Università degli Studi di Padova
Norwegian University of Science and Technology - NTNU

DEPARTMENT OF INFORMATION ENGINEERING
Master Program in Automation Engineering

Master Thesis

Scheduling of Two Real-Time Tasks with Non-Fixed Sampling Rates Modelled on an Unmanned Air Vehicle with Autonomous Navigation and Image Processing Capabilities.

Candidate:
Gloria Gambaretto

Supervisor:
Professor Ruggero Carli

Co-Supervisor:
Professor Sverre Hendseth

December 5th 2016 **Academic Year 2015-2016**

CONTENTS

ACKNOWLEDGEMENTS

É infine arrivato il momento più temuto, quello in cui spendo qualche riga in italiano per i dovuti ringraziamenti. Come chi mi conosce sa, non sono solita a ringraziamenti e vistose manifestazioni di affetto, e mi sono chiesta fino all'ultimo se fosse effettivamente possibile saltare questo passaggio e ringraziare tutti silenziosamente solo nella mia testa. Evidentemente però immagino che a qualcuno di vanitoso faccia piacere sentirselo dire anche per iscritto, quindi *here I am*.
Il primo ringraziamento va naturalmente ai miei genitori, con cui ogni momento insieme si trasforma in litigio ma che per qualche strano motivo mi vogliono comunque a casa con loro. Evidentemente sarà per la mia bravura e velocità nel pulire casa. Ringrazio mia nonna, che si lamenta costantemente (capisco da chi ho preso), ma poi vuole bene un po' a tutti, e il resto del parentado. Tanti tanti baci stellari ai miei amici Alberto, Anna, Francesco, Giorgia, Laura e Niccolò per le serate, le vacanze, i "*Stasera usciamo?*", "*Non so, voi che fate?*", "*Boh, cosa c'è in giro?*", "*Per me va bene tutto.*""*Mi si è rotta la caldaia, stasera salto.*", "*Vabbè dai, ci vediamo settimana prossima*". Baci8 anche agli amici dell'università, in particolare alle amiche del succo succoso Bea, Chiara, Eli e Marta, e a Leo, con cui ho condiviso tutto il trash che internet potesse contenere. Un ricordo va anche all'esperienza che ho fatto a Trondheim e a tutte le persone più o meno meno simpatiche che ho conosciuto, Alessandro, Filippo, Alexandrina e tanti altri. Saremo tutti perennemente legati dall'infinita ricerca delle birre meno costose. Un abbraccio anche alle varie coinquiline, Mariasole (<3), Izzy, Rosy e Giusy. Un ringraziamento a tutti i miei professori ed in particolare al Professor Carli ed al Professor Hendseth, i miei relatori. Infine, grazie serie tv, grazie streaming, grazie Lory del Santo e grazie internet in generale per essere sempre in grado di farmi trovare una scusa per evitare la luce del sole.
Concludo con una citazione di una mia guru, *Crederci sempre, arrendersi mai.*

ABSTRACT

Control tasks and scheduling problems are usually treated in separate contexts, but when they are implemented in a real-time system their co-design becomes essential, as it will allow a better use of the limited computational resources. This project regards the creation of a scheduling algorithm for two real-time tasks sharing the same Processing Unit. Once a theoretical solution has been developed, we are required to perform experiments and simulations on a real-case scenario. The two tasks involved in the realistic scenario are control-related and data processing-related. The former involves the control of a small unmanned air vehicle, with autonomous navigation skills, and the latter the image processing of photos of the environment underneath. The air vehicle has to adapt its velocity, and thus its sampling period, to the "*interestingness*" of the area it is flying over, while the image processing algorithm has to adapt its execution time on the same input. The two tasks share the same CPU, and a scheduling technique is required to share in a correct way the computational resources between the two jobs.

LIST OF ACRONYMS

CPU: Central Processing Unit;

EDF: Earliest Deadline First;

FOV: Field Of View;

GPS: Global Positioning System;

LOS: Line Of Sight;

NLGL: Non Linear Guidance Law;

PID: Proportional Integrative Derivative;

RM: Rate Monotonic;

RT: Real Time;

UAV: Unmanned Air Vehicle;

VTP: Virtual Target Point;

WCET: Worst Case Execution Time.

1 INTRODUCTION

1.1 APPROACH AND MOTIVATION

Real-time control plays a crucial role in the coordination of dynamics of various systems, ranging from micro-surgery to flight control, and computer controlled systems are often implemented using periodic real-time tasks [1]. This approach can lead to significant over-provisioning of the real-time system, since task periods are determined by the worst case time interval, assuring closed loop system stability, and traditionally these estimations are very conservative [2]. Usually, the objective of *control activities* (to control some process or plant) and the objective of *scheduling policies* (to meet deadlines) are accomplished separately [3]. This approach leads to sub-optimal solutions of both control performances and resources utilization, and makes it difficult to schedule other tasks with secondary importance.
First, control tasks are optimized regardless the computational demands of other tasks. Furthermore, in control design, a controller is designed assuming a *fixed* and *constant* sampling period: in terms of task execution, this means that at run time the controller will execute demanding a constant processing capacity. In this way it is not taken into account the possibility that the controller could take advantage of the available processing capacity, i.e. the controller design does not allow to increase the execution rate to exploit available resources.
Second, scheduling techniques optimize the use of resources regardless the dynamics of the control application, e.g. if the system is in equilibrium a control task may not require the designed execution rate, so also the designed processing capacity. The remaining resources could have been used by other tasks with higher processing demands.
This is why in the last decade the *periodic task model* has been gradually abandoned in real-time systems, in favour of new "*aperiodic models*", where the sampling rate of each task is dynamically obtained from a utilization factor and an index measuring control performances. In this project it is presented a model for the control of tasks in which computing resources and control performances are jointly considered, overcoming the problem introduced before: the combination of the two approaches makes it possible to achieve better results. A number of researches has solved the problem adopting two methods, in which tasks are either *event-triggered* or *self-triggering* controllers [4]. The former solution uses some "asynchronous" event within the control loop that, when violated, triggers the execution of a task; the latter exploits the next control update time basing it on samples of the previous data and on knowledge of the plant: each task determines the release of its next job.

It is however clear that even if these techniques involves "non-fixed" sampling rates, they cannot assume any value, but they still have to be constrained in some range to maintain the system's stability and realistic performances.
The proposed algorithm takes into account the schedulability of two tasks, and is then applied to a *real-case scenario*: the control of an unmanned air vehicle with image processing capability. It needs to be able to adapt its speed (and thus its control sampling period) and the execution time of its image processing algorithm (or, more specifically, its *deadline*) based on the variation of a certain input signal. The air vehicle is assumed equipped with all the necessary sensors required for its control and navigation, and with a camera mounted on its undercarriage, able to take pictures of the underlying terrain, resembling satellite images. In the simulation environment this has been realized by simulating the plane's flight over the maps provided by the software Google Earth. Once a photo of the environment is taken, a function computes its "interestingness" using a combination of descriptors, and based on the value of this variable, that has been called *complexity*, the plane velocity and the time that has to be waited for the next photo are obtained. The velocity is related to the plane control sampling period, while the latter is related to the execution time of the image processing algorithm (the processing of the image $I(t)$ needs to be completed before starting to process image $I(t+1)$). The direction of the air vehicle is decided based on the longest straight line detected in each picture. A fundamental detail is that both tasks shares the same Central Processing Unit (CPU), so a way to allocate in a proper fashion the available computational resources is described by the *scheduling algorithm*.
This example is indeed a good description of the two main characteristics of the proposed algorithm:

1. Differently from what is usually found in literature, only one of the tasks is control-based, while the other involves data processing. In the first task, the controlled parameter is the control sampling period, while for the second task the controlled parameter is data-related, involving performances that can still be commanded by an input. In the real-case example, the first task is represented by the plane control, the second one by the image processing algorithm;

2. In the real-case scenario, the tasks are both event-triggered and self-triggered. The scheduling function changes the value of the parameters for the both task, based on a variable representing the overall system (the complexity) and the available resources, as in a self-triggered model. On the other hand, they are modified and updated only when a certain event is violated, i.e. when the following photo needs to be taken, acting like an event-based model. In other words, it is possible to imagine it as an event-based sampling where the value of the sampling time is computed in a self-triggered model's fashion.

The task scheduling algorithm with the proposed characteristics is presented first for two generic jobs in two situations: tasks with different priorities and tasks with equal importance. Later, these theoretic solutions have been applied to the realistic scenario presented before, and simulations have been made using the software MATLAB®. On the basis of the obtained results in the simulations, conclusions are made on the effectiveness of the proposed scheduling algorithm.

1.2 OUTLINE OF THE THESIS

The thesis is divided in the following chapters.
Chapter 2 covers the necessary theoretical background for the rest of the thesis, using fundamental publications of the field. It will present the notion of scheduling, the control and scheduling co-design approach and the non-fixed sampling rate problem.
A description of an Unmanned Aerial Vehicle is given in Chapter 3, as well as the description of path planning and path following algorithms.
The second task for the realistic scenario is detailed in Chapter 4, with the description of the *complexity* function, and the designing of the *learning function* used as image processing algorithm.
The main theoretical contribution of this project is given in Chapter 5, in the two cases of prioritized and non-prioritized tasks. The proposed algorithm is then applied in both situations to the two tasks of the proposed scenario.
In Chapter 6 more detailed simulations of the theoretical results are made on the realistic scenario. A comparison between the proposed scheduling algorithm with variable parameters, and a similar framework presenting fixed sampling, is eventually pursued.
A brief conclusion and suggestions for future works are included in Chapter 7.

2 RELATED WORKS

2.1 THE SCHEDULING PROBLEM

In any kind of plant, the control system is usually implemented on a micro-processor that uses a real-time operating system [5]. This OS uses multiprogramming to multiplex the execution of tasks in that processing unit. The CPU time is then a shared resource between all those jobs, that compete for its utilization. In real-time systems processes are referred to as *tasks*, and have temporal qualities and restrictions [6]. Three basic properties are associated with a task: the *release time*, or *ready time*, indicates when the task is ready to be executed; the *deadline* designates when it has to finish and the *execution time* shows how much time it takes to run it. A task is called *periodic* when it is recurring and has to be executed every given period. To guarantee that all time constraints and requirements are met, it is necessary the use of a *scheduler* to share the resources. An example of how a scheduler works is represented in Figure 1: three tasks $T1, T2, T3$ are given, each one with its execution time and deadline, respectively $(1,3), (4,9)$ and $(2,9)$. These can all be executed without the expiration of their deadlines.

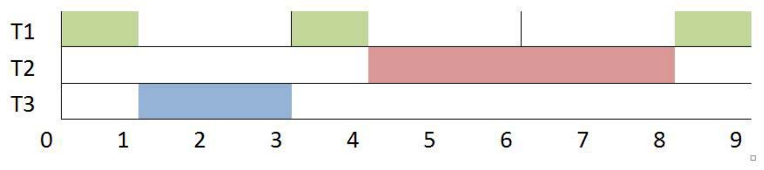


Figure 1.: Example of how a scheduler might work for three given tasks, as in [6].

When tasks are executed on a scheduler a mechanism called *preemption* can occur: this happens when a job that is currently occupying the processor becomes interrupted, its state saved and then it is changed for another task; it will finish its execution later on, always before its deadline. This switching of tasks is known as *context switch*, and every time it occurs it will take a small amount of time, that is however usually not considered[1]. Not every processors support this technique, but it is usually considered existing in a real-time system, where preemption is governed by *priority* [2].

1 The assumption of not considering context switch is justified by the involved time dimensions: when tasks are referred to, the time dimension is *milliseconds*, i.e. 10^{-3} s, while the time taken for context switching is measurable in *microseconds*, i.e. 10^{-6} s, which is three orders of magnitude smaller and therefore negligible.

2 The priority, or weight, of a task is the importance given to it in the context of the present scheduling problem.

During the last decades the CPU scheduling has been a very active research area and a lot of different models and algorithms were proposed. Most of them assume that the tasks are periodic, with period T_i, a known worst-case execution time (WCET) C_i and a *hard deadline*, D_i. All this assumptions can in reality be relaxed because practical systems are composed of tasks with varying timeliness requirements, where only a few of them are hard real-time tasks [3] [7]. Furthermore, the use of WCET as an upper bound works fine only if its value is precisely estimated: in reality, this is not easily achievable because of several low-level mechanisms in modern computers, and thus it introduces a form on non-deterministic behaviour in tasks' execution [8]. In this way, a classical off-line hard guarantee would waste the system's computational resources only to have an absolute certainty of feasibility during sporadic peak load situations, even though the usual workload is much lower. To give an example of this situation, one can consider a visual tracking system where the target is first searched in a small predicted area, eventually enlarging it step by step only if the search was not successful in the previous one. It is easy to see that the worst-case situation is very rare, but much more time and resources consuming, with respect to the ideal case, with the target found immediately in the predicted area. In this case a soft guarantee based on the average execution time allows for a general good behaviour of the software, and infrequent *overruns*, i.e. when a task executes for more than its predicted execution time, can be dealt with in other manners. For example, to prevent these unbounded delays the system could decide to end the current job, or to assign it a lower priority. Another common technique considers an off-line resource reservation, where before running the program, each task is assigned a fraction of the available resources, preventing in this way to use more than its allowed portion, and thus achieving *isolation*. This method can however lead to a waste of resources, if the initial CPU bandwidth allocation is not made in a correct fashion: this is the reason why in recent year the general idea of CPU allocation is still used, applied instead in an on-line manner, with resources fractions computed in real-time based of the actual demand of each task. For example, Caccamo et al. [8] introduced the BASH (*Bandwidth Sharing*) algorithm, that allows to manage overruns in a controlled fashion, performing an efficient reclaiming of the unused computational time through a global bandwidth sharing mechanism. It can also handle tasks with different constraints and criticality without compromising both soft and hard real-time applications.
Another way may be to instead measure the actual execution time during every task invocation, and adjust the parameters involved accordingly.

[3] A real-time system can be catalogued into *hard* or *soft*, based on the consequences of a missed deadline: the former never allows a missed deadline, because it can lead to important failure, which can harm people or the environments, while the latter allows for the missing of some deadlines, causing only degradation of the performances.

Various approach and methods to solve the scheduling problem and other related issues can be found in the literature. For example in [7] Singh and Jeffay focused on the problem of co-scheduling Real-Time tasks and Non Real-Time tasks, specifically the ones whose performance is dependent upon their response time; furthermore, they lack the notion of associated deadline. They solved the problem by scheduling the Non Real-Time tasks first, while still meeting deadlines for the Real-Time jobs: the variability in the execution time of Real-Time tasks was used effectively as a leverage to reduce the response time of Non Real-Time ones. The proposed algorithm exploits the probability distribution theory and uses it as a representation of the variability in execution time of the tasks, using not a fixed processor share but a the notion of *expected processor share* $E[\cdot]$. This approach combines the elegance of shared-based scheduling with the effectiveness of EDL: the Real-Time jobs are initially assigned lesser processor share than the worst case requirement, and then it increases with progress. This may lead to the requirement of the entire processor near the deadline, but the probability that the conditions became so hard is low.

Another important example of real-time scheduling involves its application on automotive systems, where the scheduling of tasks and messages on *in-vehicle networks* is a critical issue for offering good *Quality of Service* guarantees. Today's cars are indeed becoming more sophisticated for the presence of distributed embedded systems where various electronic devices are integrated to replace mechanical elements. These components require an in-vehicle network to communicate in real-time, and in such a way support the execution of their tasks: this network used to rely on an event-triggered protocol, but eventually shifted towards a time-triggered protocol, which grants more predictable and robust communications. Hu *et al.* [9] solved this problem by introducing the *Unfixed Start Time* algorithm, which schedules jobs and messages in a flexible way, such that start times are not fixed; this allows to enhance schedulability, producing a significant advantage when compared to a previous list of similar algorithms. Moreover, to tolerate assignment conflicts due to complex and hard timing constraints, two other methods for rescheduling and backtracking are proposed: the former, called *Rescheduling with Offset Modification*, or ROM, reschedules conflicted applications with an adjusted release time, while the latter (*Backtracking and Priority Promotion*, BPP), used only when ROM does not produce an efficient solution, backtracks a number of previously scheduled applications to create space for the conflicted one. Simulation results shows the better performances of the three combined algorithms compared to prior peer heuristics, especially in terms of schedulability and bandwidth.

2.1.1 Classical Real-Time Scheduling Algorithms

Scheduling algorithms are usually divided into two categories: *off-line* algorithms takes all decision before the system is started, and during runtime the tasks are executed in a pre-determined order. Usually, they are used when a hard real-time system is involved, because the scheduling is studied ensuring that all processes meet their deadlines. On the other hand, *on-line* scheduling takes decisions while the system is running, and is based on priorities, that can be *static*, i.e. fixed a-priori, or *dynamic*, i.e. assigned during runtime [6]. The first algorithms that were used to solve the scheduling problem were the *Rate Monotonic* (RM) and the *Earliest Deadline First* (EDF) algorithms proposed by Liu and Layland in 1973. They scheduled jobs by prioritizing them on their period or deadline, a concept that is associated only with real-time systems, while non real-time systems usually do not have a crisp deadline that they are bound to.

EDF: the Earliest Deadline First can be classified as a dynamic priority driven scheduling algorithm, and it is based on the principle that the task that is closest to its deadline should run first, i.e. the process with the currently earliest deadline during runtime is assigned higher priority. The priority between the tasks are chosen dynamically on-line, because if a task is currently being executed but another task with an earlier deadline becomes ready, then this last one will receive higher priority over the other, and therefore preempts the currently running process. EDF is capable of achieving full processor utilization.

