FPS (Frames per second) of the camera.

```
         372555 function calls (368713 primitive calls) in 100.824 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 <string>:1(__new__)
        1    0.000    0.000  100.825  100.825 LocSys_OneCamera.py:1(<module>)
        1    0.001    0.001  100.480  100.480 LocSys_OneCamera.py:6(main)
        1    0.000    0.000    0.000    0.000 __config__.py:3(<module>)
        1    0.000    0.000    0.000    0.000 __future__.py:48(<module>)
        1    0.000    0.000    0.000    0.000 __future__.py:80(_Feature)
       10    0.000    0.000    0.000    0.000 __future__.py:81(__init__)
```

Figure 52: LocSys_OneCamera profiling output

If the MPS is greater than the FPS, it means that some frames are analysed more than once, providing repetitive measurements. One the contrary, if the MPS are lower than

the FPS, a minimum requirement arises of how many measurements are wanted to be obtained every second. The TowerSat will not have fast movements, in the order of 1 $[cm/s]$, therefore a minimum of 5 measures per second are required.

These performances were assessed by running the *LocSys_OneCamera.py* for about one minute, by saving one measurement every 0,5 seconds. A camera resolution of 1280 x 720 was used. This information is important because if affects the time required to parse each individual frame. The profiling output format is shown in Figure 52.

The first line indicates the number of calls monitored and the number of the *primitive* ones. The term *primitive* mean that a function call is not induced via recursion. The columns indicate:

- **ncalls:** function number of calls
- **tottime:** total time spent in the function (excluding the time made in call to sub-functions)
- **percall:** the tottime divided by ncalls
- **cumtime:** total time spent in the function, including sub-functions (from invocation till exit)
- **filename:lineno(function):** provide the respective data of each function

The Figure 53 shows the information relating to the methods of interest.

```
   1    0.000    0.000    0.054    0.054 classCamera.py:1(<module>)
   1    0.000    0.000    1.023    1.023 classCamera.py:107(setResolution)
   1    0.000    0.000    0.001    0.001 classCamera.py:129(loadIntrinsicParamJSON)
   1    0.000    0.000    0.000    0.000 classCamera.py:13(Camera)
   1    0.000    0.000    0.000    0.000 classCamera.py:160(loadExtrinsicParamJSON)
   1    0.000    0.000    0.000    0.000 classCamera.py:17(__init__)
2034    0.049    0.000   13.278    0.007 classCamera.py:198(imageUndistorsion)
   1    1.027    1.027    2.098    2.098 classCamera.py:27(open)
2034    0.009    0.000   31.144    0.015 classCamera.py:59(frame)
   1    0.000    0.000    0.026    0.026 classCamera.py:69(close)
   1    0.000    0.000    0.343    0.343 classOneCamera.py:1(<module>)
   1    0.001    0.001   14.026   14.026 classOneCamera.py:113(testInitialization)
   2    0.000    0.000    0.019    0.010 classOneCamera.py:138(windowCreation)
   1    0.830    0.830   22.767   22.767 classOneCamera.py:154(cameraPreview)
   1    0.000    0.000    0.000    0.000 classOneCamera.py:17(OneCamera)
1482    1.614    0.001   39.250    0.026 classOneCamera.py:201(evaluatePosAndOrien)
   1    0.000    0.000    0.001    0.001 classOneCamera.py:26(__init__)
   1    0.954    0.954   63.658   63.658 classOneCamera.py:297(testRun)
   1    0.000    0.000    0.001    0.001 classOneCamera.py:62(loadCalibParam)
```

Figure 53: classOneCamera profiling

The calibration functions were not called, but methods for loading the parameters from external files were called once. These functions take so little time to be around zero.

Concerning the calibration, both intrinsic and extrinsic, since there are no time constrains during this process, no profiling was performed.

During the test, 2034 frames were used. This is the number of times the *frame* method of the *classCamera* was called. Some of these frames refer to the *cameraPreview* phase, which lasted 22.767 [s].

The test carried out saved a total of 120 measures, but from the figure above it is known that the *evaluatePosAndOrien* method was called 1482 times. Therefore, some of the frames were analysed, but the results were not saved, or the marker was not recognized and so they did not provide any measures. The duration of the function is approximately 0.001 [s] , but considering the sub-functions the duration rises to 0.026 [s]. It means that this function, which takes a frame as input and gives the marker position and orientation as output, can provide 38.46 [MPS]. The camera used works at 30 [FPS]. The sub-functions called by the method, which take most of the time, are those assigned to manage the frame. They remove some noise and detect the ArUco marker.

Between the acquisition of a frame and the next, not only the *evaluatePosAndOrien* method is performed, but other operations are also carried out, which decrease the calculated MPS. For this reason, during the tests on the *OneCamera* architecture, it was chosen to save a measurement every 0.5 [s], to avoid the problem of MPS greater than the FPS. This situation certainly does not occur in the *multiCamera* architecture, as will be described later.

The total test duration, that is of the *testRun* method, is 63.658 [s]. The 3.658 [s] more than the minute comprise the initialization and the closure of the method. Furthermore, exactly 0.5 [s] do not pass between one save and the next.

Concerning the *multiCamera* architecture, the software has some differences, namely the presence of threads for the acquisition of frames from multiple cameras.

The time needed to analyse a frame is the same as in the previous test, as the function is identical. It should be remembered that this time depends on the image size, therefore on the camera resolution. For these reasons, it is necessary to evaluate how long it takes the software to recover the frames from the cameras. This timing depends on the number of cameras used and it is not affected by the resolution. In this case two cameras have been used. In addition, *cProfile* does not provide this information, therefore timers have been added within the software. The results are shown in Figure 54:

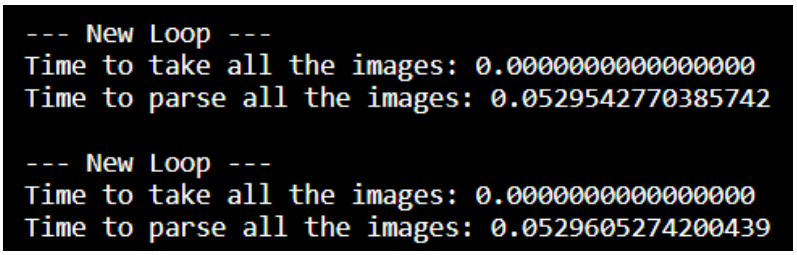


Figure 54: multiCamera frames acquisition timing

As mentioned in section 4.5.3, in each thread the frames are continuously acquired by the cameras and saved as their attribute. Within the *testRun* method, frames are retrieved from attributes. In this way, the acquisition process refers to the last frames acquired by all the cameras and it is so fast that it cannot be evaluated even with 15 decimal places. Thanks to the use of threads, the acquisition of frames from the cameras takes place simultaneously, ensuring synchronism.

Concerning the time to parse all the images, it confirms the one evaluated with a single camera architecture. As the number of cameras increases, this time increases more and more, significantly reducing the measures per second.

A solution to this problem is to adopt the multithreading approach also for the frames parsing. This method has not been implemented yet, as it complicates the synchronism between the cameras. It can certainly be a future development of the LocSys.

5.2 TEST REQUIREMENTS AND CONDITIONS

Before describing how the tests were conducted, it is necessary to explain the parameters adopted to evaluate the results and those that must be reproduced in order to replicate a test. In fact, for each test, requirements to be met and conditions that influence the test results have been defined.

Some requirements concern the performances desired by the system and define the success and failure of the test itself. In each test position and orientation are assessed and their true measurements are compared. Therefore, the results are evaluated in terms of precision and accuracy:

- **Accuracy**: it is expressed as the difference between the result and its true value. At the end of the test it is used the absolute knowledge error (AKE) as a parameter for passing the test. [15]
- **Precision**: it is expressed as the difference between the error of a measurement and the average of the errors of the test. It represents the ability to evaluate the same measure over time , despite the disturbances in the assessments. Precision is expressed in terms of relative knowledge error (RKE). [15]

Considering the accuracy obtained from other localization systems described above (section 2.2.2), for the position an $AKE = 20\ [mm]$ and an $RKE = 2\ [mm]$ have been chosen which determine the success of the test. As regards the orientation, a $AKE = 2\ [deg]$ and an $RKE = 0.5\ [deg]$ have been adopted.

Other requirements refer to what was used to perform the test, such as the ArUco marker ID and side length used to represent the object, the number and frequency of saving the results and the resolution adopted by the camera or cameras.

The Table 4 shows the requirements adopted for every test.

Table 4: Tests requirements

Test Requirements	
Marker for OBJ:	ArUco marker ID 0 from dict 6x6_250
MarkerLenght:	5.0 [cm]
N° of measurements:	120
Seconds between measurements	0.5 [s]
Camera resolution:	1280 x 720
Max Position AKE [mm]:	20.0
Max Position RKE [mm]:	±2.0
Max Orientation AKE [deg]:	2.0
Max Orientation RKE [deg]:	±0.5

In order to be able to replicate a test, it is not enough to comply with the requirements, as there are conditions that influence the results. These conditions are not always faithfully replicable, making a test unique.

In this application case, the conditions concern the computer used to run the LocSys software, the cameras and the ambient light conditions, also linked to the day and time the test is performed (for the contribution of light solar).

The computer performance influences the time spent in every method, especially the *evaluatePosAndOrien*. For cameras, it is necessary to consider their internal calibration, but above all their pose. In fact, as already mentioned, the extrinsic parameters calibration is decisive on the results obtainable by the software and is a process that requires a lot of precision from the operator. Although the operator may be able to place the calibration tool perfectly, the software will still make mistakes, making it impossible to replicate a test. For these reasons, the camera parameters can be considered as test conditions. Furthermore, the camera may have the ability to automatically vary some of its properties, such as the exposure, brightness and contrast of the frames acquired. Even if it is possible to set these properties, it is desirable to consider them as conditions of the test.

Concerning the multi-camera architecture, the conditions include the information of all the cameras, comprising their location in space to cover the whole operative area. An example will be reported in the next section, when the *LocSyS_MultiCamera* test will be described.

5.3 *LocSys_OneCamera* Test Sessions

The *LocSys_OneCamera* software has been developed specifically to test the performance achievable by this type of system. Using a low-cost web camera, the tests were conducted by varying the distance and angles, pointing the camera at the centre of the scene and keeping the reference system in a corner of the FOV. Tests were also performed by changing the light source in the environment, to understand the performance provided by the binary fiducial markers.

The steps followed during each test are represented by the simple flowcharts in the following figure:

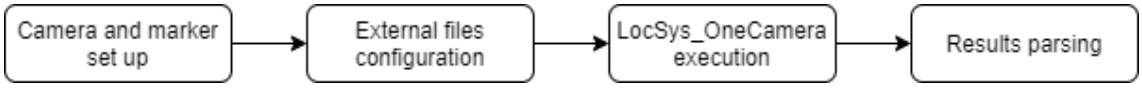


Figure 55: Test evaluation flowchart

To simultaneously evaluate the performance on the position and orientation determination, the ArUco marker representing the object was placed at 50 [cm] (40 [cm] long x and 30 [cm] long y) from the point chosen as the reference system. It is also rotated 30 degrees with respect to RS. The Figure 56 shows the OBJ marker at the top right and the calibration tool at the bottom left with the desired reference system drawn.

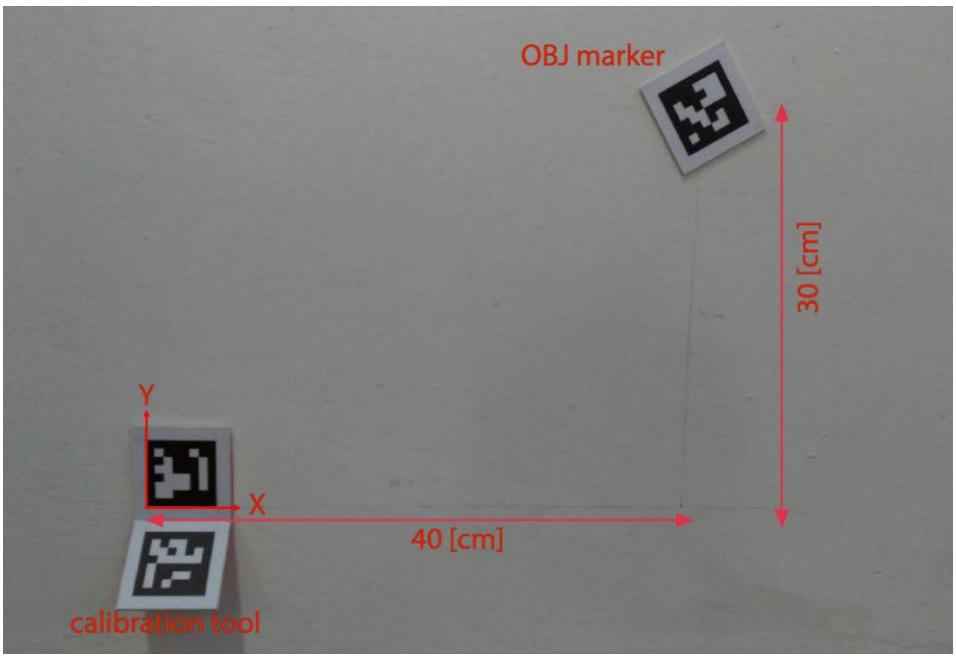


Figure 56: OBJ marker and RS for OneCamera test

Before running any tests, the settings in the external configuration files must be set. As anticipated in the previous sections, the camera calibration can be performed only once, therefore during the first test the setting “Calibrate intrinsic parameters” has been set to “true”.

For the intrinsic calibration, the pattern shown in Figure 32 was used, printed on forex in A3 format and made opaque, to reduce light reflections and improve marker recognition. The settings entered in *config_ChArUco_calibration.json* are:

Table 5: ChArUco calibration settings

squaresX	9
squaresY	7
squareLength	3.9
markerLength	2.4
min_valid_frame	10

Hence, the camera used and its calibration derived parameters are shown in Table 6, together with the reprojection error, an indicator of the calibration accuracy.