EDL: *Earliest Deadline as Late as possible* (EDL), also known as *Least Laxity First*, LLF, where the *laxity* of a task is its deadline minus the remaining computational time. It can also be described as the maximum time a process can wait before it executes in time to meet its deadline. The higher priority is assigned to the task with the current least laxity; if two tasks has similar laxity they will continually preempt each other, creating many context switches. Calculating an EDL schedule is non trivial as it requires knowledge of arrival times of all RT tasks beforehand.

RM: In the Rate Monotonic technique real-time tasks gets priority over the non real-time ones; unlike EDF, it has a static-priority pre-emptive scheme. The length of a period determines its priority, i.e. tasks with short periods have higher priority. For this algorithm to be used, it is necessary that deadlines length coincides with the period, and that each task is independent from the the other. It can also be extended for cases when the deadline occurs before the end of a period, and in that case it is called *Deadline Monotonic*.
In Figure 2 is presented a practical example of how RM works, taking into account two tasks T_1 and T_2 with execution time and period respectively $(1,4)$ and $(2,6)$: it is possible to see how that T_1 runs before T_2, because it has a shorter period, and therefore higher priority.

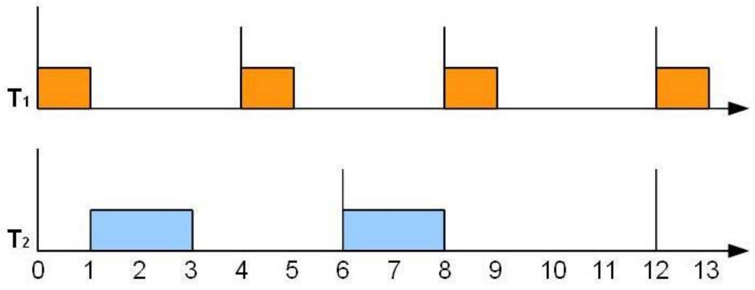


Figure 2.: Example of how a RM algorithm works for two given tasks, as in [6].

BSA: The *Background Server Algorithm* allows the scheduling of periodic and a-periodic tasks in a very simple approach, just giving priority to periodic processes. In other words, a-periodic tasks can run only when the processor is idle, they are executed in the background and are preempted if periodic tasks arrives.

2.2 CONTROL AND SCHEDULING CO-DESIGN

Until a few years ago, the design of controllers and the scheduling of control tasks were considered separately , leading to sub-optimal solutions: the fixed-period assumption of task models was indeed widely adopted by the control community, that used to develop its theory on deterministic, equidistant sampling. Controller design and its digital implementation were built in isolation, as two separate steps, without taking into account that the set of tasks needed for the control may not be schedulable with the limited computing resources available. It also seems unnatural to update the control signal in a periodic fashion, because control tasks should be executed only if - or more often when - something significant happens in the plant that needs to be controlled; the control task hence should not be executed at the same fixed rate regardless of the state of the system. On the other hand, by combining scheduling theory and control theory, it is possible to achieve higher resource utilization and better control performance: this is why the co-design of the scheduler and the controller is necessary. Such a design needs to incorporate information about the available computational resources into the control system, so that optimal solutions can be found [5]. This work on dynamic task adaptation was motivated by multimedia application, such as the one taken into account for this project. They are usually performed periodically, but with less rigid deadlines; an overrun can cause a lower *quality of service*, but without critical consequences for the global system. The possibility to vary tasks' rates could also increase the flexibility of the system in handling this overload situation: for example, whenever a new task cannot be scheduled, instead of rejecting it, the system can try to reduce the utilization of

other jobs to decrease the total load and accommodate the new request.
The first work that specifically addressed this subject was produced in 1996 by Seto et al. [1]: they allowed task frequencies to change within a range as long as such a change does not affect critically the global system. The control law is usually derived based on the properties of the physical system that one is studying, and its digital control algorithm should then be designed to optimize some system *Performance Index* (PI). The new and better *integrated* approach for real-time control systems considers the optimization of the global system performances taking into account at the same time both control performance and the available computing resources: the combination of these two areas opens up to the possibility of higher achievements. This requires knowledge of the relation between control system performance and sampling frequencies, i.e. tasks' period. We would like the frequency to be as high as possible, in order to make a better match between the continuous-time control and its digital implementation, but still imposing for it a certain *upper-bound*, due to the limitation on computing resources when shared among multiple tasks . On the other hand, to make the controller work correctly and to capture in a realistic way the system dynamics, the task period must have also a *lower-bound*, that is usually chosen $5 \div 10$ times the corresponding system's characteristic frequency. These lower and upper bounds allows the task rate to vary within this range, i.e.

$$f_{max} \leqslant f \leqslant f_{min} \qquad \text{or} \qquad T_{min} \leqslant T \leqslant T_{max} \tag{1}$$

In this way task schedulability can be enhanced, which means that one can change the task period in that given set of values so that all the periodic tasks in the set may become schedulable, while in a more fixed approach their original period could made some of them unschedulable. Tasks' period will be adjusted so that they will optimize the overall system performances, taking into account two constraints:

i. The upper bound of tasks period;

ii. The underlying scheduling algorithm and the limitations on available resources.

First, a functional $J(u)$, working as a system performance index, needs to be introduced: this can be interpreted as a measure of the total cost of the control effort generated in a specified time period $[0, t_f]$ by the control signal u. The *optimal control problem* requires to find a control u that minimizes the functional J, keeping in mind that it is subject to the constraints in 1. It is necessary to point out that the functional J do not always have the same meaning: it can measure the control effort, as mentioned before, and in this case it needs to be minimized in order to obtain the optimal control; on the other hand, it can also be designed to measure, for example, the total work the system produces, and in this case it needs to be maximized. The non-linear optimization problem is then associated with the scheduling problem, considering both dynamic assignments, in which case an EDF algorithm will be

used, and static assignment, with a RM algorithm, where simulations show that the former performs usually better than the latter. The conclusions that can be drawn from this first work is that control-scheduling co-design can make schedulable a set of tasks that previously were not because of fixed sampling periods; furthermore, if the set of tasks were already schedulable, the proposed algorithm could be used to improve the global performance.
Until now, we have assumed that all the tasks to be scheduled were control-related, but the same approach can be used for other tasks, e.g. involving data processing or display, because they can be associated with some performance indexes. One of the tasks of this project belongs indeed to this category, because it concerns image processing.

Another pioneering algorithm developed to solve the real-time scheduling-control problem was theorized by Eker et al. [10], and was based on a dynamic feedback from the scheduler to the controllers, and from the controllers to the scheduler, to keep track of the dynamics of the computation workload when this could not be determined accurately. They studied a situation in which the number of control loops and their execution time was not fixed, thus the scheduler must adjust those tasks in order to maintain the optimality of the system, working as a closed loop and changing control frequencies. An example of how the feedback scheduler works is depicted in Figure 3, where the inputs taken into account are the desired workload level (usually a CPU utilization level), the current execution time and the performance of each control loop; clearly, the output is their new sampling frequencies.
A Performance Index is used also in this case to compute how well the algorithm is

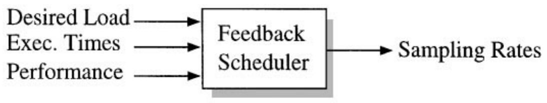


Figure 3.: Example of a feedback scheduler for control loops as in [10].

performing, hence it has to depend on the sampling rate. The most used PI in the literature is linear quadratic, and the controller needs to minimize this quadratic cost function. The feedback is especially useful in overload situations, when the closed loop system intervenes to change the sampling rates and to regain schedulability.
In his work Cervin et al. [11] extended the previous project and used a combination of feedback and feedforward to achieve the scheduling of control tasks: the former was used to measure the execution time, so to maintain a high CPU utilization, avoid overload and distribute in a proper fashion the computational resources, while the latter was interpreted as an admission controller. To be more precise, the introduction of a feedforward action makes the scheduler faster to react to sudden changes in the workload, e.g. when a mode switch condition has been detected or for changing load conditions; furthermore, it improves the regulation of the uti-

lization during mode changes. The behaviour of the proposed algorithm has been tested on an *inverse pendulum* system, where its superior performances over an open-loop scheduling or a simple feedback scheduling has been proved. It also has better results than the ones that can be obtained by using an EDF scheduling, where the controller do not adjust its parameters according to the new sampling period.

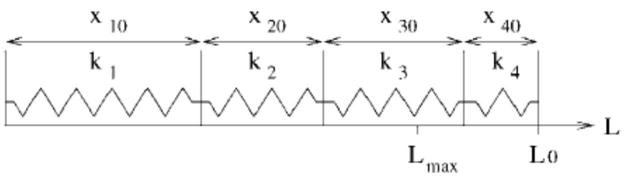
A very important contribution to this field is represented by the work of Buttazzo et al. [12], where they developed an interesting technique called *Elastic Scheduling* that allows periodic tasks to intentionally change their execution rate to provide better quality of service and to keep the system under loaded. This approach also allows for a general resource allocation strategy, not limited to task scheduling, and can be used whenever a resource needs to be allotted to objects with flexible constraints. Also, the elastic approach adapts automatically to changes of load, without specifying the WCET, using real-time and on-line readings of the actual processor utilization.
They developed this technique drawing inspiration from a system of *springs*, each one with a given rigidity coefficient and length constraints: the utilization of the task is treated as an elastic parameter which can take value inside a defined range. Each task τ_i is defined by four parameters in the following way:

$$\tau_i(C_i, T_{i_0}, T_{i_{max}}, E_i)$$

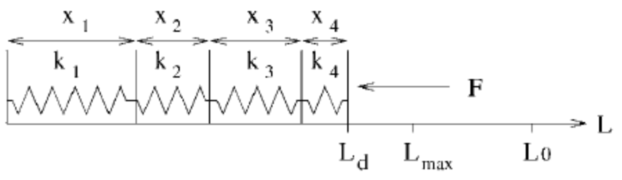
where C_i is the computation time, T_{i_0} is the nominal (minimum) period, $T_{i_{max}}$ is the maximum period and $E_i \geqslant 0$ is the elastic coefficient, that specifies how the task can vary its utilization based on the actual demands. This means that the period T_i has to belong to the range $[T_{i_0}, T_{i_{max}}]$. For a set of n tasks to be schedulable, the only requirement is that

$$\frac{C_1}{T_{1_{max}}} + \frac{C_2}{T_{2_{max}}} + \cdots + \frac{C_n}{T_{n_{max}}} \leqslant 1$$

If this condition is verified it is possible to use a smaller period $T_i \leqslant T_{i_{max}}$, and reach different period configurations that allows schedulability, with the advantage that in the elastic model the best solution is implicitly encoded in the elastic coefficients provided off-line. In this way each task is varied according to its current elastic status, and can be "compressed" to give space to other jobs, or return to their nominal period, depending on the amount of released bandwidth. The equivalence with the system of springs is easily understood: its length x_i can be compared to the utilization factor $U_i = C_i/T_i$ and the rigidity coefficient k_i is equivalent to the inverse of the task's elasticity $1/E_i$. Hence, the set of n tasks with total utilization factor of $U_p = \sum U_i$ can be seen as a system of n springs one after the other, with total length $L = \sum x_i$. An example of how the system works is depicted in Figure 4



(a) Uncompressed springs with total length L.



(b) Springs compressed by a force F with length $L_d < L$.

Figure 4.: A linear spring system, as in [12].

If a force F is applied to the system of spring, the total length of the compressed system is $L_d = \sum_{i=1}^{n} x_i$, and it is possible to find the new length x_i of each spring as

$$x_i = x_{i_0} - (L_0 - L_d)\frac{K_p}{k_i}, \qquad \text{where} \qquad K_p = \frac{1}{\sum_{i=1}^{n} \frac{1}{k_i}} \tag{2}$$

where x_{i_0} represents the nominal length of the i-th spring [4].
A set of elastic tasks can be compressed in a similar way, always keeping in mind utilization constraints, i.e. a task can be compressed up to its minimum period: when this value is reached by one or more tasks, the others will have to adapt and be compressed more than previously computed. Once the overload is over, the tasks may expand up to their original utilization, eventually recovering their nominal periods.

2.3 ON NON-FIXED SAMPLING RATES

As it has been explained in the previous Section 2.2, the purpose of the simultaneous design of controller and scheduler is the optimization of the global system, taking into account computational resources and performance of the controller. The easiest and most effective way to achieve this result is to change the sampling frequency (or the sampling period) of the application that needs to be controlled: the higher the sampling frequency, the more resources it requires, therefore its schedulability becomes more difficult, and vice versa. In other words, a short period allows a quick reaction in front of perturbations, making it good from a control prospective, but it increases the load of the processor, making it bad from a resource utiliza-

4 The procedure to derive the obtained solution is the following: for the equilibrium of the system $F = k_i(x_{i_0} - x_i)$ $\forall i$ from which we derive $x_i = x_{i_0} - F/k_i$. By summing we have $L_d = L_0 - F\sum_{i=1}^{n}\frac{1}{k_i}$, thus the force can be expressed as $F = K_p(L_0 - L_d)$. Equation 2 is its direct consequence.

tion prospective. For example, in an application where a robot has to explore an unknown environment, it will be equipped with proximity sensors that allows it to sense the presence of possible obstacles: these sensors needs to increase their acquisition rate whenever they are approaching an obstruction, in order to maintain the desired performance. The discussion on computation, communication and energy constraints became indeed very important in modern control systems: occupying the CPU for control computations when nothing significant has happened in the process is a waste of resources, in the same way the available communication bandwidth is limited, and its use for sending data in a traditional periodic way is unnecessary. In the literature, two approaches have been highlighted: *event based control* and *self-triggered control*, where sensing and actuation are performed only when needed [4]. They both consist on a feedback controller that computes the control input, and a triggering mechanism that decides when that input has to be updated. The main difference between them is that they are, respectively, *reactive* and *proactive*: the former one generates a sample when a certain even occurs; the latter instead computes the next sample ahead of time.
One of the first examples of aperiodic sampling can be found on the work of Kushner and Tobias in 1969 [13], where they proved the stability of a *random* sampled system, however not taking into account any of the thing mentioned before, such as schedulability or control performance. They studied the case of linear and nonlinear systems with feedback loop, and proved their stability through the use of Lyapunov theory when they were sampled with random periods, with independent holding times.

2.3.1 Event-based control

In an event-triggered control (also known as *interrupt-based feedback* or *Lebesgue sampling*) the triggering condition is based on the continuous monitoring of a certain condition, e.g. the system state deviated more than a certain threshold from the desired value. When this condition is violated, an event is generated, leading to the sampling action. In other words, the event-based implementation of a feedback control law consists in a *holder*[5] that keeps the actuator values constant as long as the triggering condition is not satisfied, and then recomputing that value updating the actuators only when that happens. The asymptotical stability is still guaranteed by Lyapunov theory.
Since the sampling instant are determined at execution time, an a-priori scheduling

5 Let t_k be the time instant of the k-th sample and $u(t_k)$ the value of the input at that time. A holder then maintain that same value until the next $(k+1)$-th sample is been computed. This means that

$$u(t) = u(t_k) \quad \forall t \in [t_k, t_{k+1}[, \quad k \in \mathbb{N}$$

of energy, computational and communication resources can be challenging for this kind of control: the triggering condition can be violated only a few times, resulting in a very low sampling frequency, but it can also be violated very often, with a large consume of resources.
A simple way to perform an event-triggered control is to use a PID controller, as suggested in [14]. The control is divided in two parts: the first one consists of an event detector that uses a traditional time-triggered sampling, while the second one is the PID, which algorithm designs the new input signal; Figure 5 represents the model. The sampling interval of the event detector, h_{nom}, is the same one of the corresponding time-triggered PID controller, which will be used for the final comparisons between the two methods.
The authors decided to use as a triggering event the crossing of a certain threshold

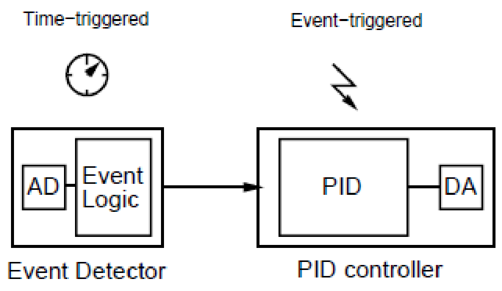


Figure 5.: Event-based PID structure. From [14].

in the error signal instead of the measurement signal: this guarantees sampling also when the set-point changes. Formally

$$|e(t_{k+1}) - e(t_k)| > e_{lim}$$

which means that the new control signal is computed when the absolute difference between two consecutive error values is greater than a certain limit. It is also possible to add the safety condition $h_k \geqslant h_{max}$, i.e. to check that the time elapsed since the last sample does not exceed the limit. The practical effect will be the execution at the higher frequency h_{nom} during transients and at the lower frequency h_{max} during steady state. The code proposed for the event-based PID is very simple and intuitive, and it consist on an algorithm executed at the nominal sampling frequency where all computations for parameters and updates are integrated in an *if*-cycle, performed only if the triggering conditions are violated. This entails a small increase in complexity, however not with drastic consequences. The author then compared the time-triggered PID and the event-triggered one with an equal simulation: the former control algorithm will be executed 600 times over a 10 minutes simulation, while the latter performs the event detection logic 600 times, but the actual control cycle only 103 times. An approximate analysis on processor utilization shows a reduction of about 58% in the used resources.