Table 6: Camera intrinsic parameters during tests

Camera Model:	Microsoft LifeCam HD-3000				
Camera Matrix:		1121.396 0.000 0.000	0.000 1119.820 0.000	663.541 359.443 1.000	
Distortion coefficient:	0.260	-1.950	0.000	-0.003	4.178
Reprojection Error	0.578				

Concerning the camera pose, having to test the operational limits of the system, the camera is moved with each test, therefore the pose estimation must be carried out every time with the external calibration tool. The physical characteristics of the developed tool are shown in the following table:

Table 7: 3D calibration tool settings

Marker ID Horizontal	1
Marker ID Vertical	2
Marker Length	5.0
Marker Relative Position	[6.7, 7.1]
Marker Relative Inclination	45.0

This information is then used in the *calibExtrinsicParam* method to evaluate the camera pose.

Finally, for all the tests, the following settings were provided to the *LocSys_OneCamera* software (to the *OneCamera class*).

Table 8: LocSys_OneCamera test settings

Camera Name	microsoft_1
Camera ID	1
Camera Width	1280
Camera Height	720
Marker Length [cm]	5.0
Z [cm]	0.3
OBJ Marker ID	0
OBJ Real Center	[2.5, 2.5]
True Distance [cm]	50.0
True Orientation [deg]	30.0
Tolerance	1e-6
Multiple tests	true
Calibrate intrinsic parameters	false
Calibrate extrinsic parameters	true
Reference System Relative Position [cm]	[5.0, 5.0, 0.56]
Reference System Relative Orientation	180

Resize window	true
Window Name	Acquisition
Use undistortion	true
Save test video	false
Seconds between results	0.5
Number of measurements	120

Two considerations must be done from this table. The reference system provided by the specific calibration tool is translated and rotated with regard to that shown in Figure 57, for two reasons: to demonstrate the ability to decouple the RS provided by the tool and the position of the tool with respect to the desired reference system; the other reason is linked to the camera FOV, in fact, in the configuration with $\theta = 0°$ the vertical marker could not be seen. The second consideration concern the results saving. The software can parse many frames every second, depending on the computer performances. However, it was chosen to save the results every 0,5 seconds for a total of 120 measurements, so each test has a duration of approximately 60 seconds.

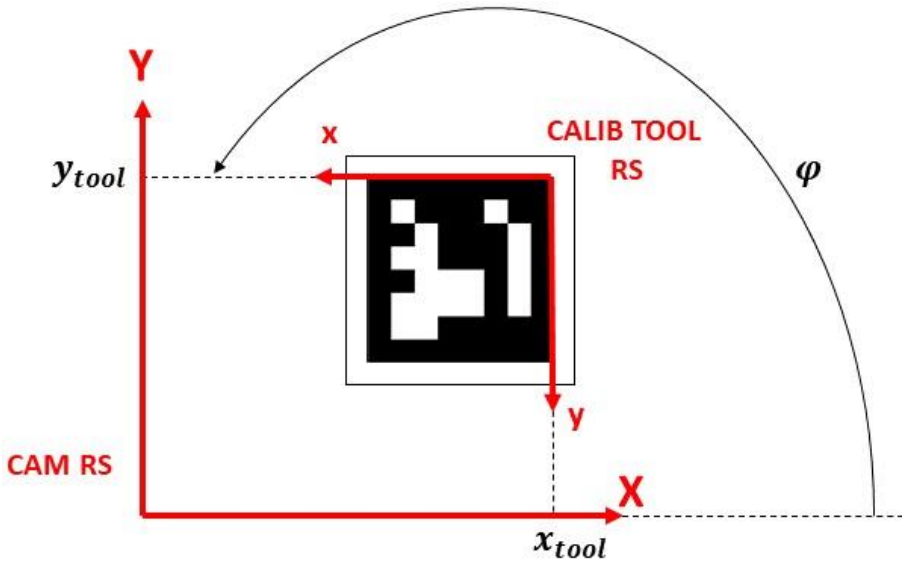


Figure 57: 3D calibration tool RS and camera RS relationship

At the end of the configuration phase, the tests can be performed. The tests carried out are summarized in the Table 9, where is reported the test reference code, a brief description and if the test is successful or failed. The success of the test depends on the satisfaction of the requirements that concern the AKE and RKE on both position and

orientation evaluations. The position of the camera is measured with respect to the desired RS provided by the calibration tool, as shown in Figure 58.

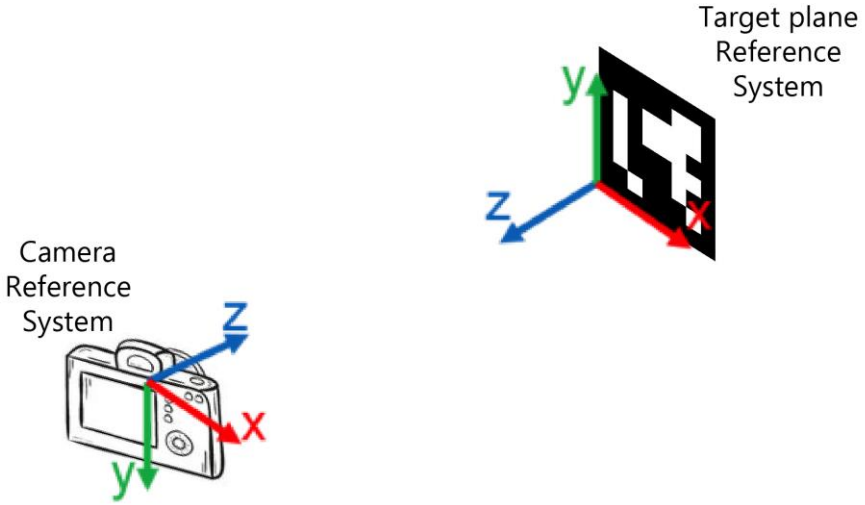


Figure 58: Camera and Marker reference systems

Table 9: OneCamera tests description

Test ID	Test Brief Description	Pass/Fail (P/F)
TS-01	Front view at a distance z of 75 cm, target centred in the image	P
TS-02	Front view at a distance z of 100 cm, target centred in the image	P
TS-03	Front view at a distance z of 150 cm, target centred in the image	P
TS-04	Front view at a distance z of 200 cm, target centred in the image	F
TS-05	Angled view of -30° with respect to Y, target centred in the image	P
TS-06	Angled view of -30° with respect to Y and -30° with respect to X, target centred in the image	P
TS-07	Angled view of -30° with respect to Y and +30° with respect to X, target centred in the image	P

TS-08	Angled view of -30° with respect to Y and +30° with respect to X, RS in a corner of the image	F
TS-09	Front view at a distance z of 150 cm, RS in a corner of the image	F
TS-10	Front view at a distance z of 150 cm, target centred in the image, lights off	P
TS-11	Front view at a distance z of 150 cm, RS in a corner of the image, lights off	F
TS-12	Angled view of -30° with respect to Y and -30° with respect to X, target centred in the image, lights off	F
TS-13	Front view at a distance z of 150 cm, target centred in the image, LED lights	P
TS-14	Front view at a distance z of 150 cm, RS in a corner of the image, LED lights	F
TS-15	Angled view of -30° with respect to Y and -30° with respect to X, target centred in the image, LED lights	P

In the following sections the results of each test will be reported, remembering that they depend strongly on the camera pose estimation.

5.3.1 TS-01

The test conditions are reported only for the first test, as an example for the information that must be considered in the results evaluation (see Table 10).

Table 10: TS-01 test conditions

Test Conditions					
Test date and start time	30/01/2020 10:58				
Computer characteristics:	Intel® Core™ i7-6500 CPU @2.50 GHz with 16 GB RAM. OS: Windows 10 Home 64 bit				
Camera Model:	Microsoft LifeCam HD-3000				
Camera Matrix:		1121.396 0.000 0.000	0.000 1119.820 0.000	663.541 359.443 1.000	
Distortion coefficient:	0.260	-1.950	0.000	-0.003	4.178
Camera properties:	Exposure:		-6.0		
	Brightness:		110.0		
	Contrast:		10.0		
Camera Relative Position (Tvec) X-Y-Z [cm]		-24.867	14.879	72.072	
Camera Relative Orientation (Rvec) Around X-Y-Z [deg]		176.538	1.890	0.587	
Light source	Neon lamp				

In this test the camera is placed in front of the reference plane and centred with respect to the scene. In fact, the X and Y coordinates of the T vector, evaluated in the camera RS, are about half of the true coordinates of the OBJ marker. Just to remember, the T vector is the translation of the camera RS to the desired RS. The camera is placed at 75 [cm] from the plane where the object is placed, and the Z coordinate confirms it. Regarding the angles, they are about zero according to the geometry just described, except for the angle around the X axis. This angle is about 180° because it represents the rotation between the two reference systems. Finally, these values are not precise because of the various errors that occur during calibration and for the uncertainty about the position of the camera optical centre.

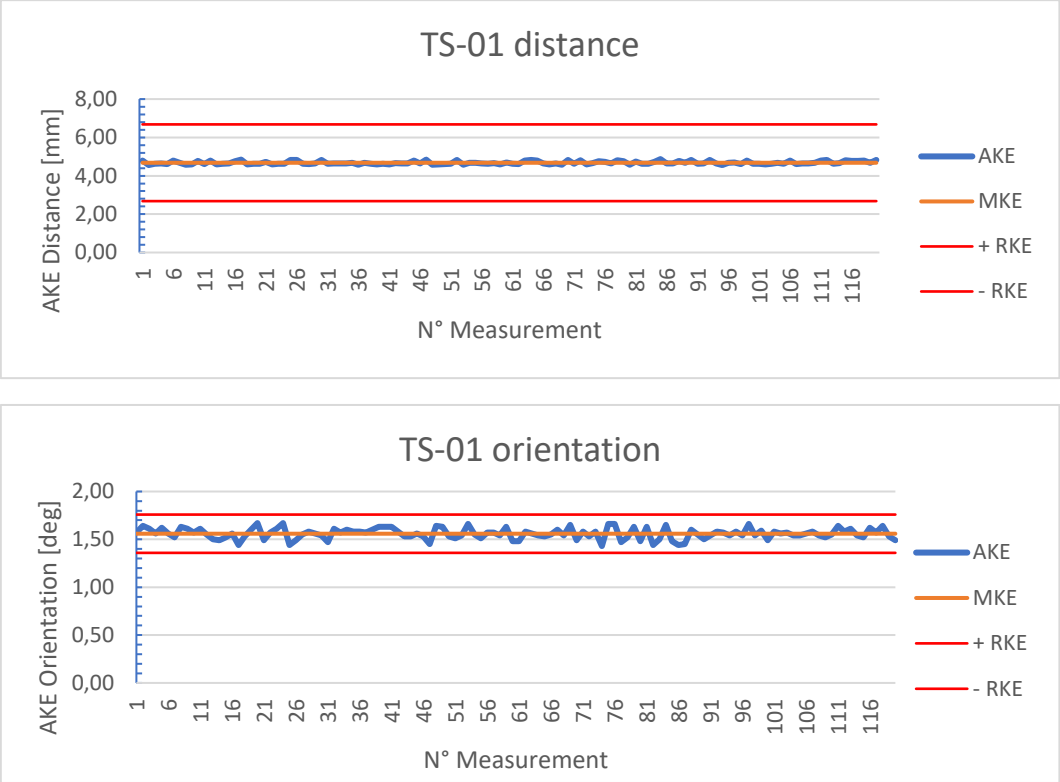


Figure 59: TS-01 distance and orientation AKE

The Figure 59 shows the results for both position and orientation. The position is evaluated by the software in coordinates, but to analyse the final error committed by the system, it was chosen to represent the distance module. The Mean Knowledge Error (MKE), or the average error committed during the whole test, is reported together with the AKEs for each individual measurement. The RKE is then evaluated on the MKE, simply by adding and subtracting the one required by the requirement.

The TS-01 was successful, demonstrating the functionality of the system with an MKE of 4.68 $[mm]$ on the position and 1.56 $[deg]$ on the orientation. However, this configuration is not practical due to the proximity between the object and the camera, and consequently a restricted FOV.

The Figure 60 shows a frame acquired by the camera with various information relating to the test in progress. Furthermore, it can be noted that the desired reference system is saved and shown on the screen, even though the 3D calibration tool is covered. This demonstrates that, if the camera is fixed in space, once its pose is evaluated, the calibration tool would no longer be needed.

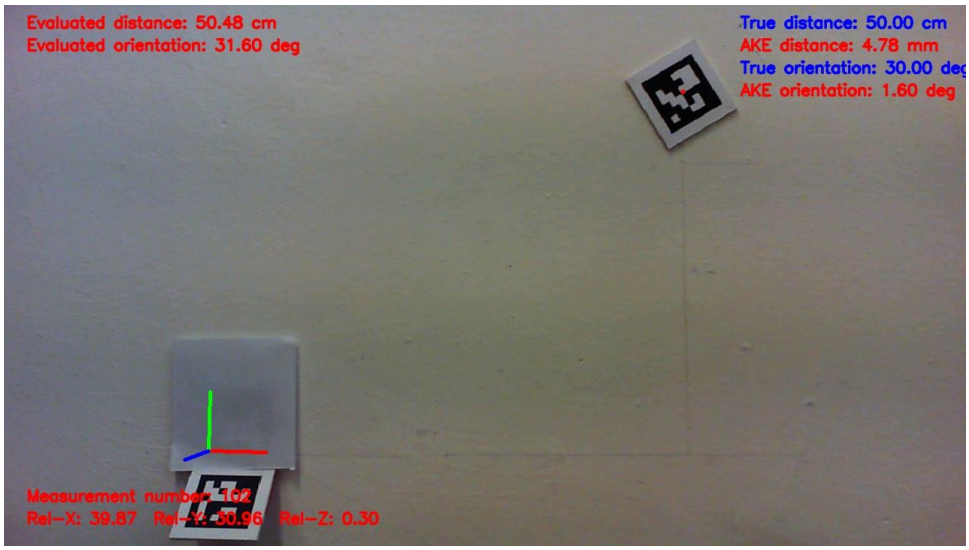
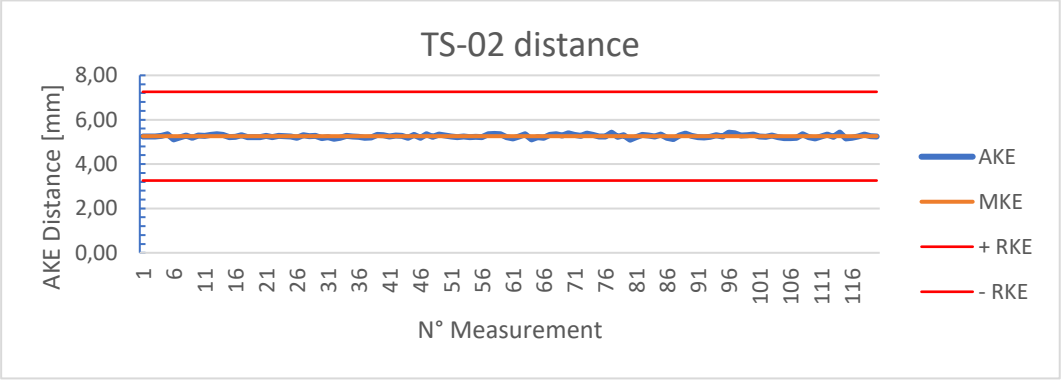


Figure 60: TS-01 example frame

5.3.2 TS-02

The TS-02 has the same configuration as the previous test, but the distance between the object and the camera is 100 [cm]. The MKE for the distance is 5.26 [mm] and for the orientation is 1.08 [deg]. The accuracy on the position determination is slightly lower than in the previous case. This may be due to the longer distance, but the calibration process is decisive, so this conclusion may not be valid. Even so, all the requirements are met, and the test can be considered passed.



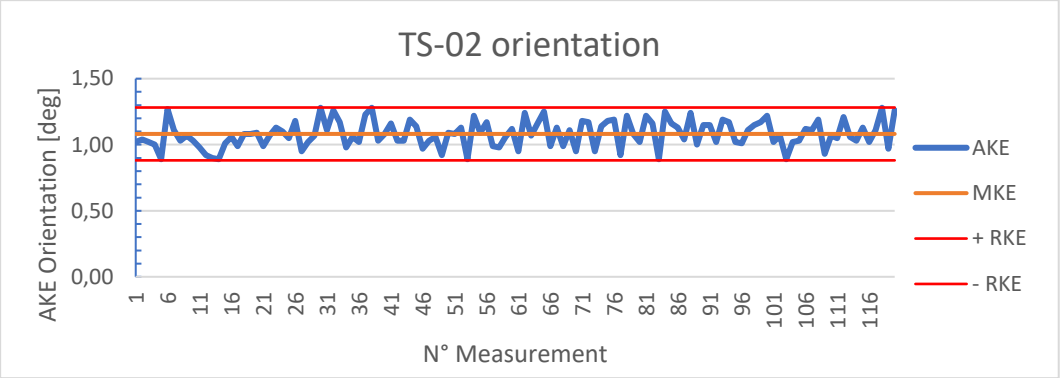
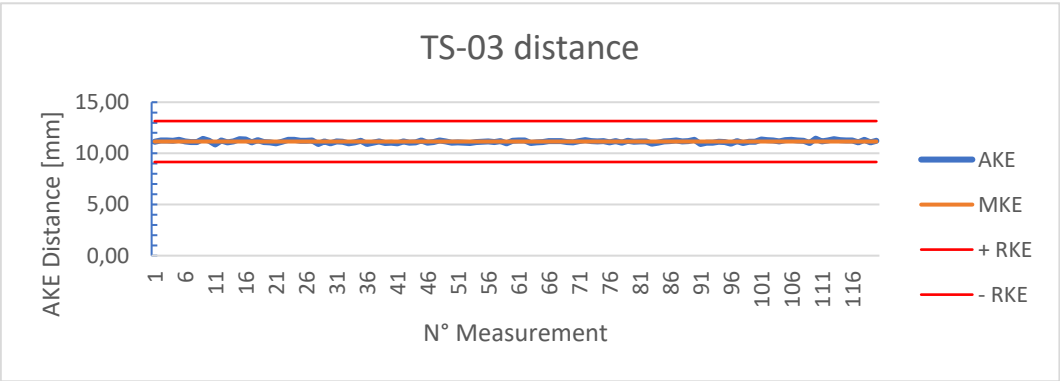


Figure 61: TS-02 distance and orientation AKE

5.3.3 TS-03

The distance is augmented to 150 [cm]. As it can be seen, the average error for the position increases to 11.16 [*mm*], but the measurements are very stable, they do not oscillate. The same cannot be said for orientation, which instead fluctuates over time. The cause can be attributed to various noises present in the images. The errors meet the requirements, but only slightly, especially for the RKE.



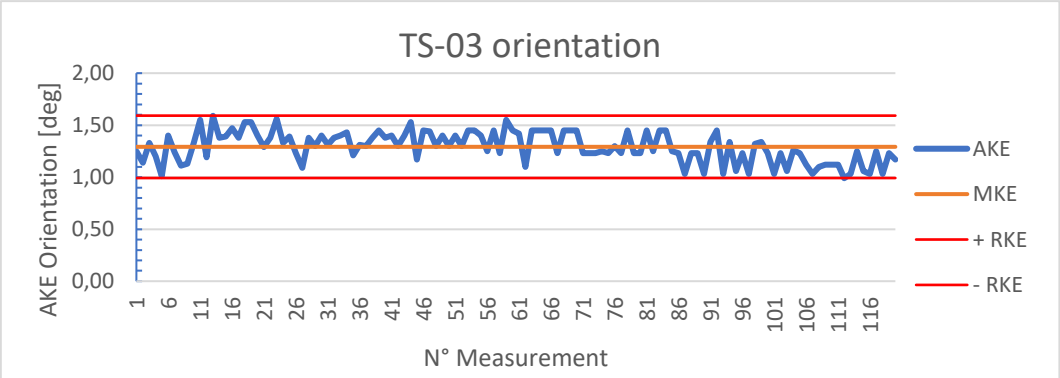


Figure 62: TS-03 distance and orientation AKE

5.3.4 TS-04

The distance is augmented to 200 [cm]. The position MKE is better than the 150 [cm] remote case, again confirming how much the calibration affects rather than the distance. Regarding the orientation determination, it does not meet the requirements, both for the maximum AKE that is 7.27 [deg] and for the RKE. In fact, there are peaks that make the measurement unstable and therefore unreliable. For these reasons the test is failed.

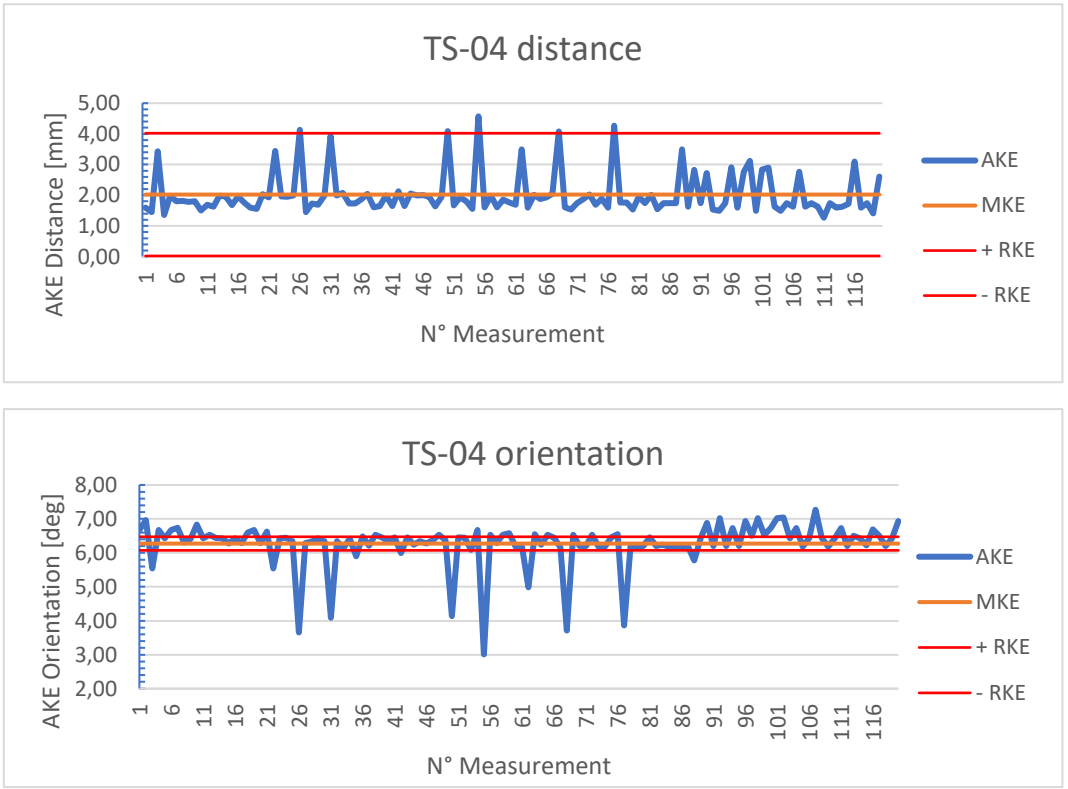


Figure 63: TS-04 distance and orientation AKE

However, the test showed a limitation of the system. While the distance increases, the accuracy decreases, especially for the orientation. To overcame this constrain, a higher quality hardware can be used, increasing the resolution of the images, or the OBJ marker can be enlarged, making it easier to be recognized. The second solution is less practical as the encumbrance on the TowerSat would increase.

5.3.5 TS-05

Having found an upper limit of the distance, it was necessary to test the performance by adopting a tilted view. This configuration is needed as it would allow a greater view of the operating area. In the TS-05 the camera was placed 150 [cm] away from the plane of the object with respect to the z coordinate and shifted to the left to form an angle of -30 [deg] around the Y axis. The Figure 64 shows the test configuration and is represented as the target is at the centre of the camera FOV.

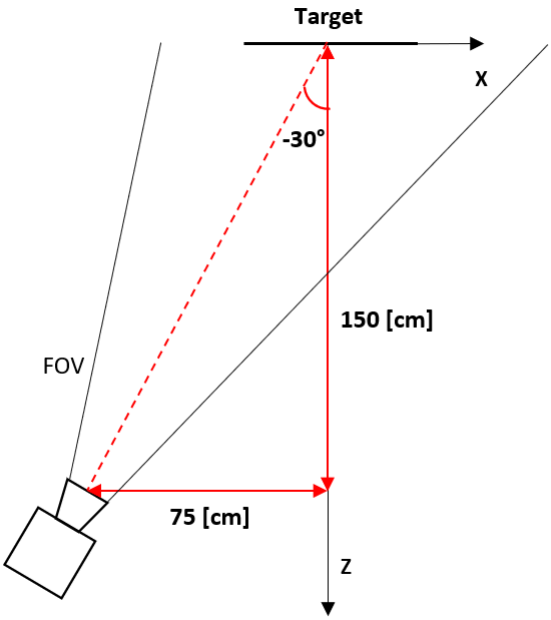
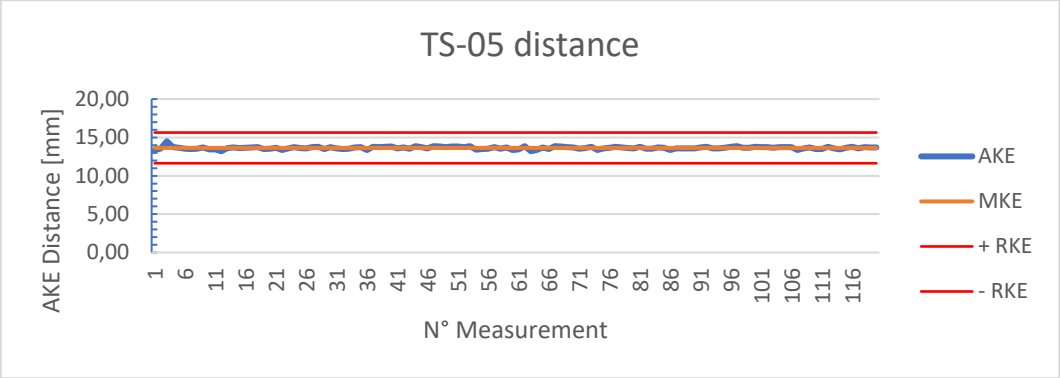


Figure 64: TS-05 configuration

The results in Figure 65 once again show the accuracy obtainable on the position determination, although it is lower. This could be linked to the calibration process. The orientation undergoes high fluctuations in the measurements, once again due to the noises present in the image, caused by the light source, the size of the marker or the quality of the camera.