2.3.2 Self-triggered control

Event-based control has the necessity to keep on monitoring the triggering condition, thus usually requiring a dedicated hardware for this purpose, which is not always possible. The integration of the analogue event detector in the physical plant makes this sampling impractical, and for this reason the more pragmatic approach of self-triggered control has been introduced: here the next update time is precomputed based on a prediction, using for example previous data and knowledge on the plant dynamics. This strategy could be viewed as a way to introduce feedback in the design of the sampling rate, in comparison to the open loop strategy of periodic implementations [15]. The main difference is that this method deals with known future periods, because they are the result of the model execution. The objective is the *simultaneous* computation of the actuator values as well as the next update time. It is possible to define a map $\Gamma : \mathbb{R}^n \to \mathbb{R}^+$ determining the triggering time t_{k+1} as a function of the system state $x(t_k)$ at the time t_k, i.e.

$$t_{k+1} = t_k + \Gamma(x(t_k)) = t_k + \tau_k$$

where τ_k denotes the inter-execution time, and it is usually belonging to a range $[\tau_{min}, \tau_{max}[$. The upper bound in particular reinforces robustness of the implementation and and limits its complexity.
This kind of control can be solved as a *minimum attention problem*: "Given the state of the system, compute a set of inputs that guarantee a certain level of performance while maximizing the next time at which the inputs need to be updated". Here the term *attention* can be interpreted as the inverse of the time elapsed between two consecutive update instants, i.e. τ_k.
For example, Velasco et al. [3] presented an algorithm that belongs to this category, when at each control task instance execution the scheduler is informed on when the next instance should be executed, hence adjusting at each run time its timing constraints. The next sample is then dynamically obtained by using a global parameter, the utilization factor, and a local parameter, the control performance. The behaviour of the overall system is depicted in Figure 6.

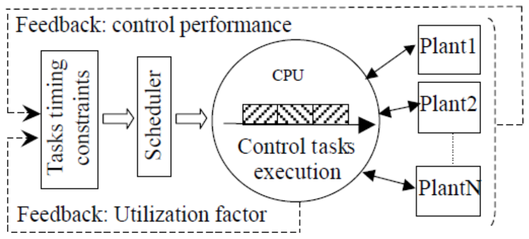


Figure 6.: System operation model for a self-triggered sampling as represented in [3].

The authors propose the use of an extended space-state representation of the system, including as new state variables the task period and the utilization factor: in this way they mix the control behaviour, as the part already present in the state-space representation, with the execution rate of the tasks and the processing demand. In a traditional model the space-state representation resembles the following equation:

$$\begin{bmatrix} x(k+1) \\ y(k+1) \end{bmatrix} = \begin{bmatrix} A_h \\ C_h \end{bmatrix} \begin{bmatrix} x(k) \\ y(k) \end{bmatrix} + \begin{bmatrix} B_h \\ D_h \end{bmatrix} u(k) \tag{3}$$

In Equation 3 $x(k)$ and $y(k)$ are the state variable at the k-th instant, and are usually values coming from sensors; $[A_h \mid C_h]^T$ is the system matrix, describing the dynamic of the plant, and $[B_h \mid D_h]^T$ is the input matrix, linking the inputs u with the system. To be more precise, the matrices A_h, B_h, C_h and D_h are written with a dependency on h, the sampling frequency, as a result of the discretization process, and is usually a constant value chosen during the design stage. Therefore, the choice of h has nothing to do with the system state, but it influences its dynamic. The goal of the paper is to accommodate different values of the sampling period according to the controller dynamics, extending the state representation with a new state variable, the task period, $h(k)$, which is too depending on the time period. The new space-state model can then be represented in the following way:

$$\begin{bmatrix} x(k+1) \\ y(k+1) \\ h(k+1) \end{bmatrix} = \begin{bmatrix} A_{h(k+1)} & 0 \\ C_{h(k+1)} & 0 \\ \alpha \quad \beta & \omega \end{bmatrix} \begin{bmatrix} x(k) \\ y(k) \\ h(k) \end{bmatrix} + \begin{bmatrix} B_{h(k+1)} \\ D_{h(k+1)} \\ 0 \end{bmatrix} u(k) \tag{4}$$

The dependency on the new state $h(k)$ with the other system variables is expressed by the parameters α, β and ω. If $\alpha = \beta = 0$ and $\omega = 1$, so that $h(k+1) = h(k) = constant$, the system is equivalent to the original one in 3. If $\alpha = \beta = 0$ and $0 \leqslant \omega < 1$ the sampling period will be decreasing at each execution, with $h(k) \to 0$, leading to a non-real system; if instead $\omega > 1$ the sampling period will tend to ∞, thus violating Shannon's sampling theorem. The situation the paper is interested in happens when $\alpha \neq 0, \beta \neq 0$ and $0 < \omega < 1$, so that the system has a variable period, each one depending on the previous system state. The resulting transitions in sampling periods are *smooth*, but the resulting space-state model becomes non-linear, where a small input change can cause chaotic and unpredictable outputs. The second step of the proposed model considers the introduction of the utilization factor ζ and of an *h-function*, given by $f(\cdot)$, bounding the possible values of h, instead of having a simple linear relation with the state variables, that could lead to its rapid increase. In this new extension, the model becomes:

$$\begin{bmatrix} x(k+1) \\ y(k+1) \\ h(k+1) \end{bmatrix} = \begin{bmatrix} f_1(x(k), y(k), h(k), \zeta(k)) \\ f_2(x(k), y(k), h(k), \zeta(k)) \\ f_3(x(k), y(k), h(k), \zeta(k)) \end{bmatrix} \begin{bmatrix} x(k) \\ y(k) \\ h(k) \end{bmatrix} + \begin{bmatrix} f_4(x(k), y(k), h(k), \zeta(k)) \\ f_5(x(k), y(k), h(k), \zeta(k)) \\ 0 \end{bmatrix} u(k) \tag{5}$$

In this new extension $h(k+1)$ is obtained by the appropriate function of the state of the controlled system and the processor resources. The focus of the problem in this case changes from the design of α, β and ω to the design of $f(\cdot)$, that determines the system performance and the CPU load. A good way of designing it is to get the system closer to the equilibrium enlarging the sampling period, and decreasing it in case of perturbations, e.g. $h(k+1) = f_3(x(k), y(k), h(k), \zeta(k)) = exp(-x^2(k) - y^2(k))\ g(\zeta(k))$.

In [16] Gommans et al. broadened the field of computational resources saving by including also communication resources saving: in a networked system, the available communication bandwidth is a scarce resource as much as the processor time. Communications should occur only when relevant informations needs to be transmitted from the sensors to the controllers, or from the controllers to the actuators. They studied how to reduce the number of times the input was updated, adopting the self-triggering control, and how this is directly correlated to the number of communications required from sensors to controllers and vice versa. The proposed solution guaranteed the desired performance level and a significant reduction in the utilization of the system's communication resources, with a self-triggered LQR strategy.

3 PLANE CONTROL TASK

3.1 UNMANNED AIR VEHICLE

An *Unmanned Air Vehicle*, or UAVs, is defined as a power driven air vehicle without a human pilot on board, where its flight control is performed automatically through an *autopilot*, without any remote controller. They also have to be *reusable*, and this is why neither missiles nor bombs are considered within this category. The term UAV is generally used with a wider meaning by the public, and it has not be mistaken with *Remotely Piloted Vehicles* (RPVs), that are controlled from an operator in the base station. UAVs should be able to perform some operations autonomously, requiring both *trajectory design* (planning) and *trajectory tracking* (control), that are challenging real-time tasks, given that the on-board computational capabilities are not always able to support complex algorithms. The first UAV was designed in the Fifties by Ryan Aeronautical for military reconnaissance and from that moment their use increased to spare human pilots dangerous or dull jobs. These aircraft are nowadays used in a variety of situations: military applications involve target and decoy, reconnaissance, combat and logistic operations, while civil missions could be weather forecasting, storm and fire detection, aero-biological sampling, mapping, inspection of power lines, traffic assistance. Their advent was helped by their affordability and ease of use, and for their technological increase in the last years. UAVs are usually divided into two categories, *fixed wings* and *rotary wings*: the former have a simpler structure and a more efficient aerodynamics, able to fly for longer distances at higher speeds, while the latter have greater complexity that translates into lower velocities, but they are able to perform vertical take-off and landing, and to hover; they also provide for higher manoeuvring. An example of both can be found in Figure 7. A new type of small UAVs is also becoming more relevant in recent years, i.e. *flapping wings* vehicles: they are inspired by flying insects, and provides many advantages that makes them a bridge between the two previous categories, e.g. simplicity of control in small environments, quiet, high endurance times.

In this work we decided to focus on fixed wing air vehicles, because of the features just presented that better adapts to the kind of civil mission we are interested in. During flight, the performance of the aircraft is influenced by external conditions, like wind, as well as aerodynamics parameters, like design and limited resources. Small UAV, also known as Micro Aerial Vehicles (MAV), usually flies at low altitude, normally less than 300 meters, to provide a precise observation of the ground, which is very important for the missions they are designed for.

(a) Fixed wings UAV.

(b) Rotary wings UAV.

Figure 7.: Example of two Unmanned Air Vehicles.

To represent the motion of the vehicle it is necessary a mathematical model that combines dynamics and kinematics with its aerodynamics parameters, obtaining a 6 Degrees Of Freedom (DOF) model.
In this project, since the main goal is not the model of the aircraft itself, which is just the real-case scenario, but the scheduling problem behind it, we will use only the kinematic equations of the air vehicle, and give just a brief description of the complete model. We will use kinematics to design a control law to command the desired heading rate, assuming constant altitude. The discussion about feedback control for the attitude angles, air speed and altitude, implemented as low-level inner loops, will be left to an *autopilot*, with which it is possible to interact just by changing the sampling rates of those inner loops, providing greater or smaller control performances [17].

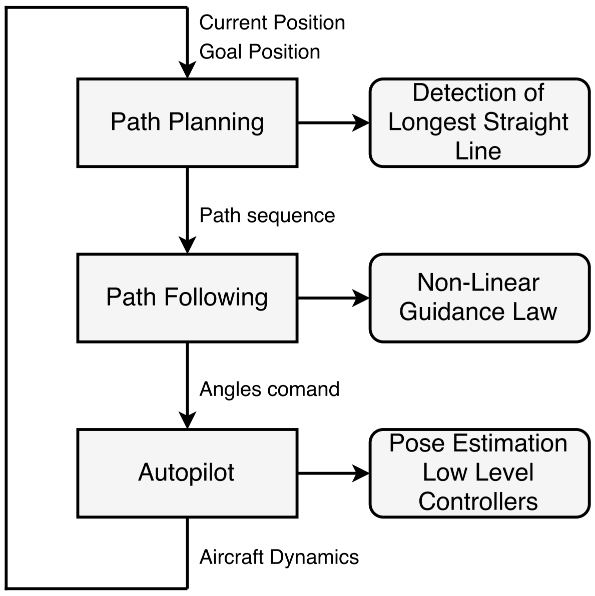


Figure 8.: Block diagram of the proposed division of the task.

The possibility to change the sampling frequency is of particular importance to autonomous robots, since it allows the designer to change not only performances, but also the number of executions of the algorithm. This hierarchical structure is divided into small tasks as represented in Figure 8.

3.1.1 Brief description of dynamics

Consider a *rigid body* in space, its *attitude*, i.e. its geometric description, how that object is placed in the environment. That is expressed in *Euler angles*, also known as *roll-pitch-yaw* angles [18], as represented in Figure 9. A plane can rotate from its center of gravity around three axis (x, y, z); these positional control is then transformed into angular control with the previously mentioned angles (ϕ, θ, ψ).

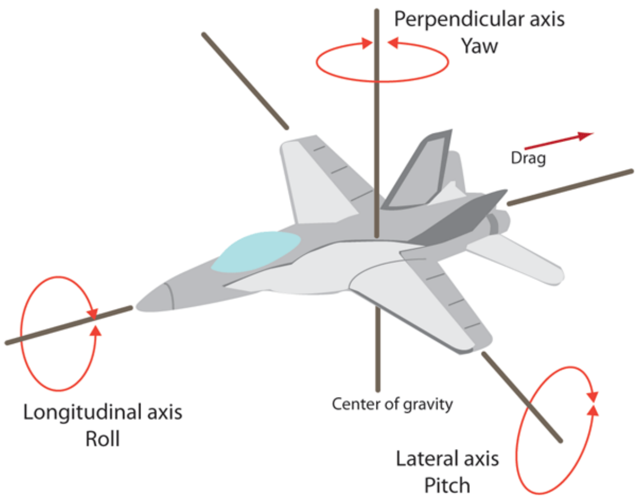


Figure 9.: UAV axes.

The control of a fixed wing aircraft is represented by these three control surfaces and by the thrust given by the engine:

- Ailerons: control of the roll angle;
- Elevator: control of the pitch angle;
- Throttle: control of the motor speed;
- Rudder: control of the yaw angle;

Not all small UAVs have to include the whole set of control surfaces: for example many of them are equipped only with throttle and ailerons, that can be mixed to work as an elevator (in this case they are called *elevons*).
The state variables are position, velocity, Euler angles and angular rate. We can define the position with $\mathbf{d} = [x, y, z]^T$, the Euler angles in the inertial frame with $\boldsymbol{\Theta} = [\phi, \theta, \psi] \in (-\pi, \pi)$, the linear velocity as $\mathbf{v} = [u, v, w]^T$ and the angular velocity as $\boldsymbol{\omega} = [p, q, r]^T$. It is necessary to stress in particular the importance of the angle ψ, called the *heading* of the UAV, since it will be used in the next Section 3.4 as reference

angle for the guidance problem. The heading, or yaw angle, is the one that defines the direction the plane is pointed toward; it must not be mistaken with the *course* angle ψ_G, which is the direction of travel with respect to the Earth's surface. This difference is better explained in Figure 10.

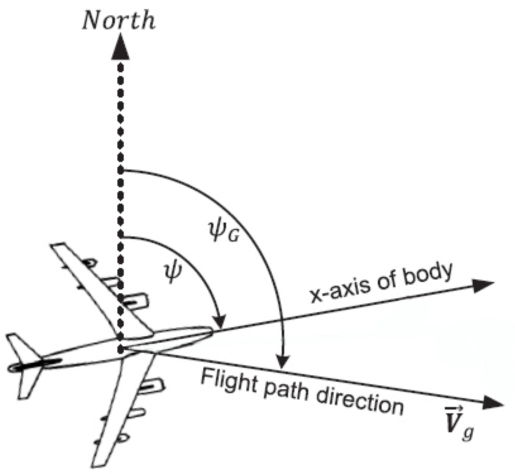


Figure 10.: Difference between the heading angle ψ and the course angle ψ_G.

Since the path that the plane has to follow is referred to way-points with locations on the ground, the guidance problem that we have to solve requires the use of the *ground velocity* V_g.
A discrete version of the nominal UAV dynamics without disturbances is:

$$\begin{aligned} x(t+1) &= x(t) + \Delta T\ V_a \cos(\psi(t)) \\ y(t+1) &= y(t) + \Delta T\ V_a\ sin(\psi(t)) \\ \psi(t+1) &= \psi(t) + \Delta T\ u(t) \end{aligned} \tag{6}$$

where ΔT is the sampling period [19]. An important characteristic that the plane dynamic has to take into account is the presence of a *minimum turning radius*, usually in the range of $10 \div 50$ meters, which is a primary constraint for path-following algorithms.
Another constraint that needs to be taken into account is related to the airspeed of the UAV, that clearly can take values only in a limited range[1].

3.1.2 Plane control through an Autopilot

A robust and stable autopilot system that allows the control of the plane is essential to perform in a good way the tasks. Autopilots are developed to assist a human

1 The terms *speed* and *velocity* are inherently different: speed is the *scalar quantity* that shows how fast an object is moving, while the velocity of such object is a *vector quantity* that refers to the rate at which such object is changing its position. For the sake of simplicity, although, these two terms will be used as synonyms.

operator or to take its place into the guidance of the UAV [20]. Small UAVs can also have another kind of control mode, i.e. Remote Control (RC), that requires a human pilot to interact with the plane through radio signals, but they never reach the enhanced navigation accuracy of an autopilot.
An on-board autopilot is equipped with a micro-controller, sensors and actuators, and usually also with communication devices that allows him to share informations with other vehicles or with a base station. The goal of an autopilot is to consistently guide the aircraft to follow a specified path or to navigate through some way-points. To achieve that it needs to receive from GPS satellite for position updates and to send out control inputs to the motors. It usually consists in two main parts, the *state observer* and the *controller*, as represented in Figure 11

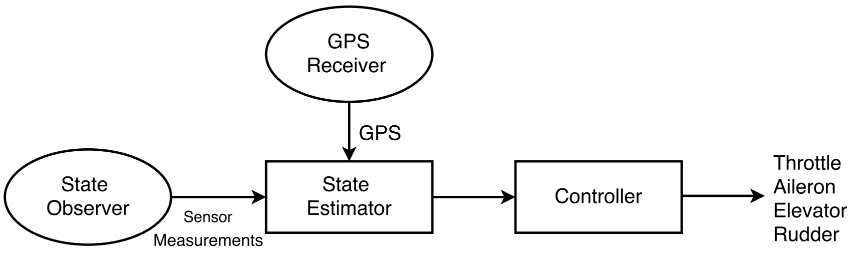


Figure 11.: Functional structure of an UAV autopilot, adapted from [20].

The State Observer is usually an *Inertial Measurements Unit* (IMU), a conglomerate where the sensors are mounted together. It can provide a complete set of sensors reading such as: rate sensors for all three axes, a three-axis magnetic compass to measure roll-pitch-yaw angles, a GPS receiver for the absolute position of the vehicle, pressure sensors for informations about body velocity and altitude, ultrasonic sensors for the relative altitude to the ground, accelerator. An alternative way to obtain these information is to use an *infra-red sensor*: the main idea is to measure temperature difference between two sensors on opposite extremities of each axis to determine the angle, because the Earth emits more infra-red than the sky. Another technique consists in *Vision Sensors*, that performs better in environments where the GPS is not always available; this method is however more used for rotary wings air vehicles for tasks like collision avoidance. Sensor readings are combined with GPS informations and can be passed to a Kalman filter to generate the desired states. The micro-controller on the autopilot carries out different basic functions, such as estimation of the attitude angles, estimation of the absolute position and implementation of the inner loop PID controllers, for attitude. The autopilot includes also an outer cycle on heading, which will be the one we are interested in designing, for trajectory or way-point tracking. Due to the presence of non-linearities in the plane dynamics, a lot of intelligent control techniques have been used to guarantee a smooth navigation, such as PID control, Neural Network, Fuzzy Logic and more.