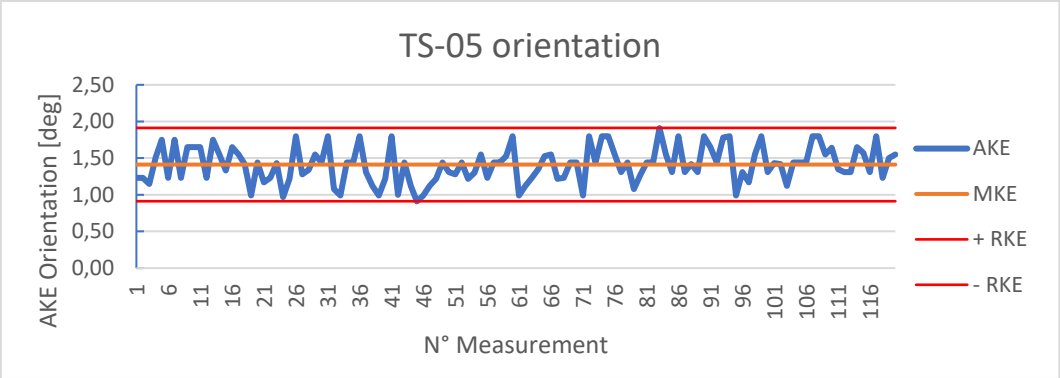
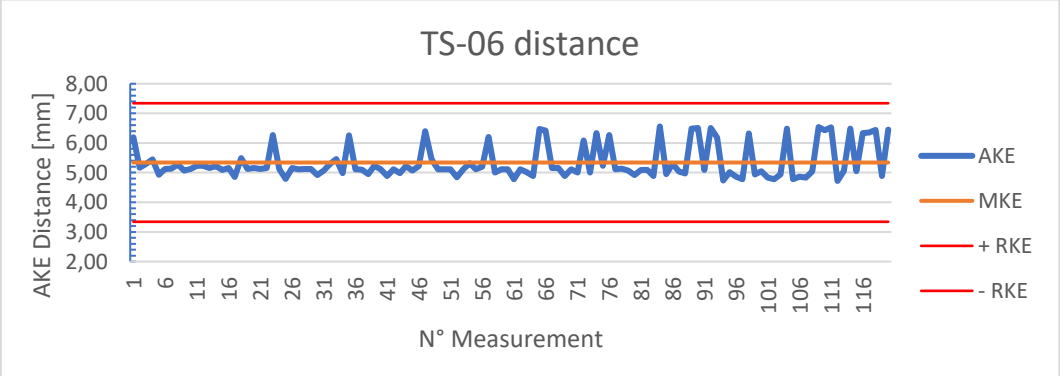


Figure 65: TS-05 distance and orientation AKE

5.3.6 TS-06

In this test the camera is doubly angled with respect to both the X and Y axis of the reference plane. Both angles are -30 [deg]. The test passed and although the configuration is more tilted than the previous test, better results were obtained. This is due to the camera pose estimation, that influences seriously the results.



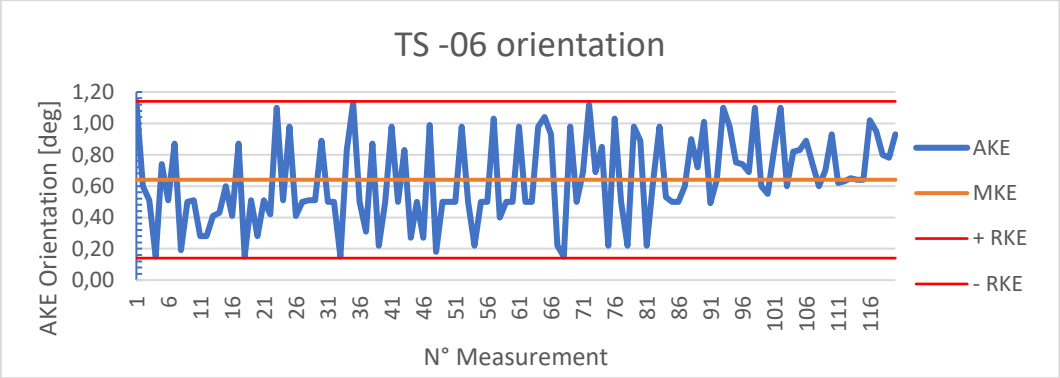
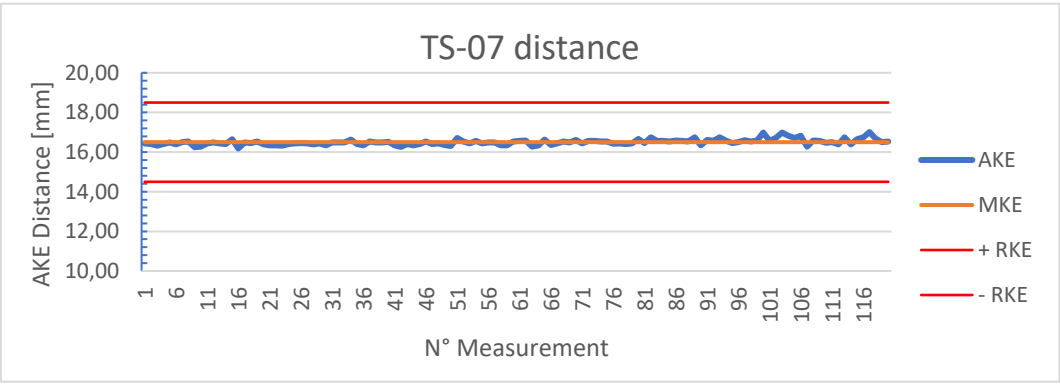


Figure 66: TS-06 distance and orientation AKE

5.3.7 TS-07

The TS-07 is very similar to the TS-06, with the difference that the angle around the x axis is +30 [deg]. In other words, if before the camera saw the scene from below, in this test it sees the scene from above. The goal is to confirm the results obtained with the previous test, showing that the system is not affected by angles in the view, but that the most influencing elements are the external calibration and the hardware.



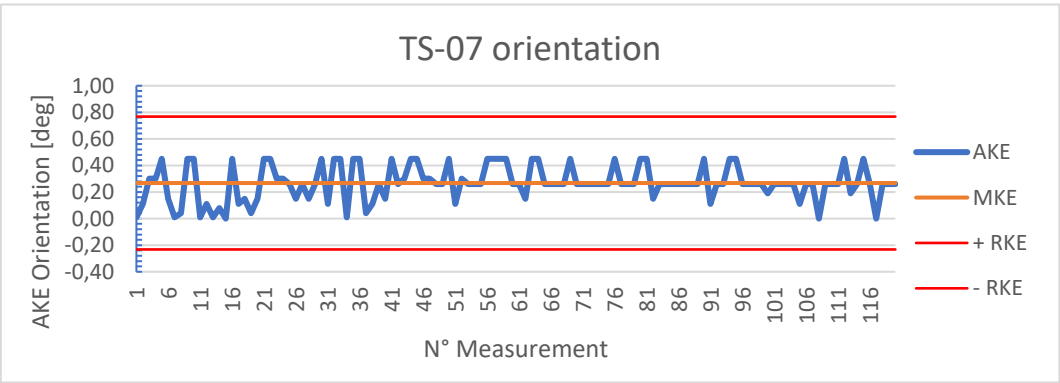


Figure 67: TS-07 distance and orientation AKE

5.3.8 TS-08

In this test, the same position of the camera has been maintained with respect to the RS of the scene, but the target is no longer centred in the FOV. In fact, the RS was placed in a corner of the FOV, to cover a larger area. The goal is to evaluate the performance in case the RS (created with the 3D calibration tool) is near the edges of the image, where the distortions described in section 3.1.1 are greater. The Figure 68 shows the configuration adopted. It can be noted that the bisector of the FOV no longer coincides with the junction between the camera centre and the centre of the scene.

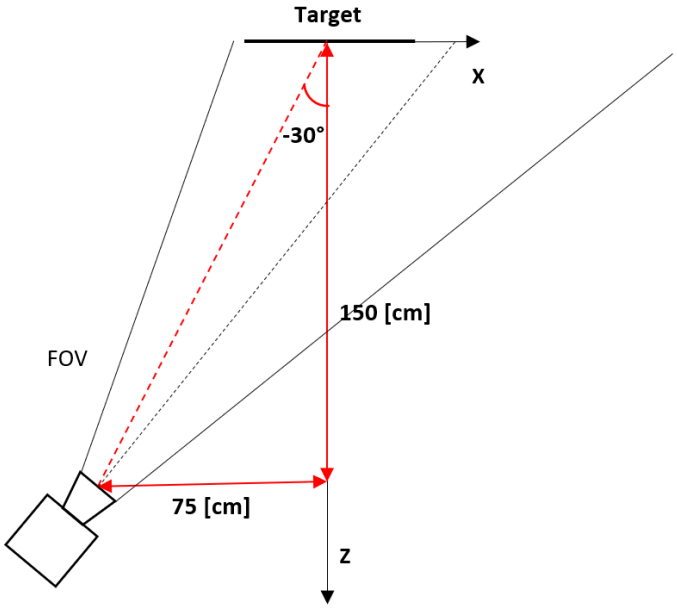
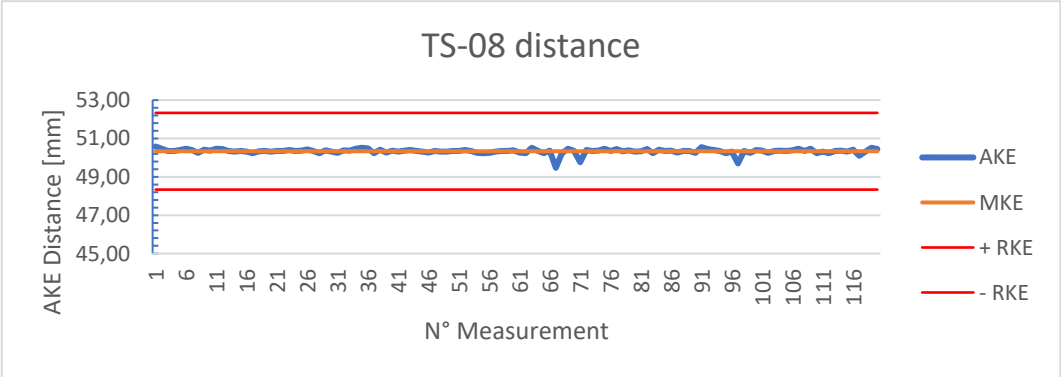


Figure 68: TS-08 configuration

The Figure 69 shows the test results. According to the desired requirements, it is to be considered failed, due to the high AKE on the position.



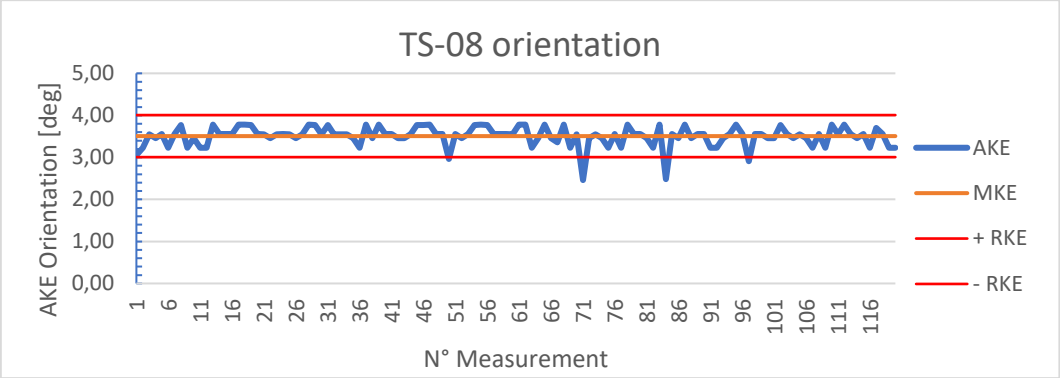


Figure 69: TS-08 distance and orientation AKE

By carefully parsing the results, the cause of the high errors was found in the camera pose estimation. In fact, the camera was placed at $(-25, 30, 150)[cm]$ (in (x, y, z) coordinates) from the desired RS, but the calibration process has provided a translation vector of $(65.782, -33.833, -145.959)[cm]$. For this reason, the test was carried out again, providing the results shown in Figure 70.

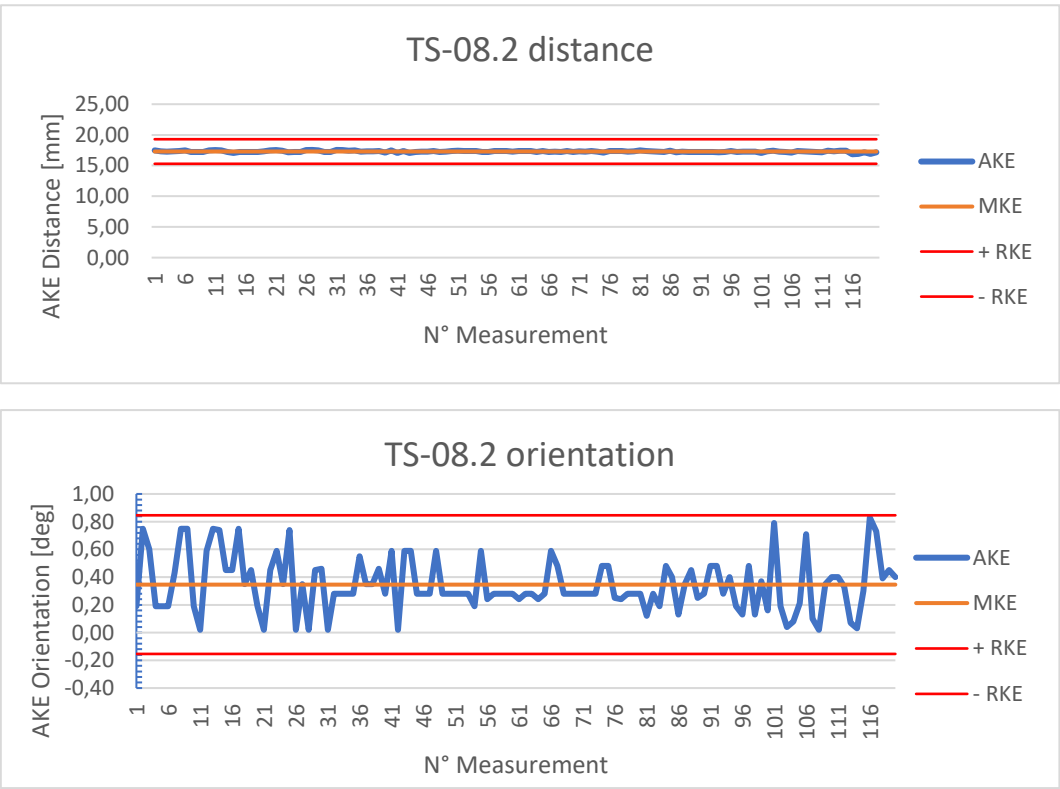


Figure 70: TS-08.2 distance and orientation AKE

The TS-08 has conclusively demonstrated the cruciality of the camera pose estimation process.