3.1.3 Kinematics

In this chapter we will assume that a low-level autopilot regulates the airspeed and the heading of the plane to the desired value. For the purpose of a high level path planning and path following control design the kinematic model of an UAV is sufficient [21].
A straight line path is parametrized in the $x - y$ plane, assuming constant altitude; moreover, the aircraft's speed during every time interval τ is also considered constant. Under this assumptions, the plane kinematics can be summarized as:

$$\begin{aligned} \dot{x} &= V_G \cos(\chi) = V_a \cos(\psi) + V_w \cos(\psi_w) \\ \dot{y} &= V_G \sin(\chi) = V_a \sin(\psi) + V_w \sin(\psi_w) \\ \dot{\chi} &= k(\chi_d - \chi) \\ u &= max\_limit(\dot{\chi}\, V_G) \end{aligned} \tag{7}$$

where v_a is the UAV airspeed and V_G is its ground speed: in particular, $V_G = \sqrt{V_{Gx}^2 + V_{Gy}^2}$, with $V_{Vx} = V_a \cos(\psi) + V_w \cos(\psi_w)$ and $V_{Gy} = V_a \sin(\psi) + V_w \sin(\psi_w)$. Furthermore, ψ is the heading angle of the plane, $\chi = atan2(\dot{x}, \dot{y})$ [2] is the course angle, $\dot{\chi}$ is the course angle rate and χ_d is the desired direction toward the goal. V_w and ψ_w are, respectively, the wind speed and direction.
The *lateral acceleration* u is a function of the course angle χ and the ground speed V_G: this function, that we called $max\_limit(\cdot)$, is needed to constrain it, because of the presence of a minimum turning radius, hence the lateral acceleration needs to have an upper bound. This condition is used only in simulation environment.

Algorithm 1 Function `maxLimit.m` pseudocode.

```
1: **Input:** $u, V_G, R_{min}$
2: **Output:** $u_{new}$
3: **if** $u < 0$ and $|u| > V_G^2/R_{min}$ **then**
4:     $u_{new} = -V_G^2/R_{min}$
5: **end if**
6: **if** $u > 0$ and $|u| > V_G^2/R_{min}$ **then**
7:     $u_{new} = V_G^2/R_{min}$
8: **end if**
9: **else** $u_{new} = u$
```

2 $atan2$ is a variation of the classic arctangent function that involves two arguments: this will allow to gather additional information on the sign of the inputs, in order to return the appropriate quadrant of the angle. This data was not available with the original function, whose range is $(-\frac{\pi}{2}, \frac{\pi}{2})$.

If we consider, for the sake of simplicity, a model without wind disturbances, then we have that $V_a = V_G$ and $\chi = \psi$, and the kinematic equations of motions are revised to:

$$\begin{aligned} \dot{x} &= V_a \cos(\psi) \\ \dot{y} &= V_a \sin(\psi) \\ \dot{\psi} &= k(\psi_d - \psi) \\ u &= max\_limit(\dot{\psi} V_a) \end{aligned} \tag{8}$$

3.2 CREATION OF A MAP WITH GOOGLE EARTH

We image the UAV taken into consideration as equipped with a small camera, so that it can take information from the ground below: a camera is a light sensor, compared to the amount of information that it can provide. This is why a camera-based approach is used for the autonomous flight of the aircraft. We will simulate the presence of a downward-looking monocular camera, that provides images with two-dimensional features and unknown depth. To explore and navigate, the UAV will use the images captured by the camera, that are simulated in this project using Google Earth, which is a popular free software that provides satellite images of places around the world.
The initial purpose of this project was to use Google Earth in an interactive way, i.e.create an interface that allows the air vehicle to navigate on-line in the environment images provisioned by the software. Unfortunately the new updates released at the beginning of 2016 made the interaction between Google Earth and the simulation environment used, MATLAB®, extremely difficult. An alternative way to simulate the flight was then chosen, which proved to be much easier, computationally lighter, but still guaranteeing the correctness of its adoption. This alternative method consisted in using a big image, created by saving smaller portions of this map one at a time and successively melting them together. This bigger image of the environment needed to be big enough so that during the simulated flight the UAV could never reach its borders (problem that would never have occurred with the on-line navigation), and needed to have a quality comparable to the one of a camera mounted on the aircraft. The place depicted in the map was chosen in such a way so that it could represent various types of environment with different degrees of complexity: in particular it shows field, with low complexity, and small towns, which center is categorized as having high complexity. The figures were captured at a height of 150 meters, which is a normal altitude for UAVs in order to avoid collisions with obstacles. The air vehicle will clearly not have knowledge of the entire map, because it is exploring an unknown area, but is only able to have visualize a portion of it, i.e. the part that the camera is capturing at the moment. This is the

Field Of View (FOV) provided by the camera, and it will be a square with 150 meters sides, as a real downward-looking camera usually allows. This is represented in Figure 12.

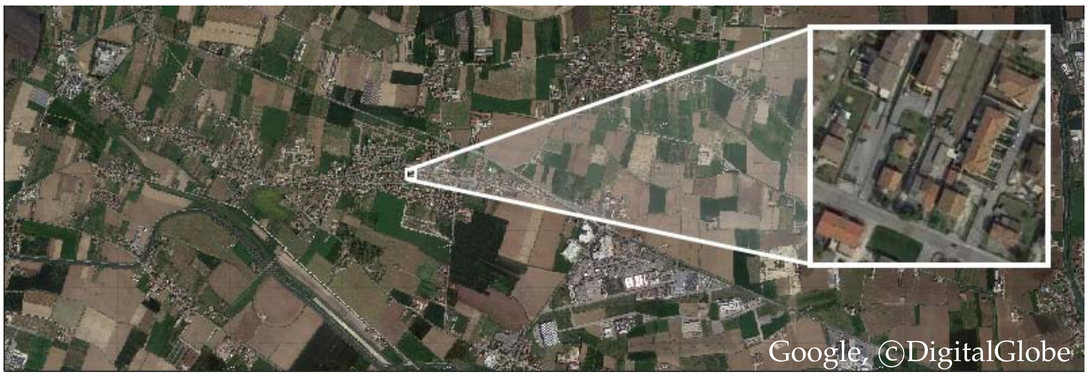


Figure 12.: Complete map of the environment with highlighted an example of the plane's FOV.

Obviously, when the map is uploaded into the programming language, its dimensions will be translated into pixels, and every information about meters will be lost. To avoid such a problem, a conversion parameter pixel/meter needs to be found. It is known, from its extraction from Google Earth, that the map represents a $7500 \times 2880\ \mathrm{m}^2$ area, i.e. an image with dimension 9600×3240 pixels in MATLAB®. From this, it was possible to find out that each pixels corresponds to 0.85 meters, i.e.:

$$n_m = 0.85\, n_p \qquad \text{or} \qquad n_p = 1.18\, n_m \tag{9}$$

where n_m is the number of meters equivalent to n_p pixels. This operation needs to be performed every time informations involving dimensions are required, e.g. the velocity of the UAV, which is expressed in *meters per second* and has to be translated into *pixels per second*, and vice versa: for example, a velocity of 15 $\mathrm{m/s}$ becomes 17.7 px/s, using Equation 9.

An important feature that one has to take into account when dealing with digital images is that, when they are processed by a numerical computing environment like MATLAB®, the origin of the image coordinate frame is in the *upper-left corner*, with the v-axis pointing down[3]. On the contrary, the inertial frame that we are using for the coordinates of the air vehicle is the standard one, with origin in the *bottom-left corner*. For what concerns the values on u and x, not additional work needs to be done, but if dealing with v or y axis it is necessary to know that they are shifted and pointing toward opposite direction, as in Figure 13.

3 The name of the axes for the image coordinate frames is u, corresponding to the x-axis, and v, corresponding to the y-axis.

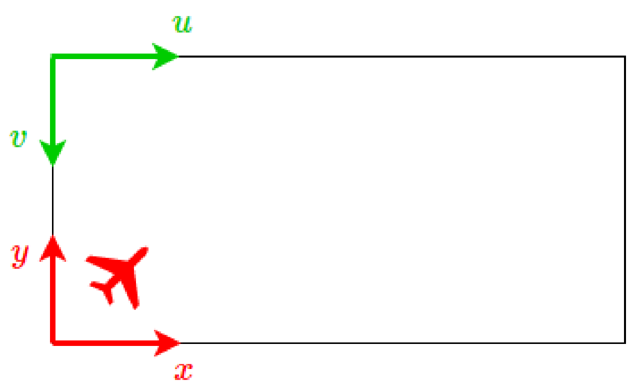


Figure 13.: Different coordinate frames, red for the UAV position and green for the image.

Copyright issues

Google company allows its users to utilize its software with a non exclusive and non transferable license , allowing them to visualize and write on the maps they provide and to publish contents when correctly mentioned. It does not allow its use for selling contents or new products, unless specifies otherwise, or to take inspiration from it to design mapping-related datasets with similar functions. This project does not intend to have any commercial purpose but it is only conceived with an academic intention, therefore the fair use of Google Earth is respected.
Every image exported from the maps has to clearly show Google's and the data provider's name (©DigitalGlobe), with a clear and visible attribution. One can decide whether to use the text directly applied by the application on the imagery or to customize its style and placement, but this has to be done in such a way so that it is as visible and legible as the original one.

3.3 PATH PLANNING

The air vehicle that has been taken into consideration until now is *autonomous*, i.e. it has the ability to perform a task without being remotely controlled by a human operator [22]. They must have advanced path planning and following algorithms, and an effective and robust autonomous navigation system provided by an autopilot, as described in Section 3.1.2.
In this context, a *path* is defined as a sequence of way-points connected by straight lines or arcs, through which the vehicle must traverse. The guidance algorithm then takes the mission plan and the plane kinematics as inputs, and generates the appropriate commands for the control system to track. These are elaborated by the inner control loops in the autopilot, that suitably actuate the control surfaces of the UAV, following the acceleration commands generated by the outer loop. In other words,

in this inner/outer loop design, the outer guidance loop considers the kinematics of the plane, while the inner control loop considers its dynamics. This inner loop is usually multiple time ($5 \div 10$) faster than the outer one, because dynamical variables respond much more quickly in time scale. The goal of the guidance problem is to make the vehicle fly exactly over the lines joining the way-points as projected on the ground picture, with minimum cross-track or lateral deviation.
The first sub-task to solve is the planning of such a path, i.e. decide where the aircraft has to fly. This can be done in multiple ways, based on the mission and the goal that the plane needs to achieve: it could be a very simple straight line or a circumference, without taking into account any feature of the environment, or it can be based on its characteristics, e.g. on following rivers, power lines, oil pipelines. We would like the UAV to move following the roads that it detects in its current view of the underlying terrain: this is a common choice in the literature, since UAVs are mostly used for aerial photography and investigation [23]. We can imagine a road as the longest straight line in the current view, so it can be detected by a *line recognition algorithm*. It is clear nevertheless that a road has also other characteristics to be defined as such, besides being the longest straight line: the color, the texture, the way it is connected with other roads; sometimes it is not even straight, in the case of turns or roundabouts. We can find in the literature many examples of how this problem has been addressed over the years, especially when using satellite imagery. Automated road detection algorithms can be grouped into five categories, as in [24]:

- Ridge finding: use edge operators to find their magnitude and direction, followed by thresholding and thinning to obtain ridge pixels;

- Heuristic Reasoning: uses a-priori knowledge and rules about road characteristics to identify them;

- Dynamic programming: model roads with a set of equation on the derivatives of grey values and use their characteristics to solve optimization problems;

- Statistical Inference: model linear features as Markox processes or stochastic models on roads width, colour, direction and background, then use maximum *a-posteriori* probability to detect road networks;

- Map matching: use existing road maps and then upgrade them to the actual road network, based on the assumption that the new ones need to be connected with the old ones.

For example, in [25] the authors used satellite images and special images from the near infra-red range, since they carry significant information on linear structures. They started by identifying some features typical of roads, e.g. thickness, grey level, colour compared to the background, contrast and minimum length, and proceeded with three major steps: line enhancement, segmentation (eliminating successively non-road pixels) and linking (recovering eliminated road pixels, if any). Arafat et al. [26] focus on colour features to extract streets, taking into consideration also noisy images, when misclassification can occur due to shadows, high buildings and illumination. In [27], high-level information are extracted from road maps (retrieved on-line from map services), and then combined with satellite images fro detecting the road network to plan the shortest path: this method takes advantage of prior knowledge of the road map, provided by map developers, to simplify road detection, with high accuracy and low computational cost. However, since the main aim of this project is not the way the air vehicle moves, but it is just a scenario on which to test the proposed scheduling algorithm, we decided to use the simplest solution possible, which is to detect the longest straight line, even if sometimes it could lead to a wrong road detection or to non-efficient solutions in the trajectory design. We evaluated the performances of diverse *line detection algorithms* that MATLAB® makes available, to find out which one is the best one in terms of computational speed and precision in the road detection.

3.3.1 Edge Detection

In computer vision, we call *edge detection algorithms* those procedures that combines mathematical methods which aim is to identify points in an image where its brightness changes sharply or, in general, have discontinuities of some kind [37]. Usually, these occurs with variations in depth, surface orientation, material properties and scene illumination. These points are generally organized into a set of curved line segments called *edges*. Applying and edge detector to an image helps to filter out informations that are considered less relevant, while preserving its important structural properties.
Algorithms for edge detection can be approximately divided into two categories: *gradient based* and *Laplacian based*. The former method detects the edges by looking for maxima and minima in the first order derivative of the image, usually looking for the direction where you have the local maximum value of the gradient; the latter searches for zero-crossing points in the second order derivative, usually the Laplacian. We inspected the behaviour of four algorithms among the most famous: *Canny*, *Sobel*, *Prewitt* and *Roberts* belonging to the first category, *Marr-Hildreth* for the second category. In literature, the most used method is the Canny Edge Detector, because it is the one the usually performs significantly better, unless some preconditions on the image are particularly suitable for another algorithm.

Consider for example the image in Figure 14. We can clearly see tall spikes corresponding to strong variations in the intensity of the image along the horizontal profile at $v = 130$. The very rapid increase over the space of just a few pixels is distinctive of an edge more than any other decision based on the actual value of grey levels.

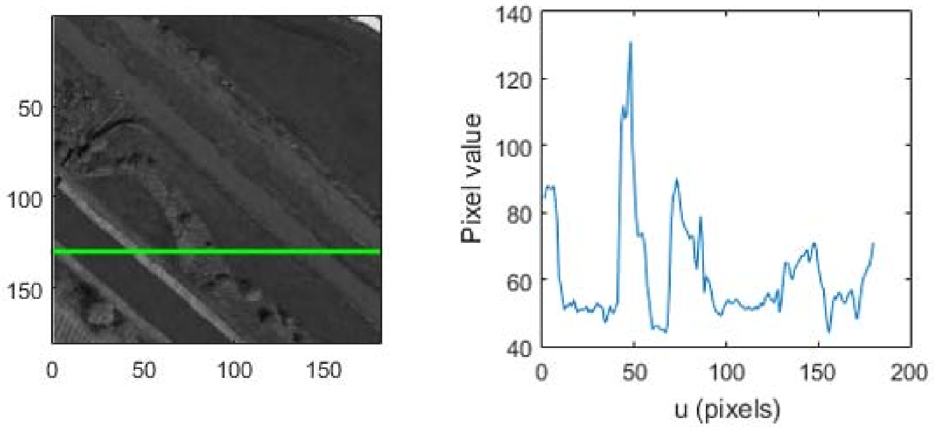


Figure 14.: Horizontal profile of a grey-level image along line $v = 130$.

The first-order derivative along this cross-section is:

$$p'[v] = p[v] - p[v-1]$$

where p is the vector containing the pixel value of the horizontal profile. This signal is nominally zero, with clear non-zero responses at the edges of an object. This derivative can also be written as a symmetrical first-order difference:

$$p'[v] = \frac{1}{2}\left(p[v+1] - p[v-1]\right)$$

which is equivalent to convolution with the one dimensional kernel

$$D = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}$$

Other convolution kernels has been proposed for computing horizontal gradient (to highlight vertical edges), and the most common is the *Sobel kernel*:

$$D_{sobel} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

The results of the application of this kernel is a weighted sum of the horizontal gradient for the current row and the rows above and below. If we want to compute the vertical gradient we just have to use the transpose of this kernel matrix, and in this way we can highlight horizontal edges. We then proceed by computing the derivatives of the image I with respect to the u and the v axis, calling them I_u and I_v respectively. If we call D a convolution matrix such as the Sobel kernel, then we can write the two operations as

$$I_u = D \otimes I \qquad I_v = D^T \otimes I \tag{10}$$

where the symbol $\otimes$ stands for the operation of convolution.

Canny Edge Detector

The Canny edge operator is a very effective and well known edge detector.
Taking the derivative of a signal such as in Equation (10) accentuates high-frequency noise, which is a stationary random process. We can reduce the effect of noise by *smoothing* the image before taking the derivative[4]. Then the equation becomes:

$$I_u = D \otimes (G(\sigma) \otimes I) = \underbrace{(D \otimes G(\sigma))}_{\text{DoG}} \otimes I$$

where, using the *associative property* of convolution, we exploited the *Derivative of Gaussian* (DoG), which can be computed analytically as:

$$G_u(u, v) = -\frac{u}{2\pi\sigma^2}\, e^{-\frac{u^2+v^2}{2\sigma^2}}$$

The standard deviation σ controls the scale of the edges that are detected. This last argument overrides the default Sobel Kernel.
We can afterwards compute the edge *magnitude* of the gradient of each pixel as $M = \sqrt{I_u^2 + I_v^2}$, and its *direction* as $\Theta = atan2(I_u, I_v)$. We then performs two additional steps:

1. **Non-local maxima suppression:** this technique is applied to *thin* the edge. By examining pixel values in a local neighbourhood normal to the edge direction, i.e. the edge gradient direction, we can find the maximum vale, and set all the other to zero. As a result we have a set of non-zero pixels corresponding to ridges and peaks lines;

2. **Hysteresis thresholding:** for each non-zero pixel that exceeds the upper threshold we create a chain of adjacent pixels that exceed the lower threshold; any other pixel is set to zero.

The Canny method differs from the other edge-detection algorithms for this two additional steps, and this is why this method is therefore less likely to be fooled by noise, and more likely to detect true weak edges.

Sobel Edge Detector

The Sobel edge detector does not perform any other additional steps besides the convolution in Equation (10) to compute the derivatives. It then computes the edge magnitude and direction, and returns edges at those points where the gradient of I

4 We call *smoothing* the result of the convolution of the image I with a square kernel $w \times w$ containing equal elements and of unit volume. We obtain an image where each output pixel is the mean of the pixels in a corresponding $w \times w$ neighbourhood. The spread is controlled by the standard deviation parameter σ.

is maximum. We can also specify the direction of detection, horizontal, vertical or both. We can see the different result that we obtain in Figure 15.

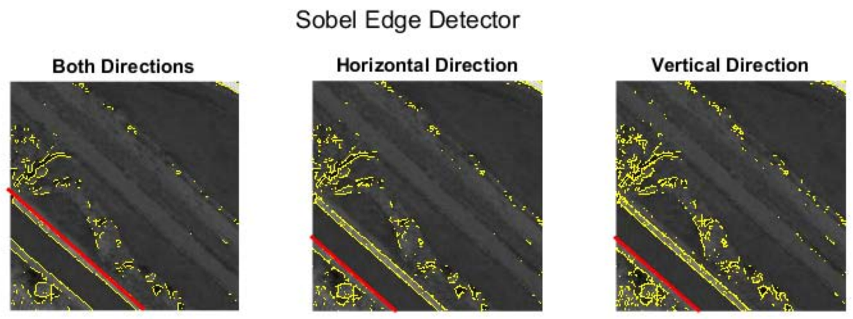


Figure 15.: Results of the Sobel edge detector with different directions of detection. The red line highlights the longest line.

Prewitt Edge Detector

The Prewitt algorithm is structured like the Sobel edge detector, but it uses a different kernel to obtain the first derivative:

$$D_{prewitt} = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{11}$$

Also the results in terms of detected edge are very similar, as one can observe in Figure 16.

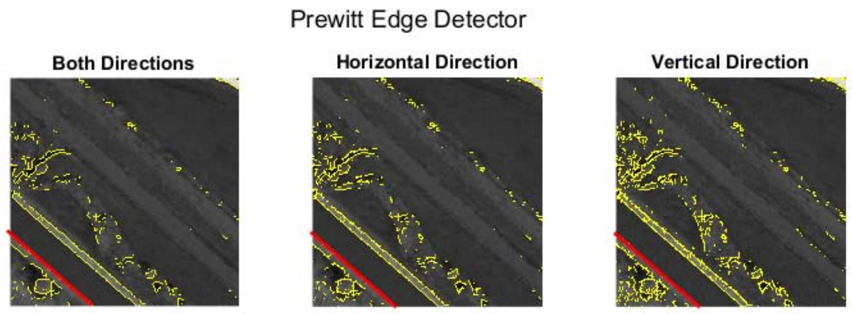


Figure 16.: Results of the Prewitt edge detector with different directions of detection. The red line highlights the longest line.

Roberts Edge Detector

Lawrence Roberts proposed this algorithm in 1963, taking into account some properties that in his vision an edge detector should have: a well-defined produced edge,

little noise in the background, edge intensity should be as close as possible to what a human would perceive.
If we call $x_{u,v}$ the intensity of the pixel in position (u, v), then we can compute its derivative $z_{u,v}$ as:

$$z_{u,v} = \sqrt{(y_{u,v} - y_{u+1,v+1})^2 + (y_{u+1,v} - y_{u,v+1})^2} \qquad \text{where} \qquad y_{u,v} = \sqrt{x_{u,v}}$$

This operation will give as result to highlight changes in intensity in a diagonal direction. This computation can be made easily by convolving the original image I with the following kernels:

$$D_{\text{roberts},1} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad D_{\text{roberts},2} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

We then proceed as in any other detector, computing the magnitude of the first-order derivative and its direction.

Marr-Hildreth Edge Detector

An alternative way of finding points of high gradient is to compute the second-order derivative and determine where it is zero.
First, we compute the *Laplacian Operator* as:

$$\nabla^2 I = \frac{\partial^2 I}{\partial u^2} + \frac{\partial^2 I}{\partial v^2} = I_{uu} + I_{vv}$$

which is the sum of the second spatial derivative in the horizontal and vertical directions. This can be accomplished with the convolution of the original image with a Laplacian kernel, such as:

$$L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

which is *isotropic*, i.e. it responds equally to edges in any direction. This second derivative is even more sensitive to noise, so also in this case we need to smooth the original image with a Gaussian filter:

$$\nabla^2 I = L \otimes (G(\sigma) \otimes I) = \underbrace{(L \otimes G(\sigma))}_{\text{LoG}} \otimes I$$

where on the third equivalence we highlighted the *Laplacian of Gaussian* kernel (LoG), that can be written analytically as:

$$\text{LoG}(u, v) = \frac{\partial^2 G}{\partial u^2} + \frac{\partial^2 G}{\partial v^2} = \frac{1}{\pi\sigma^4}\left(\frac{u^2+v^2}{2\sigma^2} - 1\right) e^{-\frac{u^2+v^2}{2\sigma^2}}$$

This is known as the Marr-Hildreth operator, or *Mexican hat* kernel. This technique allows us to find the maximum gradient when the second derivative is zero, but a significant edge is a zero crossing from a strong positive value to a strong negative value.

Comparison of the algorithms

If we apply to the same image, the one in Figure 14, the five methods that we have explained so far, we obtain the results in Figure 17

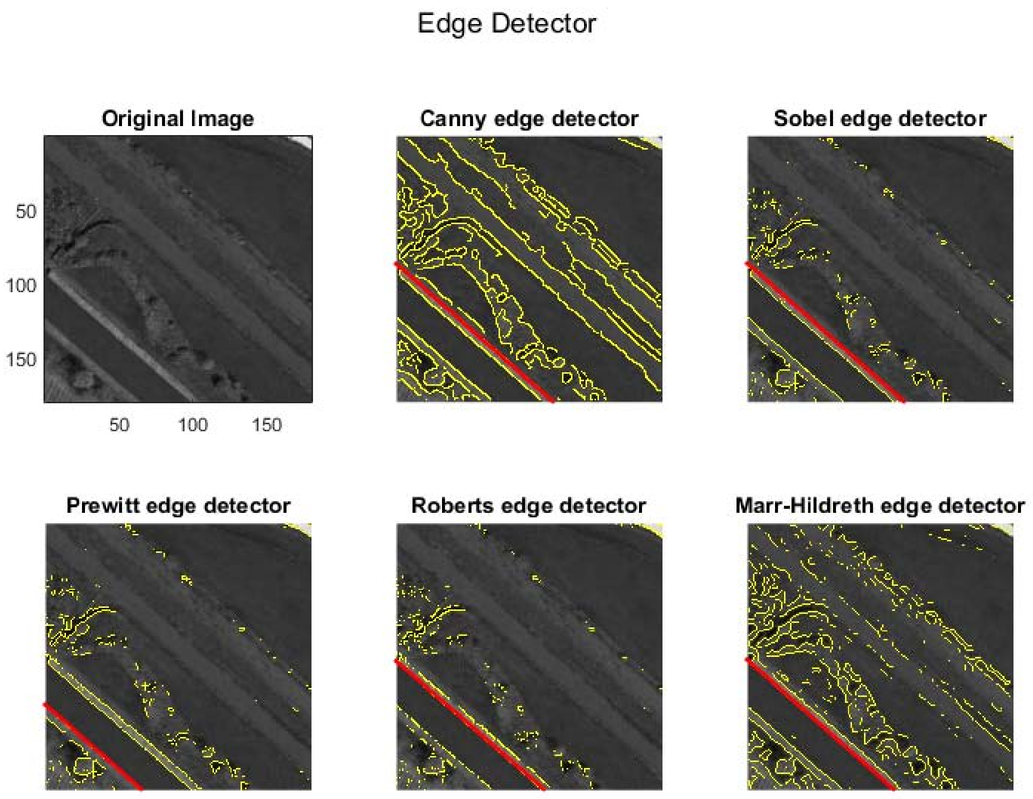


Figure 17.: Results of different algorithm of edge detection. The red line highlights the longest line.

We can see that the Canny algorithm detects a greater number of edges, but almost all the methods detect as longest line the same one, which is the one that actually has the strongest change in intensity in the original image, even though it is not the longest. The only method with a different behaviour is the Prewitt edge detector, that in this case is the one with the "poorest" performance.
We tried to pre-process the image applying a Gaussian filter with different values of the smoothing factor σ, the standard deviation. We can observe in Figure 18 and in Figure 19 the result of the edge detection with this initial smoothing, respectively with $\sigma = 2$ and $\sigma = 4$. With a smoothing factor of $\sigma = 2$ almost all the detector find a smaller number of edges, but choose as longer one the same one as before, except for the Prewitt algorithm, that in this case selects the same line of the other methods. On the other hand with $\sigma = 4$ the Canny detector chooses the line that is actually the longest in the original image, probably because of the removal of some noise through the smoothing. We can also make another observation valid in all cases, that becomes clear by looking at all the images: the Canny algorithm always

finds edges that are well connected among them, smooth and precise. This is why this algorithm is often considered the best edge detector algorithm, conclusion that we can apply also to this project.

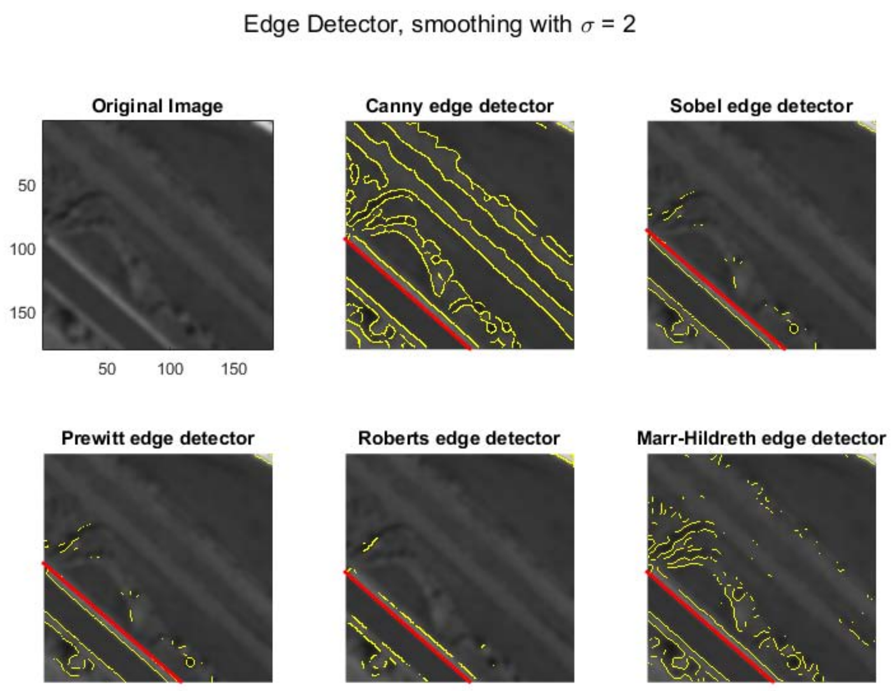


Figure 18.: Edge detection with previous smoothing, σ = 2.

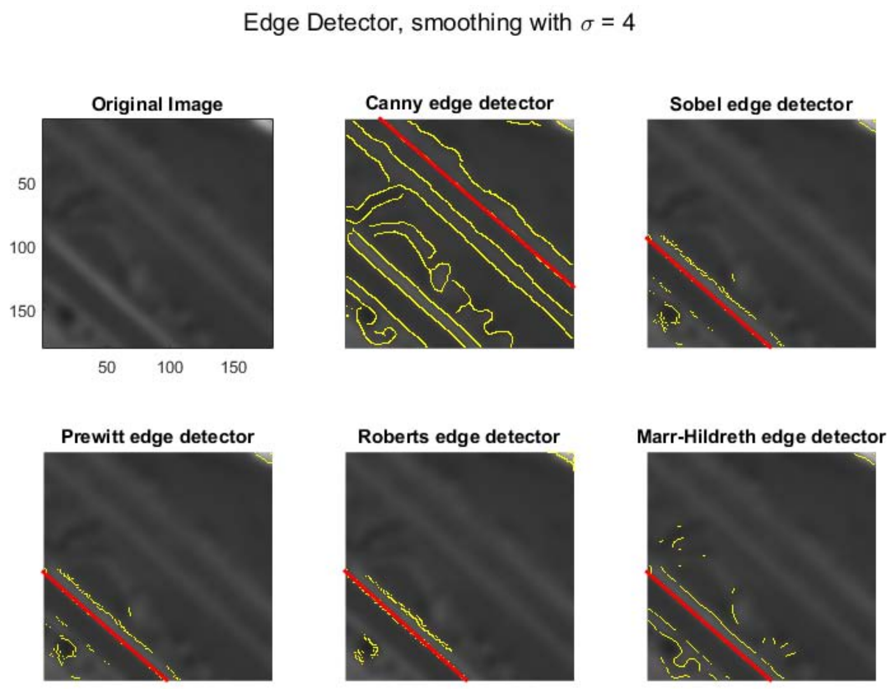


Figure 19.: Edge detection with previous smoothing, σ = 4.

We tried to apply the same methods to different images, with a greater degree of complexity: we can observe the results in Figure 20 and Figure 21.

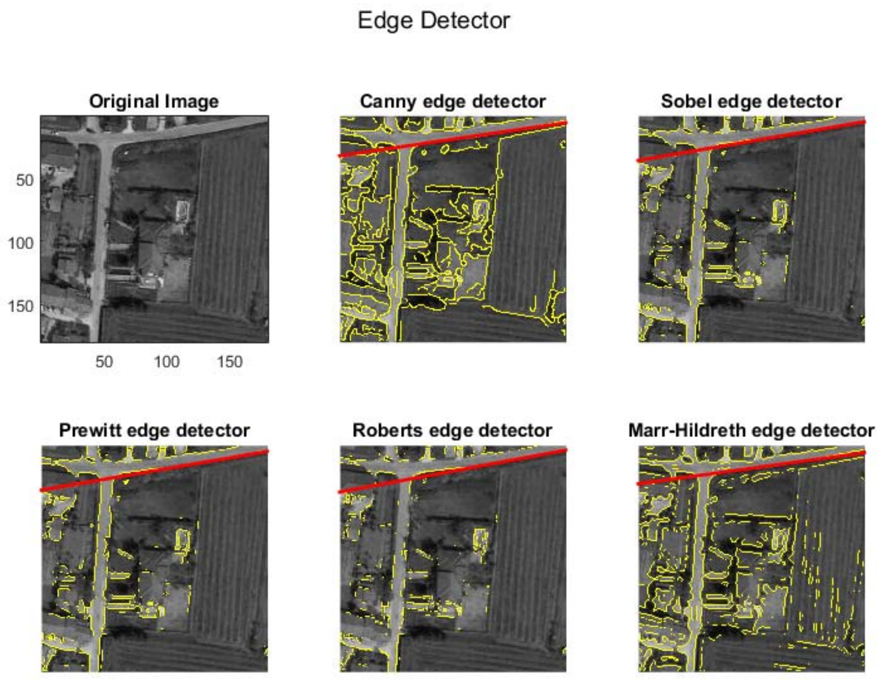


Figure 20.: Edge detection of an image with medium complexity, without pre-processing.

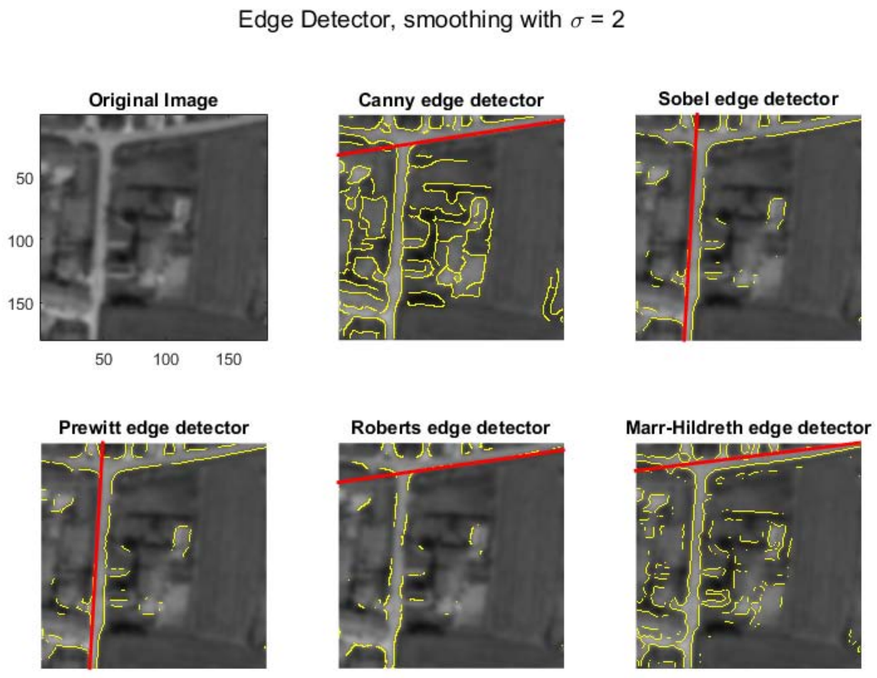


Figure 21.: Edge detection of an image with medium complexity, previous smoothing, $\sigma = 2$.