5.3.9 TS-09

In this test the camera was placed in the same position as the TS-03, but with the RS in a corner, as in the previous case. In this session the first test has failed due to the wrong external calibration too, therefore the results of the test performed correctly are reported.

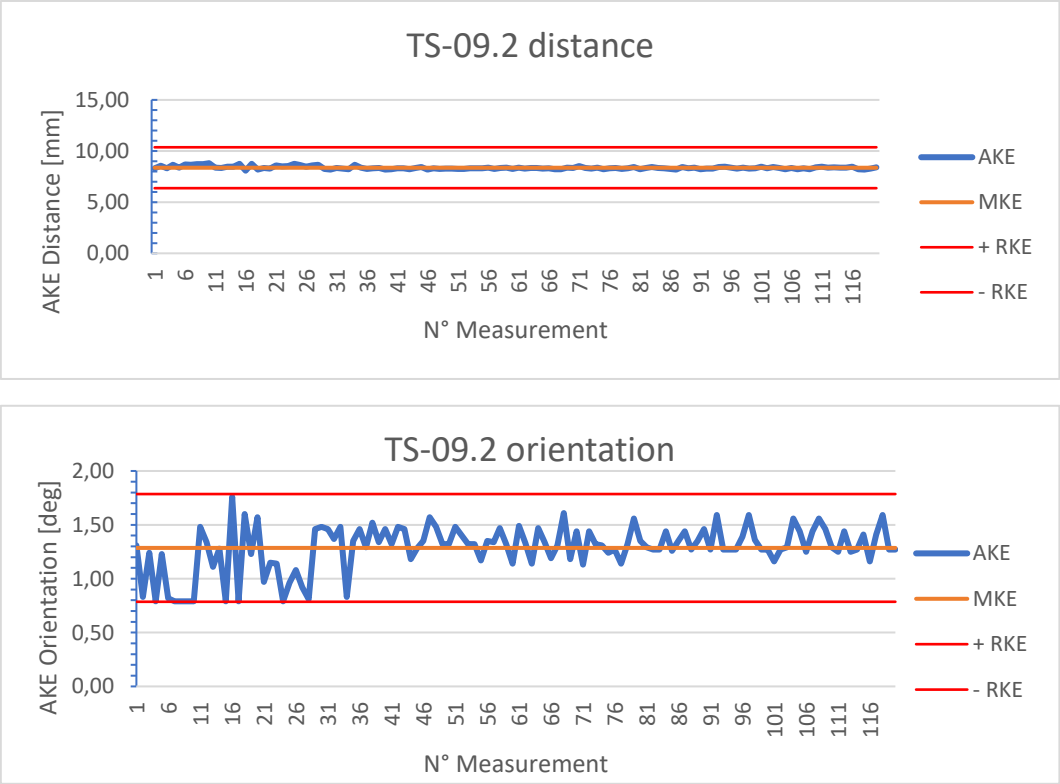


Figure 71: TS-09.2 distance and orientation AKE

5.3.10 TS-10

The previous tests have evaluated the performance of both the hardware and the software. In this session, on the other hand, the system has been tested to assess its behaviour by varying the light conditions. In fact, the tests have been conducted in a laboratory where there are the neon lamps. The artificial light has a pulsing characteristic that is caused by the 50 Hz or 60 Hz AC voltage used to power the lights.

The camera shutter works with a certain frequency, depending on the amount of light in the environment. The more light is in the scene, the faster the shutter will be set. For this reason, it is very capable of detecting these high pulses frequencies in the light intensity.

If the two frequency are similar, they can interfere, generating an effect on the image called *banding* (or *flickering*). It appears as a series of dark and light bands horizontally across the image, that could create some noise on the marker determination. The Figure 72 shows an example of this effect.

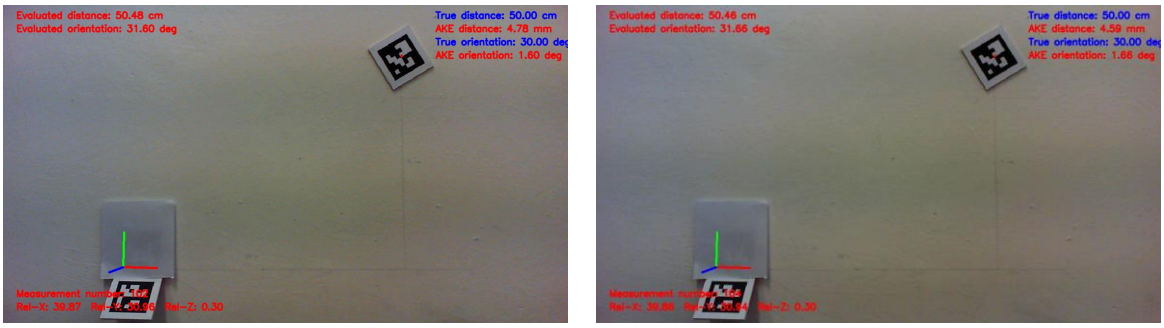


Figure 72: Flickering effect

The goal of this test is to determine an influence on the performance of the light condition. The position of the camera is the same as that of the TS-03 test, i.e. frontal view at 150 [cm] from the target. However, this test was performed with the lights off, therefore with only sunlight entering from the laboratory windows, as shown in Figure 73.

Figure 73: TS-10 example frame

According to the results (see Figure 74) the test was successful, despite the absolute error on the position is almost at the limit imposed by the requirement, but, once again, it can be caused by the pose estimation.

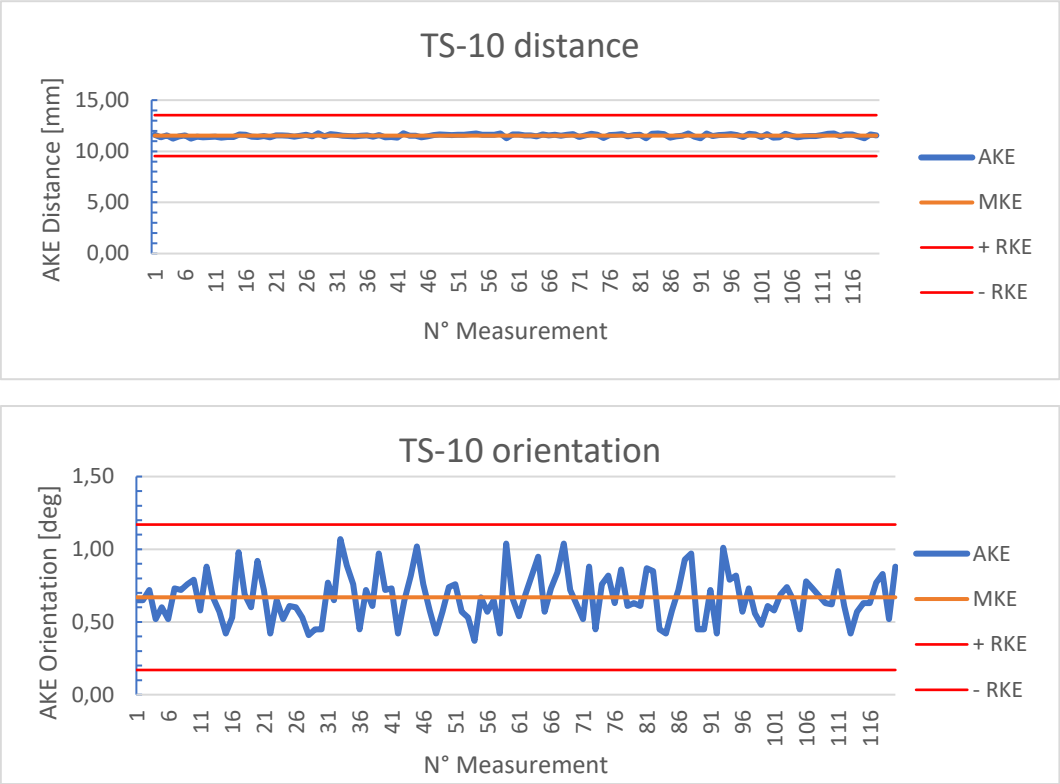
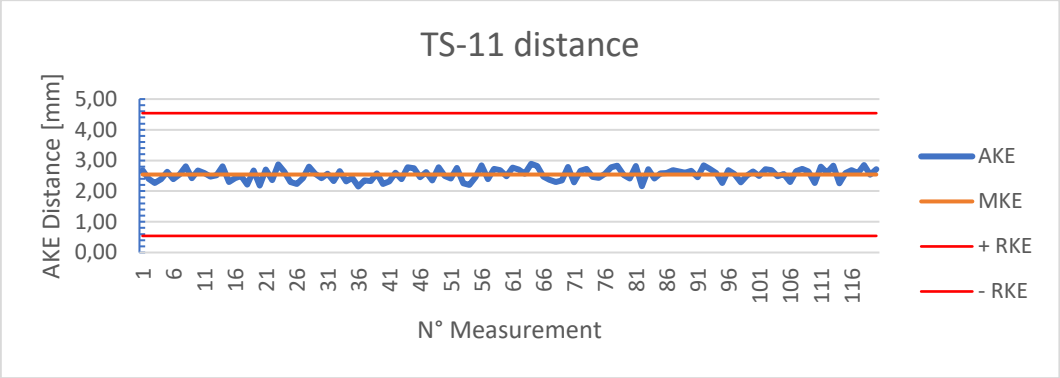


Figure 74: TS-10 distance and orientation AKE

5.3.11 TS-11

In the TS-11 the configuration adopted is the same as the TS-09, but with the lights off. The test was unsuccessful only for the stability on the orientation evaluation, therefore on the RKE requirement. The motivation is closely linked to the light condition. A variation in the light intensity generate a noise on the image, causing a different measurement than in the previous frame



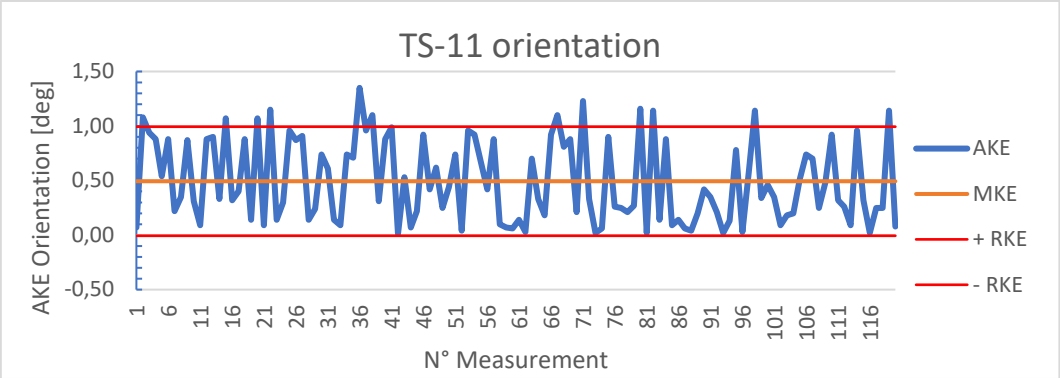
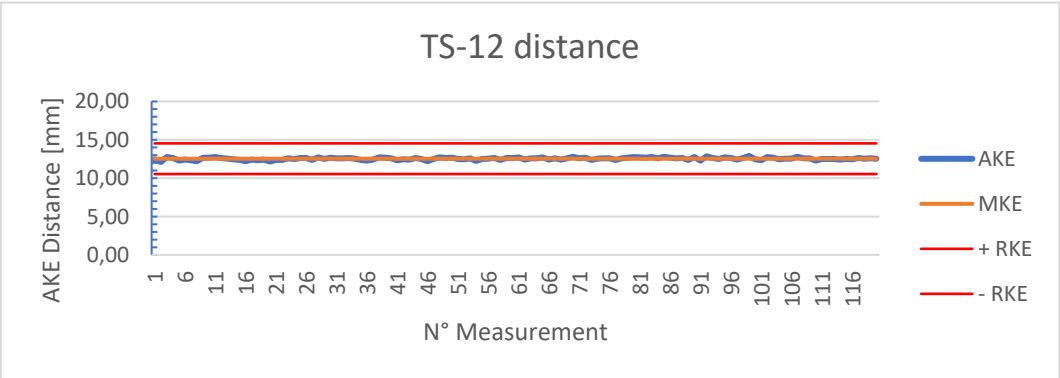


Figure 75: TS-11.2 distance and orientation AKE

5.3.12 TS-12

With the same configuration as the TS-06, this test also failed. In this case, however, it is the maximum AKE that exceeds that required.



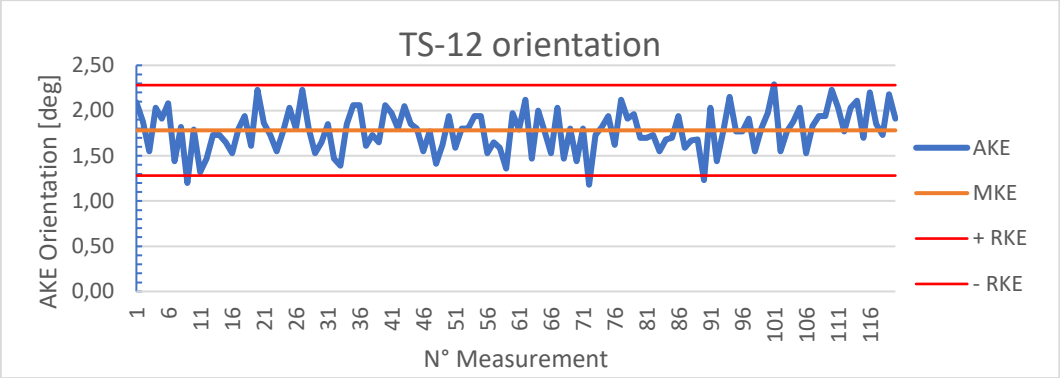
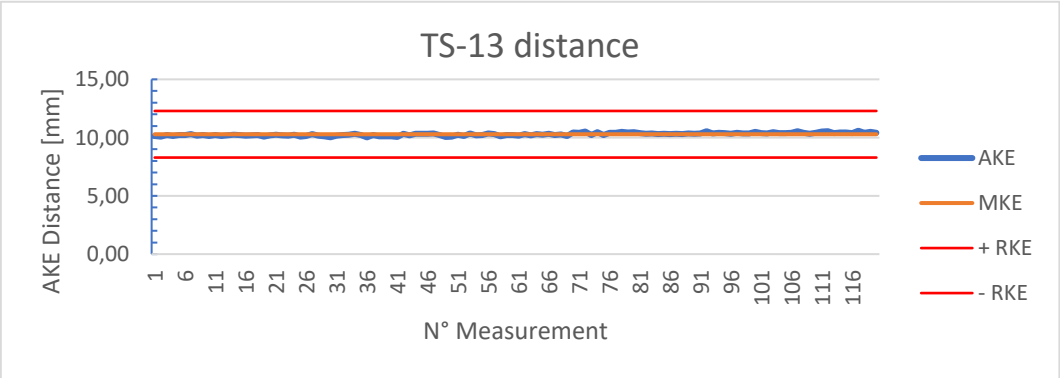


Figure 76: TS-12 distance and orientation AKE

Obviously, using sunlight would never have been a viable option, but these three tests served to understand the behaviour of the system.

5.3.13 TS-13, TS-14 and TS-15

To improve the lighting condition, it is possible to adopt LEDs, which provide a continuous source and do not interfere with the camera shutter. The following three tests were carried out with the same configurations as the TS-10, TS-11 and TS-12 respectively, but with LED lamps as light source. The results are shown in the next three figures.



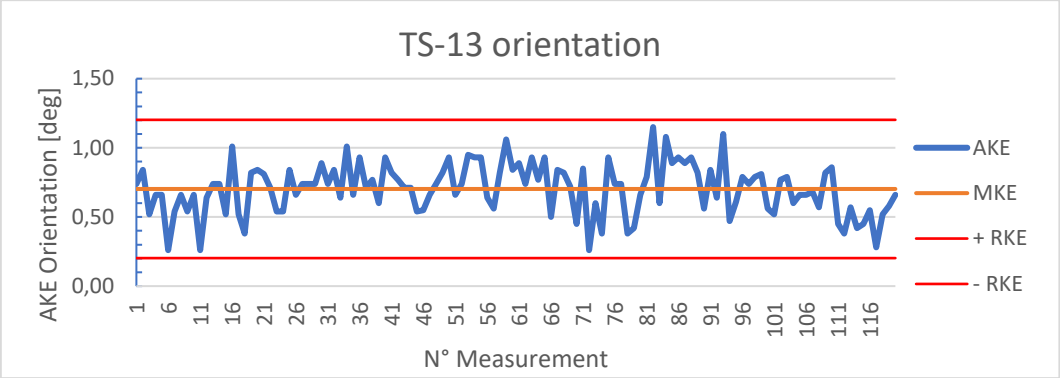
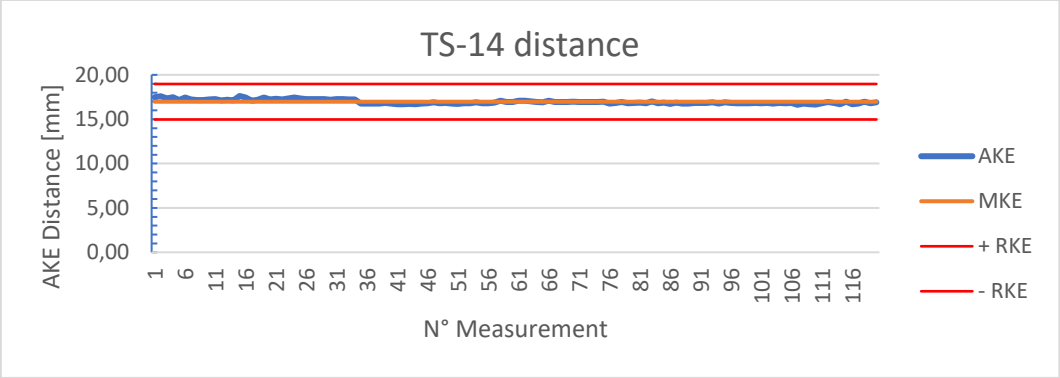


Figure 77: TS-13 distance and orientation AKE



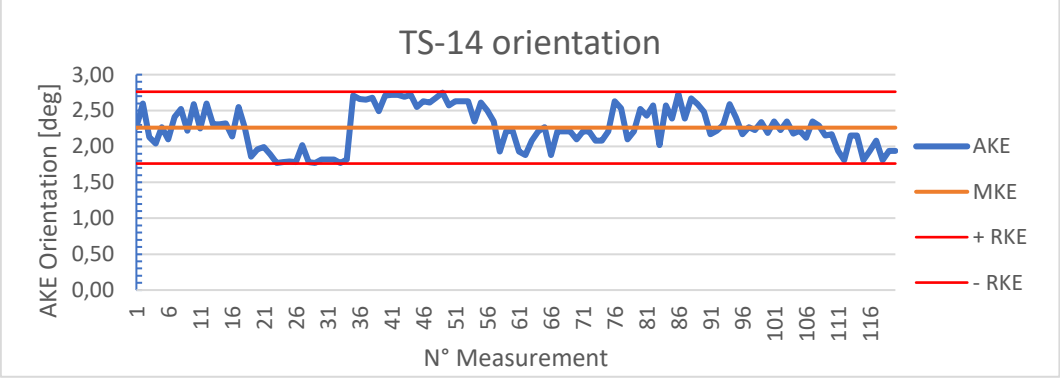
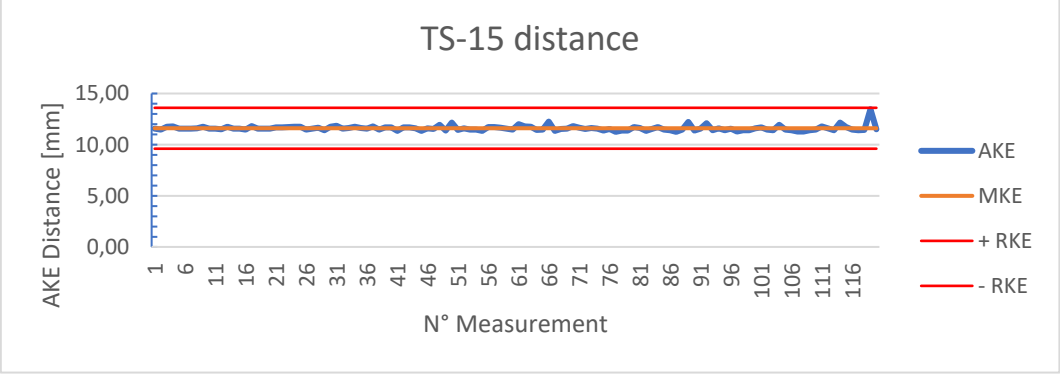


Figure 78: TS-14 distance and orientation AKE



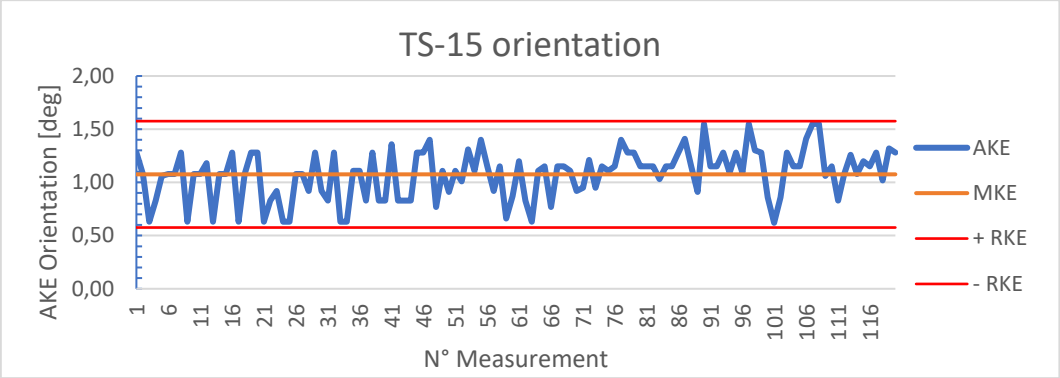


Figure 79: TS-15 distance and orientation AKE

The TS-14 test failed, again due to the orientation determination. The position instead meets the requirements but with an MKE of 16.97 [mm], that is very high. The cause can again be traced back to inaccuracies during the external calibration process.

The configuration of the TS-15 test is that assumed of the real application case. In fact, the camera could be placed in the corners of the laboratory, to cover a wider area of the frictionless table. The reference system to use will be near the edges of the camera FOV.

5.4 *LocSys_MultiCamera* Test Sessions

The *LocSys_MultiCamera* software has the purpose to make multiple cameras work together, to cover a larger area of the frictionless table. Tests were conducted using two cameras of the same model, to demonstrate the functionality of the software and its performance.

The test procedure for the multi-camera architecture is like the previous one. The cameras intrinsic calibration is not necessary, because, as mentioned in section 3.4, the intrinsic parameters does not change. The main problem with this architecture is the accuracy required to evaluate the cameras pose. The calibration process is crucial and must be done with the utmost attention. The results depend on it, as demonstrated in the previous architecture. For this reason, this step has been repeated several times for each test, to achieve high accuracy. In the final development of the system, the cameras shall be fixed on the walls of the laboratory, therefore it will not be necessary to carry out this procedure with each test.

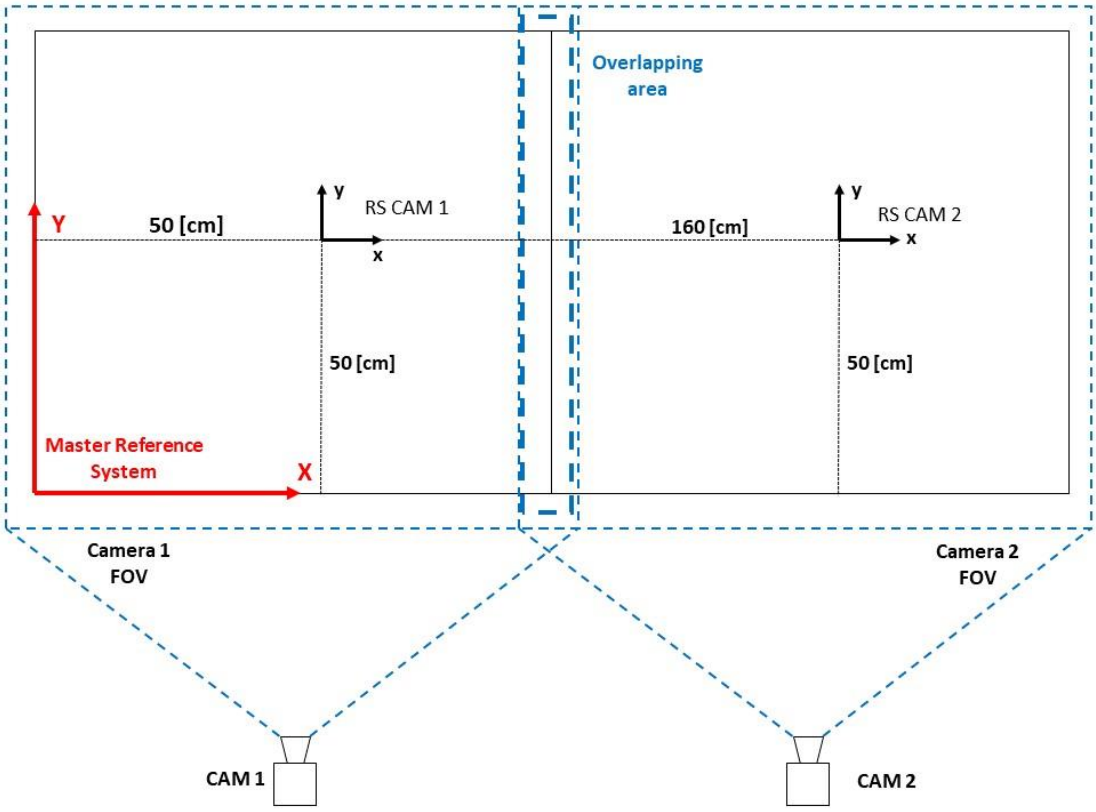


Figure 80: MultiCamera test configuration

To carry out the tests, a $220x80$ $[cm]$ rectangle was created to simulate the operating area. Each camera perfectly covers half of the area but straddling the two areas the camera FOVs overlap. The cameras are placed at 150 [cm] of height with respect to the rectangle, to have the least inclined vision possible.

The Figure 80 shows the configuration adopted. The dotted blue squares represent the cameras FOV. The overlapping area is highlighted by the thick blue rectangle. The reference systems of the individual cameras and their position with respect to the Master Reference System are also illustrated.