In particular, in Figure 20 the five methods were applied without any pre-processing on the original image. We can see how all the methods chooses the same strongest line. If we apply a previous Gaussian filter, with smoothing factor $\sigma = 2$, we obtain the images of Figure 21, where three out of five algorithm keeps the same strongest line, while Sobel and Prewitt edge detector find a different line, that is anyway acceptable in this case. If we apply a stronger smoothing factor the results remain almost the same, except for the Roberts algorithm that selects a line that is not acceptable.
If eventually we study an image with high complexity, we will see that all the algorithms performs more poorly than before, because of the great number of edges found: not only streets (which are usually smaller and partially hidden by palaces), but buildings and other elements of an urban context. We can observe this behaviour in Figure 22: no algorithm is able to select the main road in the original image, which is on the bottom left corner, but they all choose other lines that looks almost straight because of the buildings edges. In particular, Roberts method chooses a line that is not acceptable.

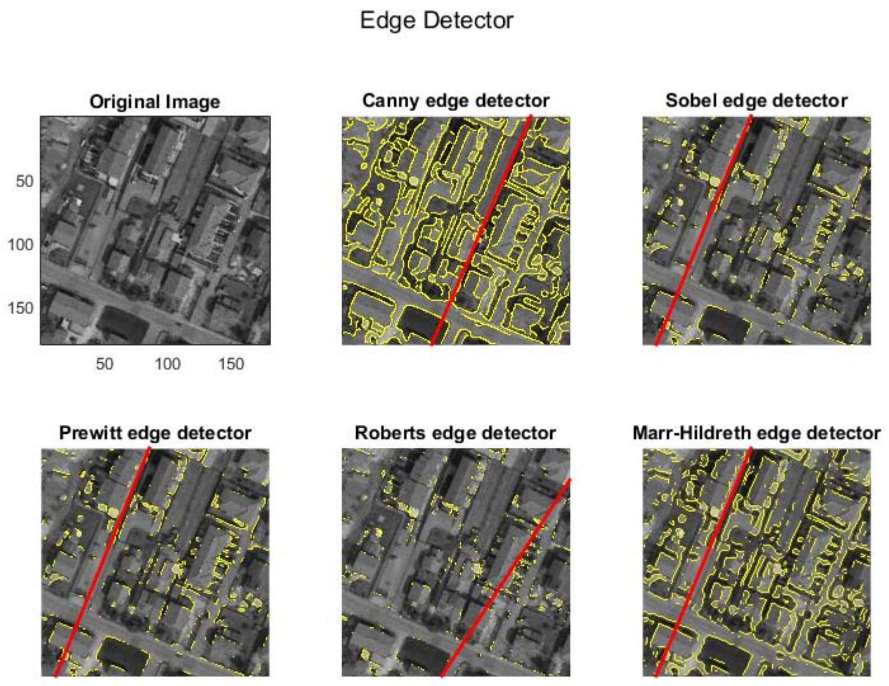


Figure 22.: Edge detection of an image with high complexity, without pre-processing.

We applied again a previous smoothing with $\sigma = 2$. In this case we have the same behaviour of the previous case, where almost all the algorithms can select a line that is reasonable but not an actual road. Similar results are found with a smoothing factor $\sigma = 4$.

3.3.2 Line Features

In all the previous figures we have always highlighted the longest line, but in order to do that we first need to extract some line features, and this can be made through the *Hough Transform*[5].

Consider a point in space and the infinite number of lines that pass through that: if the point could vote for these lines, then each possible line would receive one vote. If we consider another point in the same environments, using the same system to cast votes for all the possible lines that pass through it, one line, the one that connects the two points, will receive two votes. All the other possible lines will receive one or zero votes. We need now to parametrize each line in terms of a minimum number of parameters: in this case is common to use the (ρ, θ) parametrization:

$$v = -u \ \tan\theta + \frac{\rho}{\cos\theta}, \qquad \theta \in \left[-\frac{\pi}{2}, \frac{pi}{2}\right), \quad \rho \in [\rho_{min}, \rho_{max})$$

instead of the classical $v = mu + c$, because it is problematic for the case of vertical lines, where $m = \infty$. In this way each line can be considered a point (ρ, θ) in the two-dimensional space of all possible lines.

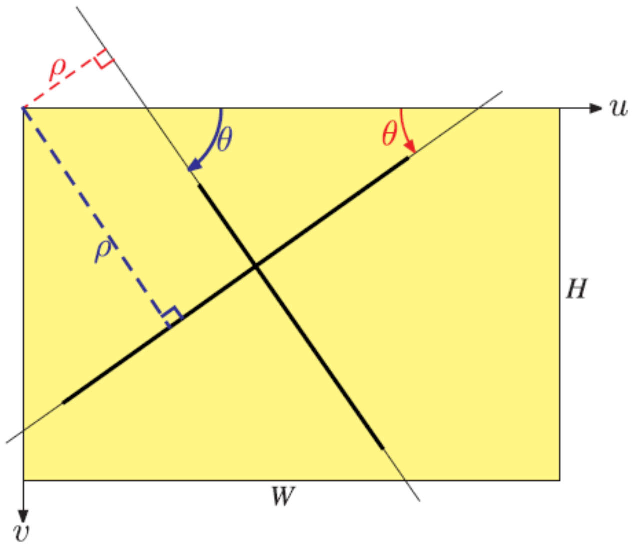


Figure 23.: (ρ, θ) parametrization for two line segments; in blue positive quantities, in red negative quantities.

Hough Transform

If we use in practice the technique described in Section 3.3.2, we cannot consider an infinite number of lines, so we reduce them to a finite set. The $\rho\theta$-space is quantized and a corresponding $N_\theta \times N_\rho$ array A is used to tally the votes: this is called the *accumulator array*. For a $W \times H$ input image we have that $\rho_{max} = -\rho_{min} =$

5 From here on we will use as edge detector the Canny edge detector.

$\sqrt{W^2 + H^2}$. The array A has N_ρ elements spanning the interval $[\rho_{min}, \rho_{max})$ and N_θ elements spanning the interval $\left[-\frac{\pi}{2}, \frac{pi}{2}\right)$.
An edge point (u, v) votes for all the parametrized lines, i.e. all the pair (i, j) for which $\rho = u \sin\theta + v \cos\theta$ holds; in this case the elements $A[i, j]$ are all incremented. At the end of the process the elements of matrix A with the largest number of votes correspond to *dominant lines* in the scene. We can clearly see this in Figure 24, representing the accumulator array: most of the array contains zero votes, and is represented as a black pixel, while the red curves are trails of single voters; they intersect in those points that correspond to lines with more than one vote. The more bright the color is, the greater number of votes that line received; we see several bright spots that are closer together, and this is due to quantization effects. We highlighted in the Figure the five strongest peaks in black, and the strongest among all in blue.

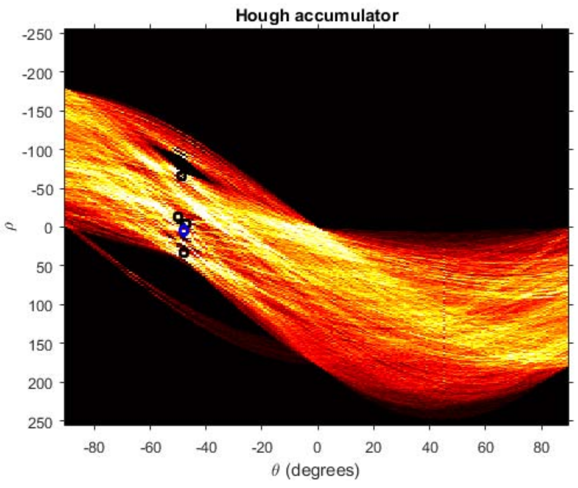


Figure 24.: Hough accumulator array.

We also superimposed to the original image the detected lines, that we can see in Figure 25.

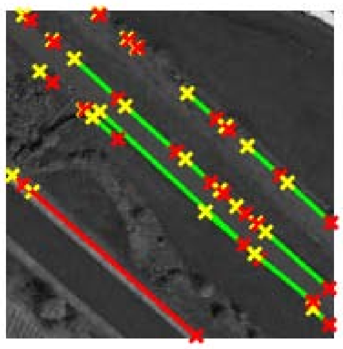


Figure 25.: Lines detected by the Hough transform, in red the longest line.

3.4 PATH FOLLOWING

When performing an autonomous flight for tasks such as mapping, search and rescue, patrol and surveillance, a UAV is required to follow a predefined path at a prescribed height [21]. In this case nonetheless the path is not *a-priori* known, but it is computed on-line and in real-time based on the structure of the terrain over which the UAV is flying, as seen in Section 3.3. Path-following is in general a basic requirement for any kind of unmanned vehicle, and this problem has application in aerospace, underwater and ground robots [28]. In path-following, instead of tracking a *time-parametrized* reference, i.e. a *trajectory*, the vehicle is required to converge to and follow a path without temporal restriction. Examples can be found in literature showing that path-following strategies performs consistently better than trajectory-tracking algorithms, with enhanced results, smoother convergence and less demand on the control effort.
The paths that it is usually required to follow are *straight lines* and *circular orbits*, also called *loiter*. Once a path has been fixed, on-line or off-line, through a certain amount of way-points, a path following algorithm has to be designed: this ensures that the UAV will follow the predefined path in two or three dimensions. In this project only paths with constant altitude will be developed, so we will use algorithms for path following that will concern only two dimensions, because we stated as an initial assumption that the air vehicle would always fly at a constant height, for the sake of simplicity.

3.4.1 Overview on path following algorithms

Once a path has been planned as in Section 3.3, the *path following* problem is to determine the commanded heading angle that accurately tracks the path. In this project the path that the air vehicle has to follow is a composition of straight lines computed and updated in real-time, each one defined by its initial and final points, respectively W_i and W_{i+1} [21]. We define also the *line-of-sight* (LOS) angle θ as the angle formed by the current line that the plane is following with respect to the $x-y$ coordinate frame. The distance d from the vehicle to the path is the *cross-track error*, and this quantity has to be minimized as the UAV approaches the line. In addition to that, the path-following algorithm has to minimize the *heading error* $\xi = |\theta - \psi|$, where ψ is the current heading angle of the air vehicle [6].
The goal of any path-following algorithm is therefore to have the quantities $d \to 0$

6 The notation $|\cdot|$ represents the absolute value, while $||\cdot||$ is used for the Euclidean norm.

and $\xi \to 0$ as the mission time $t \to 0$ [7]. Then, it is possible to write the state vector of the errors as $e(t) = (d(t), \xi(t))^T$, and the error dynamics become:

$$d(t+1) = d(t) + \Delta T \; V_a \; \sin(\xi(t))$$
$$\xi(t+1) = \xi(t) + \Delta T \; V_a \; u(t)$$

In this way the original path tracking problem becomes a regulation problem, driving the state to zero.

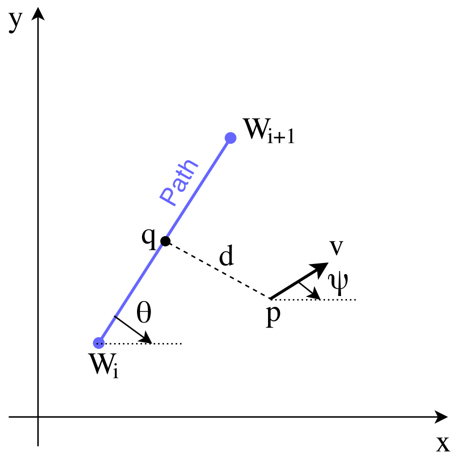


Figure 26.: The UAV located at p needs to follow the straight line defined by the way-points W_i and W_{i+1}.

The path following problem is usually solved using two kind of strategies: *geometric* and *control-based*; the former are widely described in missile guidance literature, while the latter are gaining popularity for their robustness to wind disturbances. Examples for both the techniques are presented below.

- **Geometric Algorithms:** Geometric techniques are commonly based on the concept of *Virtual Target Point* (VTP) on the path, which is an imaginary point moving along the desired flight path as a *pseudo-target*. The vehicle has to chase the VTP, which eventually drives the UAV onto the desired path. These algorithms have performances with high variability based on the chosen *virtual distance* parameter, where the latter is the distance between the VTP and the aircraft position projected on the path (i.e. the distance d in Figure 26), that influences also its stability.

- **Control Algorithms:** Control techniques are very popular for solving path following problems, especially non-linear control algorithms. A common approach is based on a classic (PID) controller, which performances are however

7 The formal definition is the following: *"Let an unmanned aerial vehicle have position $p(t)$, and assume that it has to follow the path $P(\gamma) \in \mathbb{R}^2$, parametrized by $\gamma \in \mathbb{R}$; let P be sufficiently smooth with bounded derivatives. The objective is to define a feedback control law such that the closed-loop signals are bounded, $\| p(t) - P(\gamma(t)) \|$ converges to a neighbourhood of the origin, and the velocity error $| \dot{\gamma} - v_p(\gamma(t)) | < \epsilon$, for $\epsilon > 0$, where $v_p(\gamma) \in \mathbb{R}$ is the desired velocity."*

not satisfying; its extension with feedforward capability is proved to have a better behaviour. Other widely known control-based methods includes linear quadratic regulator, sliding mode control, model predictive control and adaptive control. In literature, many studies on stability and performances guarantee the accurate tracking of the path under different environmental conditions for this class of algorithms.

We focused our attention on the first class, due to its simplicity, robustness and ease of implementation, and chose and compared two algorithms belonging to it: *carrot chasing* and *Non-Linear Guidance Law*. The plainness of these programs must not be associated with poor performances, but it is instead their strength, and makes them very useful in systems with limited resources. For example, carrot chasing algorithm is the most used approach, and it is the default path-following program in the *Paparazzi autopilot*, one of the most common open-source autopilot systems [29].

Carrot Chasing

This algorithm uses a VTP to direct the air vehicle toward the desired path. As the time progresses, the VTP position updates and consequently also the UAV heading direction: in this way the vehicle will move toward the path and asymptotically follow it. In this algorithm the VTP is also called the "*carrot*", hence the name of the technique; in the literature we can also find mentions of this same method under the name "*rabbit-chasing*" algorithm.
When it is required to follow a straight line, this is uniquely identified by two points W_i and W_{i+1}: this notation implies that the plane has to move from the first point to the second one. Assume that p is the location of the UAV, and ψ its heading. We will call q the projection of p onto the LOS at a distance R from W_i and $s = (x_t, y_t)$ the VTP, located at distance δ from q, as shown in Figure 27.

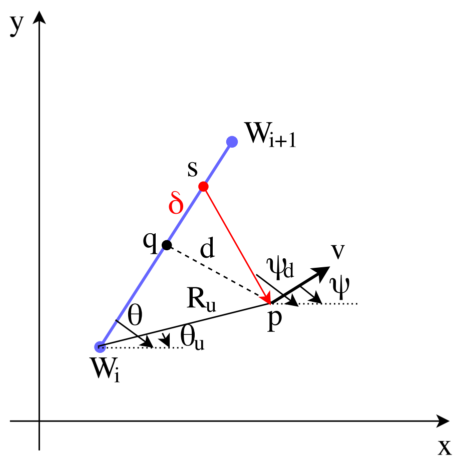


Figure 27.: Straight line path following using the carrot chasing algorithm.

The algorithm can be summarized by these three steps:

1. Determine the *cross-track* error d, which is the distance between p and q;
2. Update the location of the VTP;
3. Update ψ_d and u based on the location of s. In particular, the lateral acceleration is defined as:
$$u = \kappa\,(\psi_d - \psi)\,V_a$$

These three steps are performed iteratively until the UAV reaches the point W_{i+1}. The control input is the lateral acceleration u, and it uses a Proportional controller with gain $k > 0$. The performance of this algorithm is regulated by the value of the parameters k itself and δ. In particular, low values of the latter ($\delta = 0 \div 10$) force the UAV to move directly toward the LOS, resulting in a trajectory that is normal to the path, and then causing the presence of overshoots. Hence, it will take more time to settle on the path, making the cross-track error higher. On the other hand, setting δ to a larger value will direct the vehicle toward the path very slowly, causing again a high cross-track error. It is therefore necessary a good calibration in order to settle the UAV to the path as quickly as possible.

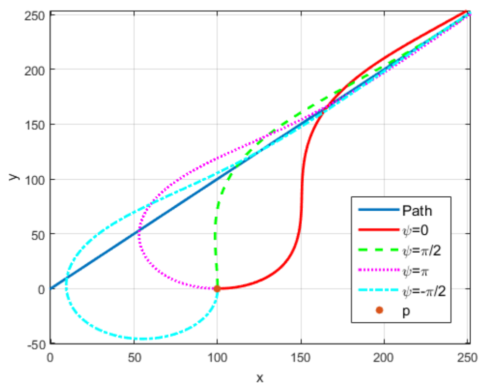
Algorithm 2 Function `carrot_chasing.m` pseudocode.

```
1: Input: $W_i = (x_i, y_i)$, $W_{i+1} = (x_{i+1}, y_{i+1})$, $p = (x, y)$, $\psi$, $\delta$, $V_a$, $\kappa$
2: Output: $u$
3: $R_u = \| W_i - p \|$, $\theta = atan2(y_{i+1} - y_i, x_{i+1} - x_i)$
4: $\theta_u = atan2(y - y_i, x - x_i)$, $\beta = \theta - \theta_u$
5: $R = \sqrt{R_u^2 - (R_u\ sin(\beta))^2}$
6: $(x_t, y_t) \leftarrow ((R+\delta)cos\theta, (R+\delta)sin\theta)$, $s = (x_t, y_t)$
7: $\psi_d = atan2(y_t - y, x_t - x)$
8: $u = max\_limit(\kappa(\psi_d - \psi)V_a)$
```

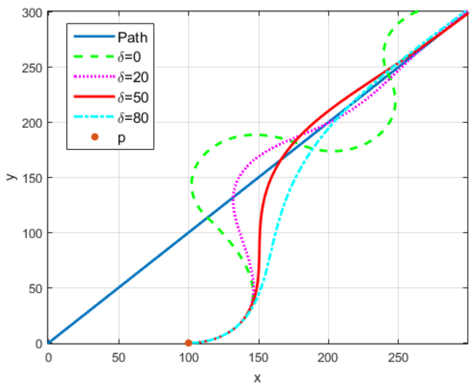
In Figure 28 it is possible to observe some of the outcomes of the algorithm: in particular, in Figure 28a the values of the parameters were fixed, with $\kappa = 0.05$ and $\delta = 50$, and the initial heading angle was varied. In Figure 28b instead the initial heading angle was fixed to $\psi = 0$ (and $\kappa = 0.05$) and we tried to change the value of the parameter δ: it is possible to see the presence of multiple overshoots for low δ, while for high values of the parameter it takes a longer time to reach the path. In every case the chosen velocity for the plane was $V_a = 20$ m/s.

Non-Linear Guidance Law

The NLGL algorithm uses again the VTP concept to make the UAV approach the desired path. It is more flexible that the previous algorithm because it can be applied



(a) Fixed $\kappa = 0.05$ and $\delta = 50$.



(b) Fixed $\psi = 0$ and $\kappa = 0.05$.

Figure 28.: Simulations for the carrot chasing algorithm.

to any type of trajectory, not only straight lines or loiters.
The VTP position $s = (x_t, y_t)$ is determined as the intersection of the path and a circle with radius L, as in Figure 29. Obviously they will intersect in two points, that we will call s and s': we will then choose the one that will represent the VTP based on the direction in which the UAV has to move, which is the one nearest to the way-point W_{i+1}.

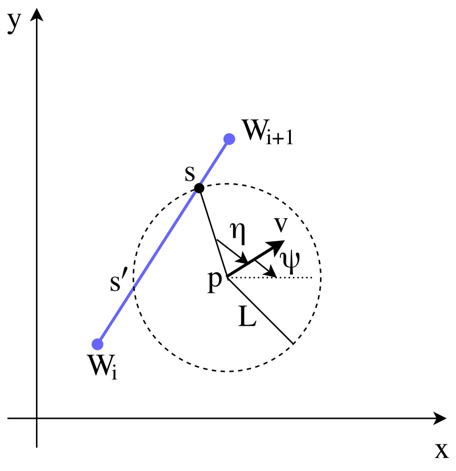


Figure 29.: Straight line path following using the NLGL algorithm.

The lateral acceleration u is generated according to the direction of the reference point, it is relative to the vehicle velocity [30], and it is determined as:

$$u = 2\,\frac{V_a^2}{L}\,\sin(\eta)$$

The direction of the acceleration will align the air vehicle velocity with the direction of the segment $\overline{ps}$; furthermore, if the UAV is far from the path, the algorithm tend to rotate the velocity direction so as to approach the path at a large angle. This guidance law, when following a straight line path, can be linearised and approximated

to a Proportional Derivative (PD) controller[8], where V_a works as the proportional gain, and L as the derivative gain; the ratio L/V_a is the time constant.
The value of the parameter L is chosen constant, with the assumption that it will always intersect the path: if this does not happen the algorithm will have a cross-track error grater than L, therefore producing no VTP and giving $\theta = \pi/2$. This will the vehicle move in the orthogonal direction toward the path, not providing the desired heading command. It is possible otherwise to slightly modify the algorithm by adding a *for* cycle that increases the value of L by a fixed quantity, until it is large enough to return two valid points s and s'. The value of L nonetheless must not be too big: in this case the circle L will always intersect the line, and the VTP will always exist, but it might take a much longer time to converge.
Stability is assured in [30] by a Lyapunov function when the trajectory is circular, and can be extended to the straight line case when the radius approaches infinity.

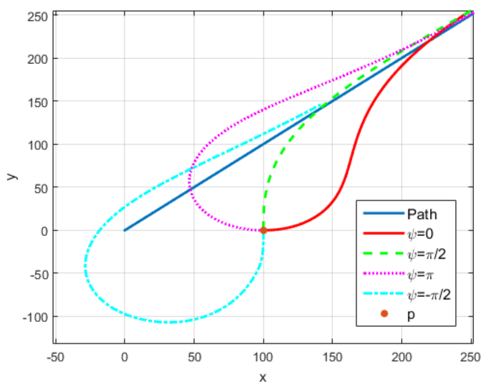
Algorithm 3 Function `nlgl.m` pseudocode.

```
1: **Input:** $W_i = (x_i, y_i)$, $W_{i+1} = (x_{i+1}, y_{i+1})$, $p = (x, y)$, L, $V_a$
2: **Output:** $u$
3: Draw a circle of radius L centred in $p$
4: Determine $s = (x_t, y_t)$ as intersection between circumference and segment $w_i - W_{i+1}$
5: $\psi = atan2(y_t - y, x_t - x)$
6: $\eta = \theta - \psi$
7: $u = max\_limit(2\, V_a^2 \sin(\eta)/L)$
```

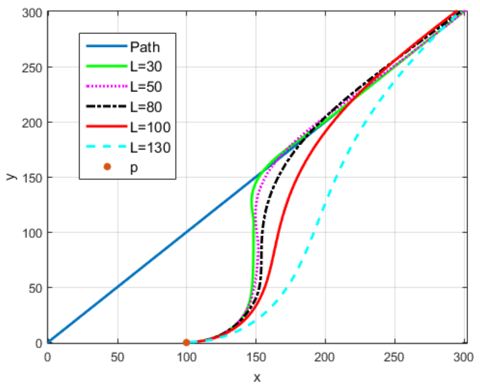
Simulations for the algorithm are reported in Figure 30, with a vehicle velocity of $V_a = 20m/s$. In Figure 30a are presented the results obtained by changing the initial heading angle, and keeping a fixed parameter $L = 100$. In Figure 30b instead we used the same initial heading angle $\psi = 0$ and changed the value of the parameter L, i.e. the radius of the circumference that defines the VTP: when this radius is too small, the VTP does not exist, therefore the final trajectory is not optimal, as mentioned before; on the other hand, high values of L always provide the VTP but takes a much longer time to approach the path.

8 Assuming small angles and straight line path one can derive the formula

$$u = 2\,\frac{V_a^2}{L}\,\sin(\eta) \approx 2\,\frac{V_a}{L}\left(\dot{d} + \frac{V_a}{L} d\right)$$



(a) Fixed $L = 100$.



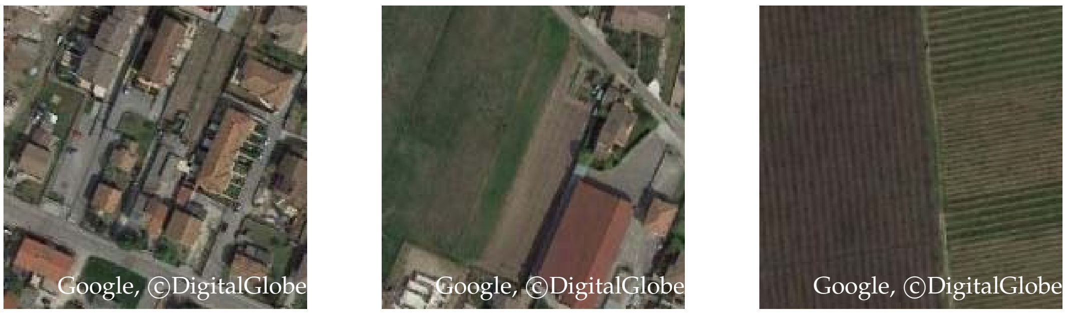
(b) Fixed initial heading angle $\psi = 0$.

Figure 30.: Simulation for the Non Linear Guidance Law algorithm.

4 IMAGE PROCESSING TASK

4.1 IMAGE'S COMPLEXITY

One of the goal of this project is to adapt the velocity of the air vehicle to the underlying environment, making it fly faster when the area is considered not important or not interesting. It is therefore necessary to establish some features and peculiarities that makes an image have a certain degree of *interestingness*, i.e. to decide the quantity of *information* that every image is associated with. With regards to satellite imagery, a particular photo can be considered relevant when it includes a large number of buildings and streets, i.e. when the UAV is flying over a urban conglomerate. On the contrary, a rural area without constructions can be considered less important for the goal and thus requires a smaller amount of image processing. From now on, we will refer to this concept as *complexity*[1] of an image: the higher the complexity, the more interesting the area (e.g. city centres), and the more time it takes to be processed. This means that once the complexity has been evaluated, the system uses that value to generate image-based velocity references to the flight control. An example of three images with *high, medium* and *low* complexity can be found in Figure 31.



(a) High complexity. **(b)** Medium complexity. **(c)** Low complexity.

Figure 31.: Example of images with different levels of interestingness.

It is obviously possible to change the meaning of complexity depending on the purpose of the mission: for example, if the main goal is to find oil pipelines, an image can be considered more complex when it detects a high presence of such pipelines; if the air vehicle is looking for a particular target, the underlying area is evaluated as more important if something in it resembles that target, thus requiring more in-

1 Formally, in literature, complexity has the meaning of *"how much attention is required to detect and recognize objects by a person, and to set relations among them"* [31].

vestigation. In many cases the idea of image complexity is rather used to determine compression levels or bandwidth allocation, and to determine similarities among images.
There are many approaches that one can follow when it comes to decide whether an image can be classified as important or not: fuzzy logic [32], gray level co-occurrence matrix analysis [33], neural network [34]. We tried two methods in this project: the first one involves a *classifier* able to distinguish whether an image has high, medium or low complexity, by using a training set of already categorized images; the second one assigns to each image a value from 0 to 1 (0 being that image not complex at all), using a set of *descriptors* that can expound different features. The former technique proved not to be very efficient, and is illustrated thoroughly in Appendix A, while the latter turned out much more effective and is discussed in Section 4.1.1.

4.1.1 Use of binary descriptors

This technique is inspired by the *Independent Component Analysis* (ICA) developed by Perkiö et al. [35], and it consists on associating to every image a number from 0 to 1, where 0 means that the image has the lower complexity possible, and 1 represents the highest degree of complexity. From this point on, the letter c will be used to indicate the complexity of the image. Such value will be computed as the mathematical mean of different parameters that can be associated to each image. The *descriptors* that has been used are all *binary*, which means they are based on the *grayscale* version of the image, and do not take into account features connected to the color levels.

Entropy

It is a scalar value representing a statistical measure of *randomness* that can be used to characterize the texture of the input image. It was introduced in Shannon's information theory to measure the amount of information in a set of symbols or in an image. Images with low entropy have very little contrast, and large runs of pixels with the same or similar value; an image that is perfectly flat will have an entropy equal to zero. This measure is often used in compression algorithms, because it is a good description of the amount of information that can be coded. This measure is defined as:

$$E = -\sum_i P_i \log_2 P_i$$

where P_i is the probability that the difference between two adjacent pixels is equal to i. It can also be seen as the histogram count of the corresponding intensity image [36].
To decide whether this feature is a good descriptor, we applied it to three sets of images just like the ones in Figure 31, i.e. classifying a-priori some images through

a simple decision made by a human operator. What is expected is to find that all the images belonging to a data set, or at least most of them, will have a similar value of this descriptor. The result of this operation can be observed in Figure 32: the value of the entropy is indeed sufficiently different for the three categories, and only some images fall into an area where images belonging to another categories lie.

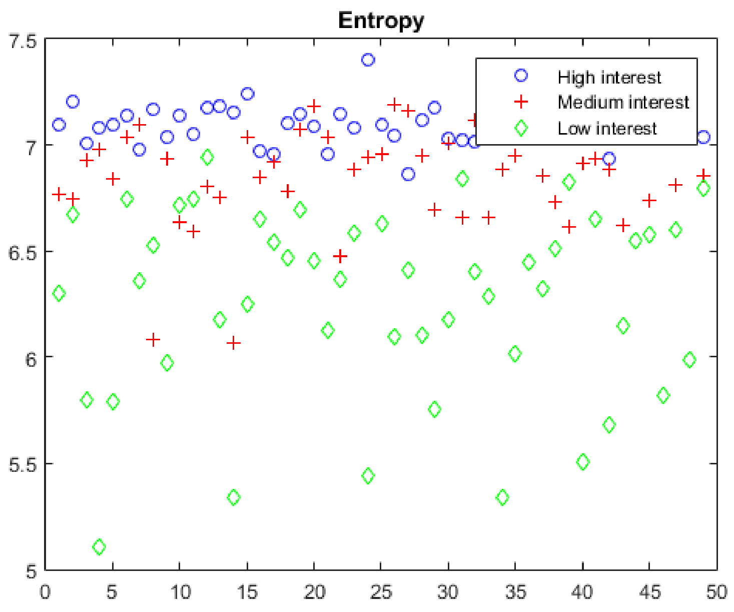


Figure 32.: Value of entropy for three sets of images with different degrees of complexity.

We computed eventually a trimmed mean, i.e. we excluded 10% of the outliers from the computation, leaving out the highest and lowest k data values, where $k = 0.5\ n\ \frac{p}{100}$, with n number of images and p the percentage (in this case $p = 10$). We also computed a normalized mean, where the minimum and maximum value for the normalization are found observing the image: in the case of entropy, for example, we used $m = 5$ and $M = 7.5$, and we obtained a range of values equal to $M - n = 2.5$ [2]. The results can be seen in Table 1.

Table 1.: Average value of entropy for three sets of images with different complexity.

	High	Medium	Low
Real Value	7.0978	6.8676	6.3765
Normalized Value	0.8391	0.7471	0.5506

2 In case of an image with higher entropy than the value M, to avoid having normalized values higher than 1, we cast its value to M. We use the same procedure in case of a value of entropy lower than m, casting it to m.

SURF

We have already discussed briefly in Appendix A the SURF algorithm, used to detect key-points in an image. Theoretically, an image with low complexity is supposed to have a small number of interesting points, while an image with high complexity should have a bigger number of key-points. We can see from Figure 33 that this behaviour is generally respected, as the number of interesting points associated to each image usually lies within a band with images of the same category.

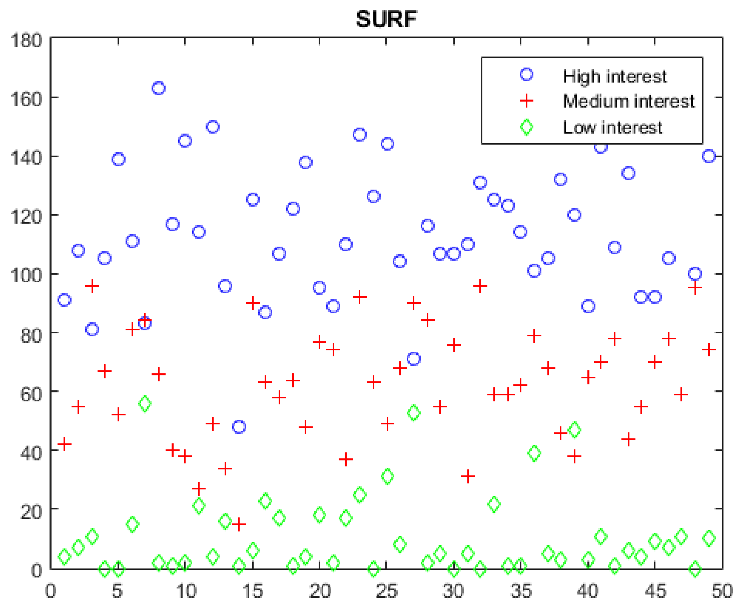


Figure 33.: Number of SURF features for three sets of images with different degrees of complexity.

We can also observe the same behaviour in Table 2, where we can see that the trimmed mean (without 10% of the outliers) of the three categories are well distanced, and thus provide a good measurement for the complexity.

Table 2.: Average number of SURF features for three sets of images with different complexity.

	High	Medium	Low
Real Value	113.9778	62.1778	9.4889
Normalized Value	0.6332	0.3454	0.0527

MSER

The *Maximally Stable Extremal Regions* (MSER) is a feature detector algorithm that extract from an image a number of co-variant regions caller MSERs. This technique was proposed by Matas et al. [38] to find correspondences between elements from two images with different viewpoints. This algorithm is based on the idea of taking regions which stay nearly the same through a wide range of thresholds. The word *extremal* in the name of the technique refers to the property that all pixels inside a region have either higher or lower intensity than all the pixels on its outer boundary [39]. Also in this case, we will expect that a figure with low complexity should have a smaller number of regions detected, with respect to an image with higher complexity, because it does not contain many change in intensity levels. We can see from Figure 34 that this behaviour is respected.