For this purpose, 5 different tests were performed, listed in Table 11:

Table 11: MultiCamera tests description

Test ID	Test Brief Description	Pass/Fail (P/F)
TM-01	The target in the field of view of the first camera	P
TM-02	The target in the field of view of the second camera	P
TM-03	The target in the overlapping area of the two cameras	P
TM-04	Dynamic test with cameras on the same side	P
TM-05	Dynamic test with cameras on opposite sides	P

The first two tests aim to demonstrate the ability to operate the two cameras individually and thus demonstrate the operability of the software. In the third test, the target is placed in the FOV overlap zone to demonstrate the software management of this situation. Finally, the last two tests are dynamic, with the target moving, to understand how performance varies, using two opposite configurations.

Like the single camera architecture, the characteristics of the cameras and the parameters used in the external configuration files are shown below. The Table 12 contains the intrinsic and extrinsic parameters for both the cameras used. In the first four tests the cameras were fixed in the laboratory, therefore the translation and rotation vectors do not change, there was no need the performed the extrinsic calibration. This information together with the computer used, the date and start time of the test and the light source, represent the test conditions.

Table 12: Cameras parameters

Camera 1	Camera Model		Microsoft LifeCam HD-3000		
	Reference System		Relative X		50.00
			Relative Y		50.00
			Relative Z		0.00
	Camera Name		microsoft_1		
	Camera Matrix		1138.970	0.0	660.606
			0.0	1138.185	359.034
			0.0	0.0	1.0
	Distortion coefficients				
	0.241	-1.711	-0.001	-0.003	3.519
	Camera properties:		Exposure:		-6.0
			Brightness:		143.0
			Contrast:		10.0
	Camera Relative Position (Tvec) X-Y-Z [cm]		-6.945	-12.975	149.608
	Camera Relative Orientation (Rvec) Around X-Y-Z [deg]		153.690	-5.319	1.721
Camera 2	Camera Model		Microsoft LifeCam HD-3000		
	Reference System		Relative X		160.00
			Relative Y		50.00
			Relative Z		0.00
	Camera Name		microsoft_1		
	Camera Matrix		1138.970	0.0	660.606
			0.0	1138.185	359.034
			0.0	0.0	1.0
	Distortion coefficients				
	0.241	-1.711	-0.001	-0.003	3.519
	Camera properties:		Exposure:		-6.0
			Brightness:		143.0
			Contrast:		10.0
	Camera Relative Position (Tvec) X-Y-Z [cm]		0.817	-11.245	149.053
	Camera Relative Orientation (Rvec) Around X-Y-Z [deg]		153.271	3.068	0.056

In the *multiCamera* architecture it is necessary to provide all the information related to the cameras used, such as the position of the desired reference system for the camera and the position and orientation of the calibration tool, as described in section 4.5.2. This

data is reported in the *config_camera_list.json* file and the following settings have been used for these tests:

Table 13: Cameras settings

	Camera 1	**Camera 2**
Camera ID	1	2
Camera Name	microsoft_1	microsoft_2
Camera RS position	$(50.0, 50.0, 0.0)$ $[cm]$	$(160.0, 50.0, 0.0)$ $[cm]$
Calibration tool relative orientation	0.0 $[deg]$	180.0 $[deg]$
Calibration tool relative position	$(0.0, 0.0, 0.56)$ $[cm]$	$(0.0, 0.0, 0.56)$ $[cm]$

As it can be seen in the Table 13, the reference systems are in the centre of the interested area of the cameras. The position and orientation are determined by detecting the marker position in pixels and then projecting this point in 3D space, using the chosen reference system. The further the point is from the RS the more errors increase.

For this reason, instead of placing the virtual RS in a corner of the area, it has been positioned in the centre. In any case, the software is provided with information on the position of the RS with respect to the origin of the reference plane, or the table corner. In this way, errors are reduced, keeping the same reference system desired (see Figure 80).

Finally, for all the tests, the following settings have been adopted in the *config_init_MultiCamera.json* file:

Table 14: LocSys_MultiCamera test settings

Number of cameras	2
System Width	1280
System Height	720
Marker Length [cm]	5.0
Z [cm]	0.3
OBJ Marker ID	0
OBJ Real Center	[2.5, 2.5]

Tolerance	1e-6
Multiple tests	true
Calibrate intrinsic parameters	false
Calibrate extrinsic parameters	false
Resize window	true
Screen height resolution	1080
Window Name	Acquisition
Use undistortion	true
Save test video	false
Seconds between results	0.5
Number of measurements	120

5.4.1 TM-01 and TM-02

In the first two tests, the OBJ marker is seen by the cameras individually. The goal is to demonstrate the functionality of the software and that the performance achieved with the single architecture is respected in this one. The results are shown in Figure 81 and Figure 82.

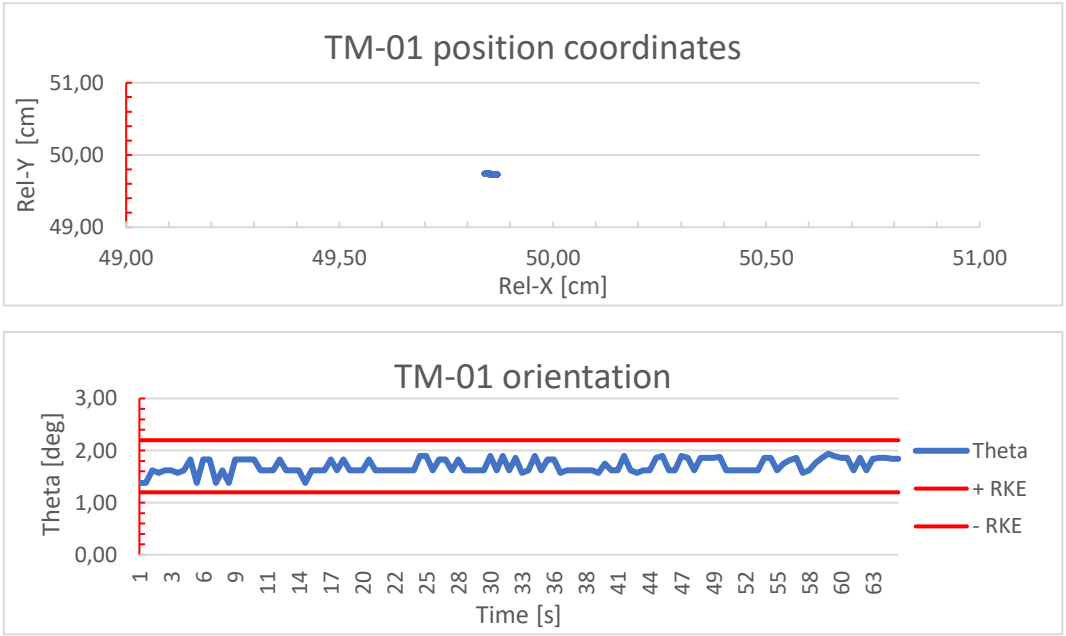


Figure 81: TM-01 position and orientation results

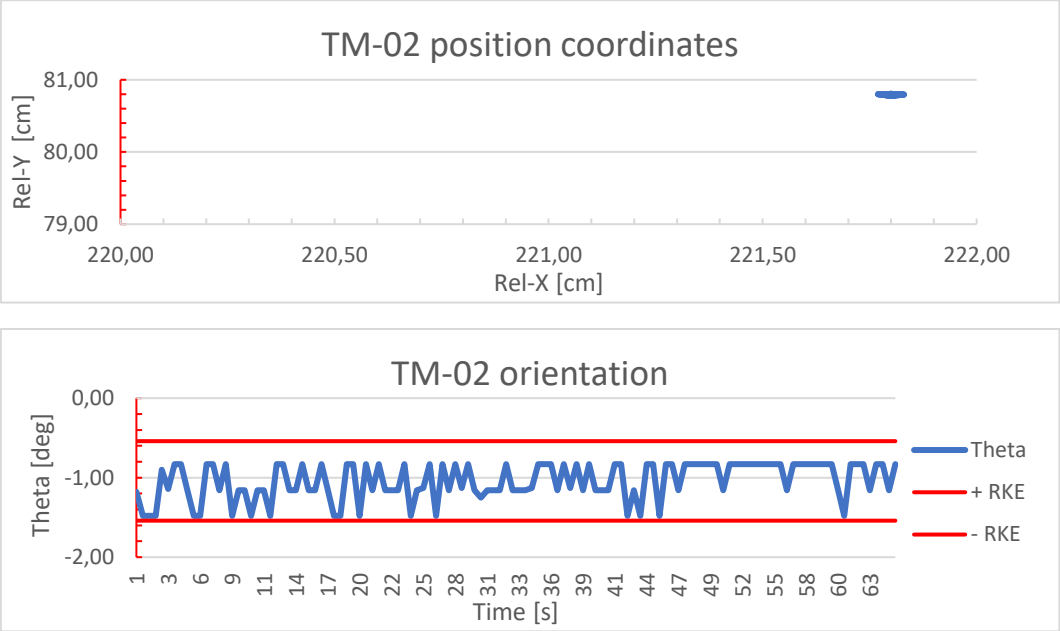


Figure 82: TM-02 position and orientation results

In both cases, 120 measures were collected. Both position and orientation meet the requirements on AKE and RKE. Two frames of both tests are also shown, where it can be seen the position of the virtual reference system used and the position of the markers. The Figure 83 shows the reference systems adopted by the cameras.

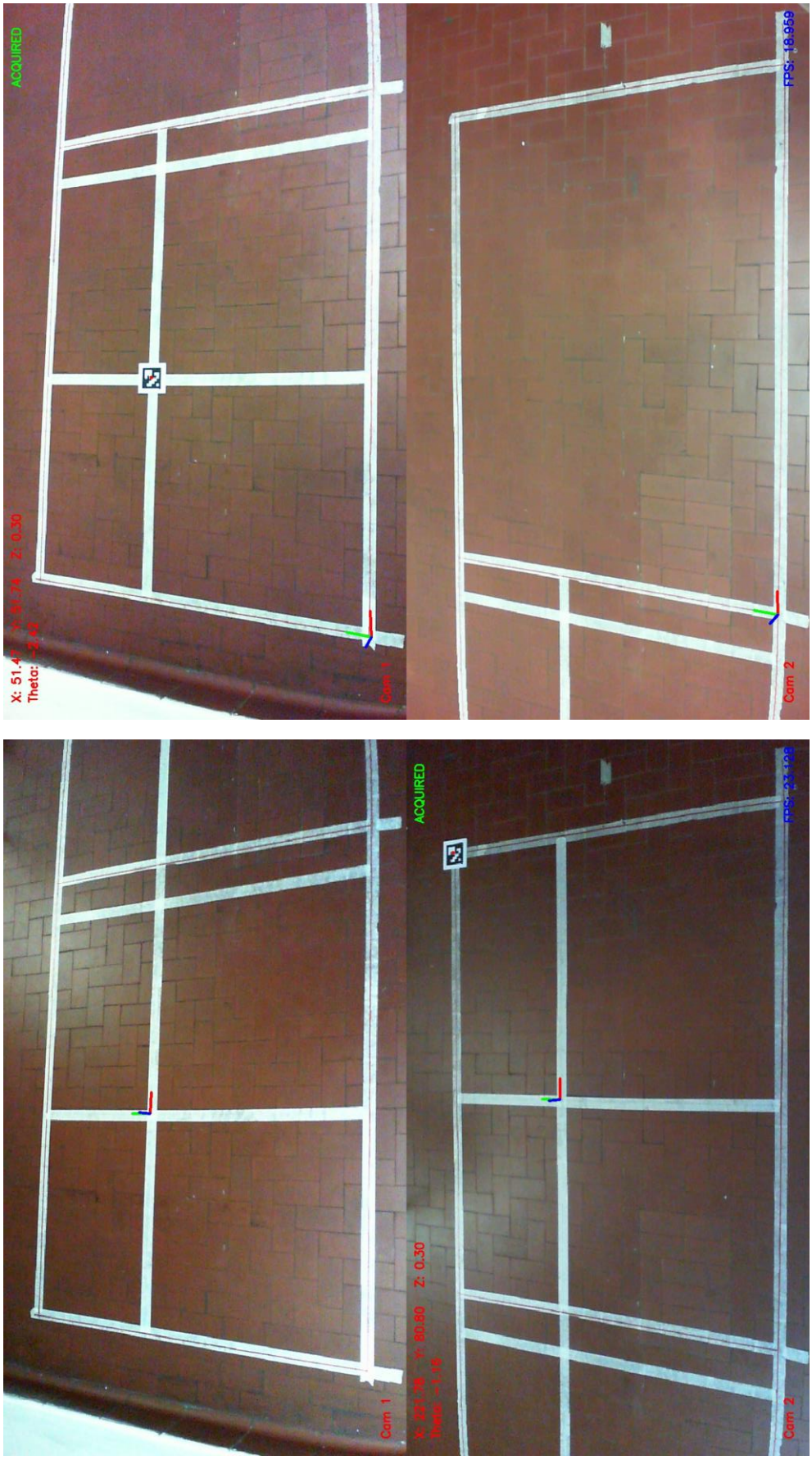


Figure 83: TM-01 (above) and TM-02 (below) frames

5.4.2 TM-03

The TM-03 aims to demonstrate the software ability to manage the situation in which the OBJ marker is in the overlapping area. When two or more cameras see the same marker, the software evaluates the coordinates separately and then averages them, to improve the accuracy on the evaluation.

If one the measurements is wrong, it would seriously affect the average taken. For this reason, the software needs improvement, for example by using knowledge of previous measurements and the variation speed to make estimates. In the Figure 84 the position coordinates are concentrated around the point $(110, 81.5)$. The position error is concentrated on the y coordinate, but despite this the MKE resulting from the test is 8,89 $[mm]$, so it respects the requirement. The orientation MKE instead is 0.068 $[deg]$.

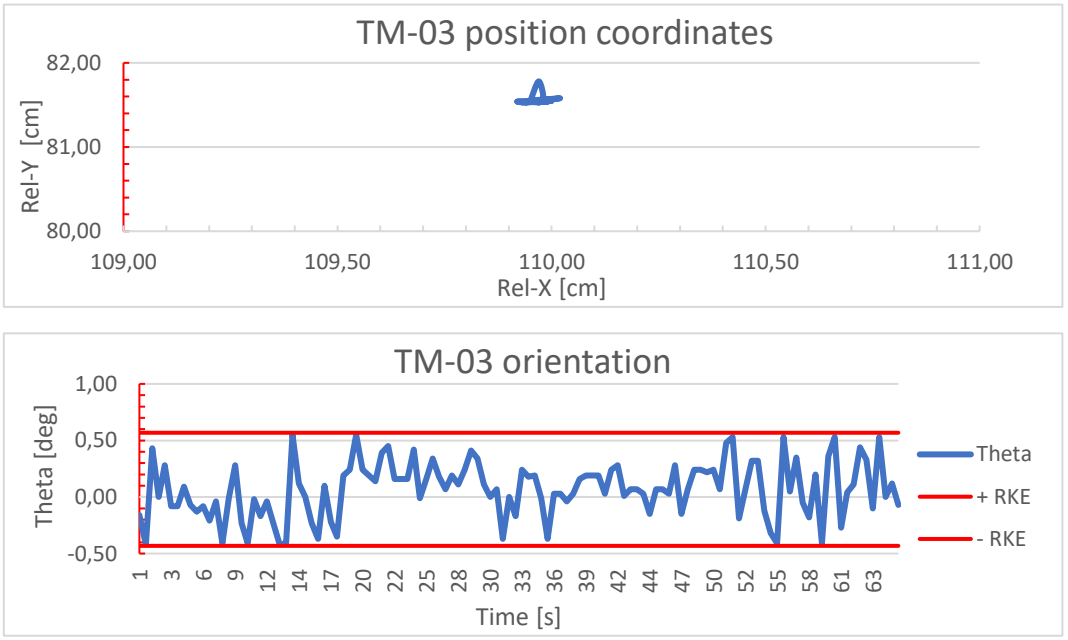


Figure 84: TM-03 position and orientation results

5.4.3 TM-04 and TM-05

The two final tests were made in dynamics, that is, moving the OBJ marker. The movement was simulated by dragging the marker along a line. Considering the human error in dragging the object, not having a remote-controlled mechanism available, the goal was to understand the ability to not skip measurements. In fact, with the object in motion, the image may present more noise, making it more difficult to locate the exact position of a marker corner. The Figure 85 and Figure 86 show the trend over time of the orientation and the trajectory followed by the marker. This type of graph is the one assumed in the CAST Control Centre, like an interface to the operator, who can monitor the movements of the TowerSat.

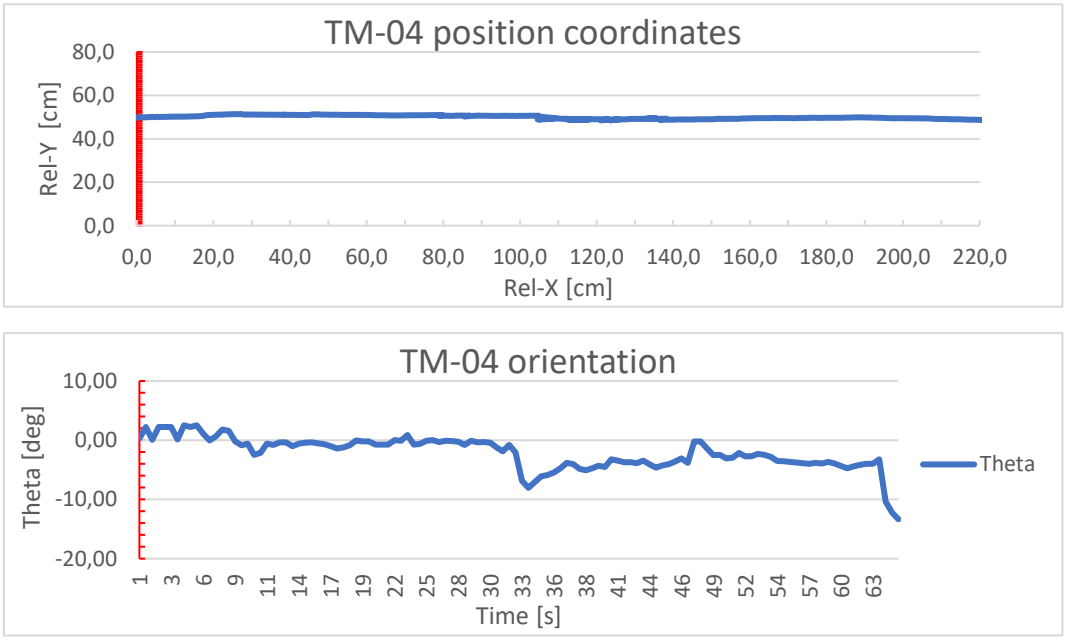
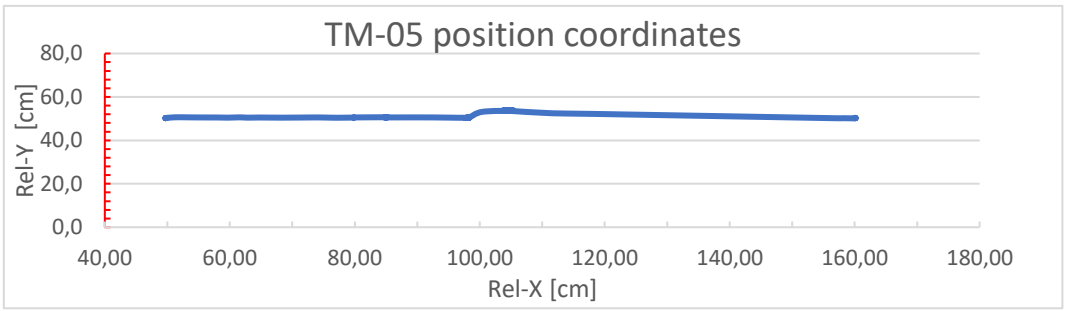


Figure 85: TM-04 position and orientation results



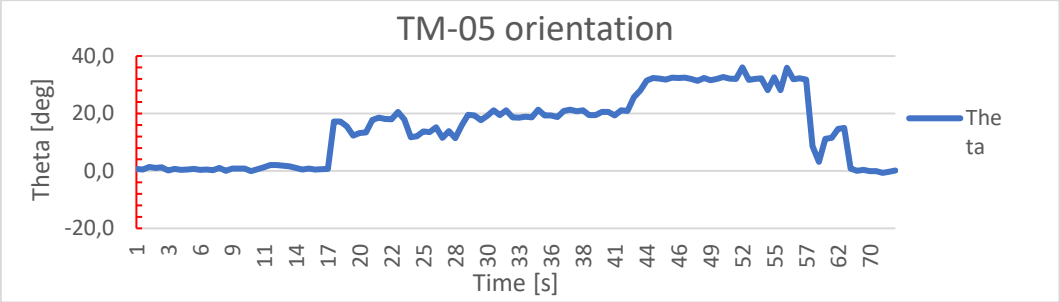


Figure 86: TM-05 position and orientation results

In the TM-05 test another configuration was adopted, different from the one in the first four tests. The "Camera 1" has not change position, while the "Camera 2" has been placed on the opposite side of the area concerned. In this configuration (visible in Figure 87), it was necessary to estimate again the camera pose, by varying the position and orientation of the calibration tool. Being on the opposite side, the tool must be rotated 180 [deg] in order to perform the calibration.

Whit this test, the software ability to calibrate the camera with respect to any position has again been demonstrated, correctly entering the information in the external files and using the operator's utmost precision.

Figure 87: TM-05 frames

5.5 LocSys Baseline

Following the tests results, it is possible to identify a baseline to obtain the best performance from the localization system.

The tests were carried out with cheap web cameras, using a resolution of 1280 x 720. The ideal features for this hardware are listed below:

- Maximum distance between camera and target of about 150 [cm]
- Camera centred and tilted with respect to the area to be covered, so to obtaining a FOV like the one shown in Figure 80
- Side length of the ArUco marker to be placed on the object of 5 [cm]
- Use LED lamps as light source

Concerning the calibration tools, the larger their dimensions, the more the accuracy increases. An A3 format ChArUco board was used during this thesis, obtaining excellent results in terms of reprojection error. A larger board could further reduce this error. The same reasoning can be done for the 3D external calibration tool.

It is necessary to remember the importance of the precision by the operator while using the 3D calibration tool. LocSys works thanks to the reference systems. Translations and rotations are carried out between the various RS. If errors occur during the calibration, they propagate between one reference system and another.

As already illustrated during the tests on multi-camera architecture, the position of the 3D object point is projected into space starting from the camera reference system. To avoid high projections, that create high errors, it is advisable to position the camera RS in the centre of its FOV, as shown in Figure 80. The master reference system can be positioned anywhere in the entire operating area.

All markers used, including patterns for calibration, must have an opaque surface, to reflect less light. In this way their recognition is easier, reducing the noise in the images.

Finally, with two cameras placed at 150 [cm] from the table, it is possible to cover an area of 220 x 80 [cm]. This area obviously depends on the camera FOVs. Since a single camera can cover a large area, it is possible to double the number of cameras, to increase the number measurements and consequently the accuracy of the system. LocSys is already set up to use an indefinite number of cameras.

6 CONCLUSIONS

The goal of this thesis was to develop a localization system (LocSys) to be integrated into the CAST project. Among the various localization technologies, it was chosen to use optical cameras together with some fiducial markers, in this case the ArUco markers.

During the thesis two software have been developed. The first was to test the performance provided by a system so defined, using a single cheap web camera. The second implements a multiple camera architecture, adopting the multithreading programming concept, and it stands as the first version of the localization system operating software.

The LocSys aims to determine the position and orientation of the TowerSat moving on the Frictionless Table. These measures must meet accuracy and precision requirements. During the tests, the requirements were set to:

- 20 [mm] of maximum absolute error and 2 [mm] of relative error (with respect to the test mean error) for the position
- 2 [deg] of maximum absolute error and 0,5 [deg] of relative error for the orientation

To understand its performance, a single camera was tested, varying the distance and the angle to the target. In addition, different light sources were adopted determining the behaviour of noise on images.

The results showed that the system provides better measurements for the position, reaching an accuracy around 1 [cm] with the optimal set up. Concerning the orientation, some tests led to the fulfilment of the requirements, but the performance can be improved, especially for the measure stability.

Regarding the multiple camera architecture, different configurations have been tested, demonstrating the software ability to adapt to N cameras and to manage the measurements provided by more than one camera that simultaneously see the target.

The correct functioning of the software requires that the cameras are calibrated. For this purpose, various calibration methods have been analysed, both for the intrinsic parameters determination and for the camera pose estimation, with respect to the reference system desired in the Frictionless table System.

A ChArUco board in A3 format printed on forex was adopted for the intrinsic calibration. This pattern allowed to correctly calibrate the cameras using at least 10 photos taken from different angles.

A tool formed by two ArUco markers has been developed for the camera pose estimation. This tool requires high precision by the operator in the extrinsic calibration process. In fact, this phase crucially influences the performance of the entire localization system. In the fourth chapter, all the necessary parameters and measures for its correct use have been defined.

The LocSys described in this thesis has possibilities for improvement on several fronts. Some future developments concern:

- The use of 4 ArUco corners, instead of one, to determinate the position and to average then the results. In this way it is possible to increase both accuracy and precision, at the expanse of few measures per second.
- The implementation of threads for image processing in multi-camera architecture. This would also allow to recover the hundredths of a second required by the previous point.
- An improvement on the orientation evaluation. For example, by taking advantage of the marker pose with respect to the camera and comparing it with the master reference system.
- The use of measurements made on previous frames. Knowing the change rate of position and orientation, it is possible to predict the target behaviour , compare it with the assessments made and consequently improving them.

In conclusion, the thesis is a starting point for the development of a unique localization system based on ArUco markers, capable of achieving high performance, as demonstrated by the tests carried out.

REFERENCES

[1] J. A. Ledin, "Hardware-in-the-loop simulation," *Embed. Syst. Program.*, vol. 12, pp. 42–62, 1999.

[2] M. Romano, D. A. Friedman, and T. J. Shay, "Laboratory experimentation of autonomous spacecraft approach and docking to a collaborative target," *J. Spacecr. Rockets*, vol. 44, no. 1, pp. 164–173, 2007.

[3] A. Mehrparvar *et al.*, "Cubesat design specification rev. 13," *CubeSat Program, Cal Poly San Luis Obispo, US*, vol. 1, no. 2, 2014.

[4] D. A. Friedman, "Laboratory experimentation of autonomous spacecraft docking using cooperative vision navigation," NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 2005.

[5] Y. Zhou, C. L. Law, and J. Xia, "Ultra low-power UWB-RFID system for precise location-aware applications," in *2012 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2012, pp. 154–158.

[6] A. Kaehler and G. Bradski, *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. " O'Reilly Media, Inc.," 2016.

[7] R. Hartley and A. Zisserman, *Multiple view geometry in computer vision*. Cambridge university press, 2003.

[8] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers," *Image Vis. Comput.*, vol. 76, pp. 38–47, 2018.

[9] R. M. Salinas, "ArUco: An efficient library for detection of planar markers and camera pose estimation." 2019.

[10] "OpenCV Online Documentation." [Online]. Available: https://docs.opencv.org/4.1.1/index.html.

[11] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 11, pp. 1330–1334, 2000.

[12] J.-Y. Bouguet, "Camera calibration toolbox for matlab," *http//www. vision. caltech. edu/bouguetj/calib_doc/index. html*, 2004.

[13] E. R. Davies, *Computer vision: principles, algorithms, applications, learning*. Academic Press, 2017.

[14] L. Wall *et al.*, "Concurrency in Python," p. 2, 2015.

[15] ECSS, "ECSS-E-ST-60-10C - Space engineering - Control Performance." 2008.