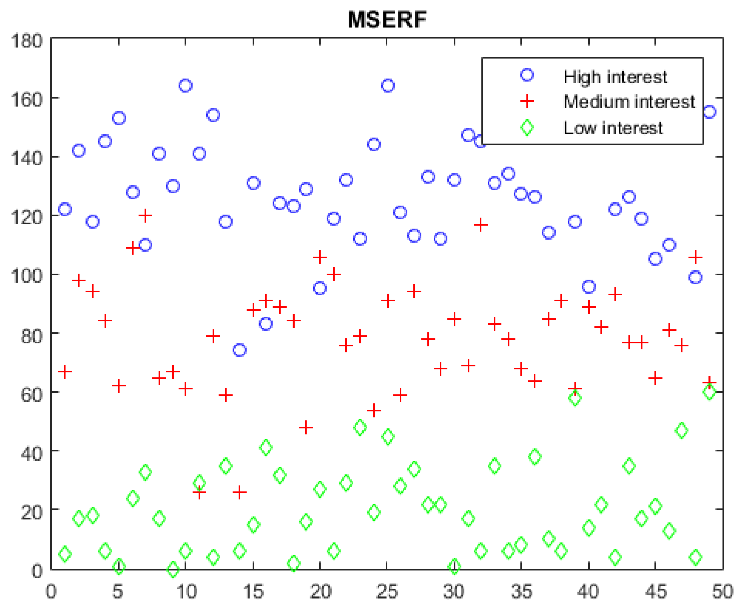


Figure 34.: Number of MSER features for three sets of images with different degrees of complexity.

It is also possible to confirm that the number of MSER's regions is a good descriptor of the complexity of an image by observing the trimmed mean for the three categories in Table 3.

Table 3.: Average number of MSER features for three sets of images with different complexity.

	High	Medium	Low
Real Value	128.2222	79.2000	19.7778
Normalized Value	0.7123	0.4400	0.1099

FAST

FAST (*Features from Accelerated Segment Test*) algorithm was proposed by Rosten and Drummond in 2006 [40], and it is a corner detection algorithm widely used in real-time application for its velocity. FAST corner detector uses a circle of 16 pixels to classify whether a candidate point p is actually a corner. If a set of N contiguous pixels in the circle are all brighter than the intensity of candidate pixel p (plus a threshold value t), or all darker than its intensity (minus a threshold value t), then p is classified as corner. We can observe from Figure 35 that the algorithm works well with images with low interest, having a smaller number of corners detected, while it has some difficulties distinguishing between medium and high complexity images. Anyhow the performances are quite good, and the distance between the mean in each category is high enough, as one can see from Table 4.

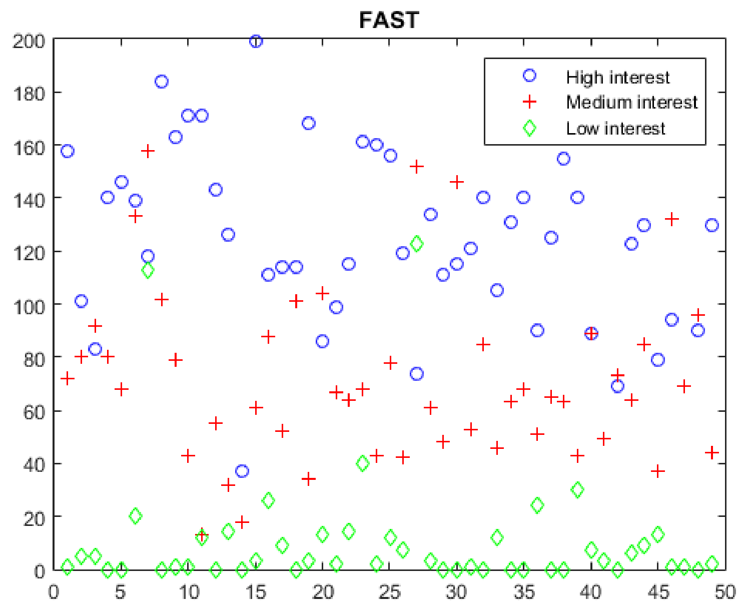


Figure 35.: Number of FAST features for three sets of images with different degrees of complexity.

Table 4.: Average number of FAST features for three sets of images with different complexity.

	High	Medium	Low
Real Value	127.3556	72.9556	7.1778
Normalized Value	0.6368	0.3648	0.0359

BRISK

BRISK, *Binary Robust Invariant Scalable Keypoints*, is an algorithm for the detection of key-point in an image, like the SURF algorithm, theorized by Leutenegger et al. [41]. It is the fastest algorithm for the detection of interest points, while maintaining a high quality description for them. Using this algorithm leads to good results, as one can see from Figure 36 and Table 4: the higher the degree of complexity in an image, the greater the number of key-points found by BRISK.

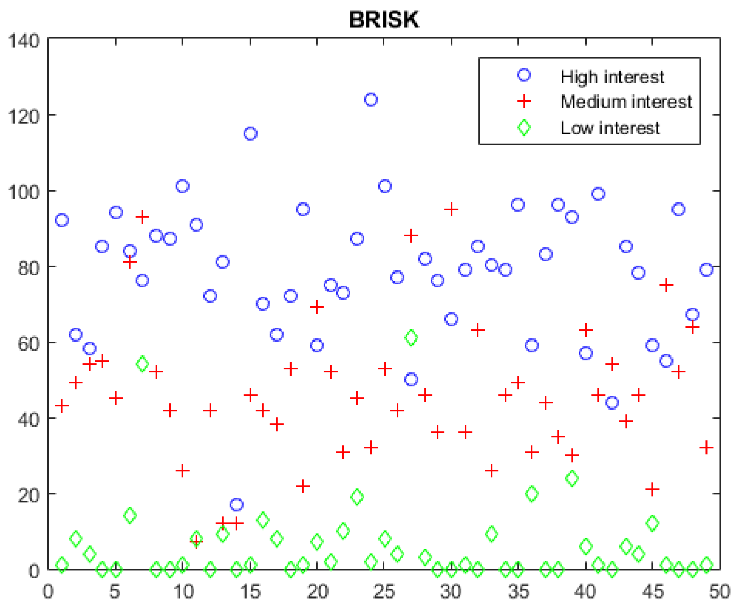


Figure 36.: Number of BRISK features for three sets of images with different degrees of complexity.

Table 5.: Average number of BRISK features for three sets of images with different complexity.

	High	Medium	Low
Real Value	78.6667	47.2000	5.1556
Normalized Value	0.5619	0.3371	0.0368

After normalizing all the previous five parameters, so that their values fall between 0 and 1, we computed their mean, and called the resulting number *complexity* of the image, as already mentioned at the beginning of this Section.

$$c = \frac{E + N_{SURF} + N_{MSER} + N_{FAST} + N_{BRISK}}{5}$$

We eventually obtained the results in Figure 37, where it is easy to see that the three classes under consideration are well distributed, and almost each one of them falls into the correct band of values.

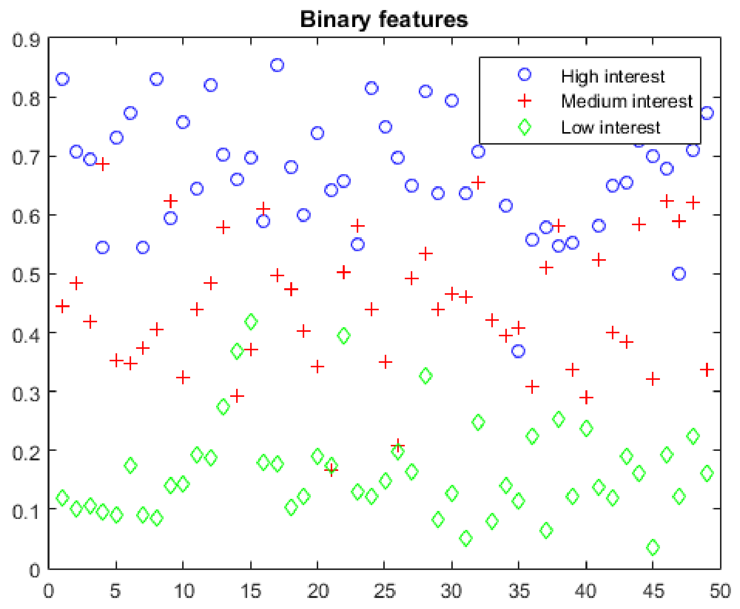


Figure 37.: Average value of all the features for three sets of images with different degrees of complexity.

We can observe the same results in Table 6: the three trimmed mean are well distanced, allowing us to consider this method a good way to measure the interestingness, or complexity, of an image.

Table 6.: Average value of all the features for three sets of images with different complexity.

	High	Medium	Low
Normalized Value	0.6784	0.4481	0.1604

Indeed, if we apply this to a brand new image, representing the current view of the UAV of the underlying terrain, we can attach to that image a number from 0 to 1 representing its interestingness, and use this value to adjust some of its characteristic, e.g. the plane velocity or its sampling frequency.

A final adjustment needs to be made: so far in the description of how to manage the ideal complexity of the image, the execution time has never been taken into account. Because of the limited amount of computational resources that an on-board CPU has, this algorithm needs to be quite fast and simple. All of the features analysed up until now fulfil these requirements, except for the MSER. By using the useful *run and time* tool made available by the MATLAB® software, we were able to measure the execution time of each operation in the program, and to find out that the MSER algorithm itself contributed for more than 50% the total time of the complexity function. For this reason it has been left out from the final writing of the function, which produces anyway an optimal output.

Others binary descriptors

The field of *Computer Vision* is very wide, and make a lot of algorithms to detect relevant features available. To compute the images complexity we only applied five of them, because they are the ones that guarantee the higher performances, in terms of computational time and relevance of the results. For example, we could have used *Harris Detector* for the detection of relevant corners instead of FAST algorithm, but its results are not satisfactory: it takes more time and, as we can see clearly from Figure 38, it does not discriminate properly between images with growing complexity. A great number of pictures belonging to the "Low Interest" class are indeed associated with a high value of complexity, i.e. a high number of corners are identified (this is probably due to some pattern or particular texture present in the terrain).

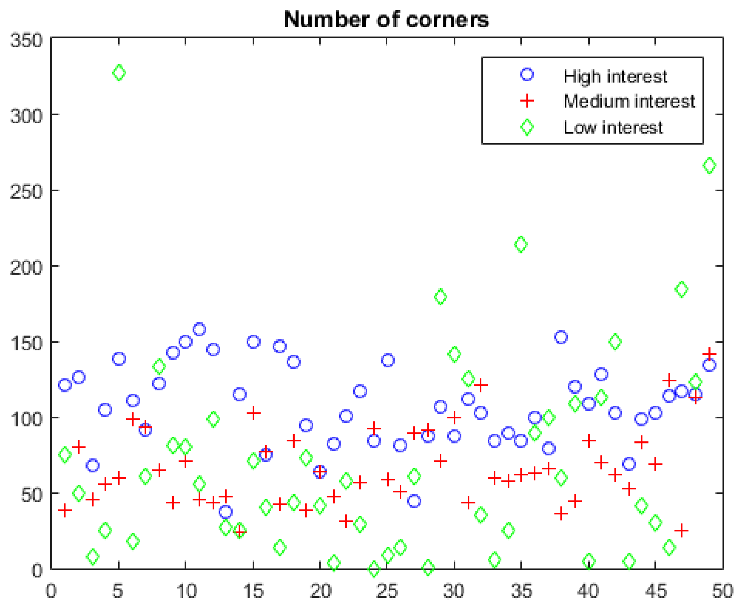


Figure 38.: Number of corners for three sets of images with different degrees of complexity.

From Table 7 we can observe the same behaviour: images belonging to a "Low Interest" class have a trimmed mean greater than the ones belonging to a "Medium Interest" class, which is not the desired conduct from our algorithm.

Table 7.: Average number of corners for three sets of images with different complexity.

	High	Medium	Low
Real Value	107.5556	63.4667	74.5333
Normalized Value	0.5378	0.3173	0.3661

4.2 IMAGE PROCESSING ALGORITHM

The image processing task can have multiple applications, e.g. looking for specific patterns in the territory, tracking some targets, research for damages in power lines, hunt for differences from previous maps of the same place. All these missions are applicable to this project, because they need more time and resources to analyse points with high complexity.

4.2.1 Learning function

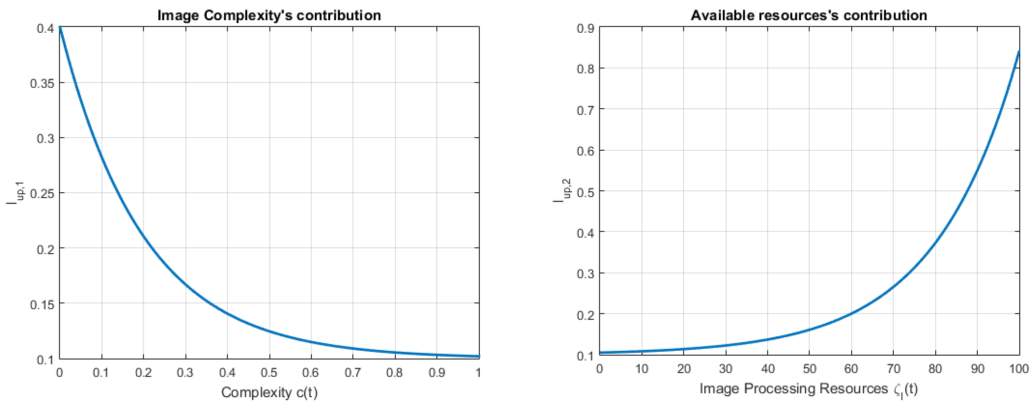
To generalize the concept expressed in the beginning of Section 4.2, instead of focusing on a specific application, a *learning function* $\mathcal{L}$ is used [42]. It receive as input the image $I(t)$ that it has to examine, its complexity $c(t)$ and the computational resources ζ_I that it can use. With this informations it will update a *search map*, which is the representation of the environment. This function will take values in the interval $[0, 1]$, i.e. $l(p, c, \zeta_I) = z$ [3], where $l(\cdot) = 0$ means that the terrain we are flying over is completely *unknown*, on the contrary if $l(\cdot) = 1$ it means that it is completely known and we have all the possible informations about that point. The map $l(p, c, \zeta_I)$ is defined as $l : \chi \times [0, 1] \times [0, 100] \to [0, 1]$, and it is updated on-line based on the value of c and ζ_I, that work as *weights* for the learning function. As the air vehicle moves around in the region, it gathers new informations about the environment, which is incorporated in its search map. In general, this map serves as a storage place of the knowledge that the vehicle has about the region. The use of this method allows some advantages, for example it can be dynamically adjusted with a *forgetting factor* to incorporate changing environments (in case of targets moving around). The search map is represented by a matrix L associated with the image: to each pixel p of the image corresponds an entry in the matrix L, associated to that pixel's value of the learning function, that can be seen as an updating rule for that matrix.
As mentioned before, the function is influenced by the complexity and the computational resources: the former contributes in a *decreasing* manner, i.e. the more

3 $p = (x, y)$ is a point in the search map χ, i.e. the complete map of the region defined in Section 3.2, c is the complexity of the image currently representing the FOV of the plane and ζ_I is the resources available for the image processing task as computed by the scheduling algorithm, and it will be defined in the next Chapter 5. The output $z \in [0, 1]$ corresponds to the certainty about knowing the environment at the coordinate.

complex the image, the more difficult it is to "learn" its features, while the latter contributes in an *increasing* way, i.e. the more resources are available, the more operations can be performed, hence making the exploration easier. For example, it is possible to use two exponential to represent the two contributes:

$$l(p, c, \zeta_I) = L(p) + l_{up,1}(c) + l_{up_2}(\zeta_I) = L(p) + \alpha e^{\beta c} + \gamma e^{\delta \zeta_I}$$

where p is the pixel being examined in that moment, $L(p)$ is the value of the



(a) Function $l_{up,1}(c)$.

(b) Function $l_{up,2}(\zeta_I)$.

Figure 39.: Contributions of image complexity and available resources.

matrix L corresponding to that pixel and α, β, γ and δ are, respectively, magnitude coefficients and decay rates of the two functions $l_{up,1}$ and $l_{up,2}$. Of course, if the value of $l(p, c, \zeta_I)$ is already equal to 1, it cannot grow any more and it must remain fixed. In Figure 39 it is possible to see two examples of how the two contributions $l_{up,1}(c)$ and $l_{up,2}(\zeta_I)$ can be implemented, in particular the functions chosen are:

$$\begin{aligned} l_{up,1}(c) &= 0.3\ e^{-5\ c(t)} + 0.1 \\ l_{up,2}(\zeta_I) &= 0.005\ e^{5\ \zeta_I(t)} + 0.1 \end{aligned} \tag{12}$$

A practical example of how the function $l(\cdot)$ works is here presented, using a matrix L of dimension 5×5 representing the whole map, and one of its sub-matrix of dimension 2×2 representing the plane's *field of view*, highlighted in blue. It is possible to observe that, when the complexity is low or the available resources are high, the learning function assumes a higher value, thus indicating that in those conditions the knowledge of the specific region is increasing faster. Another observation that can be made is the following: when the complexity of the image is high, the learning mechanism requires more time to have a complete knowledge of the area, but that is also the case when the speed of the air vehicle is the slowest, and the time between two consecutive images is the smallest. This means that two consecutive pictures are not very different from one another, because the plane movement was small, so with high probability the same pixel can be analysed more than one time, causing also in this case the increasing of the informations gathered about that area. In the example, the fact that, when the complexity is high, the matrix representing the Field of View of the plane is not moving, expresses this behaviour.

$$L(t) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & 0 & 0 & 0 \end{bmatrix} \qquad c = 0.5, \zeta_I = 50 \Rightarrow l_{up} = 0.29$$

$$L(t+1) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0.29 & \mathbf{0.29} & \mathbf{0} & 0 & 0 \\ 0.29 & \mathbf{0.29} & \mathbf{0} & 0 & 0 \end{bmatrix} \qquad c = 0.3, \zeta_I = 40 \Rightarrow l_{up} = 0.31$$

$$L(t+2) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 0 \\ 0.29 & 0.59 & \mathbf{0.31} & \mathbf{0} & 0 \\ 0.29 & 0.59 & 0.31 & 0 & 0 \end{bmatrix} \qquad c = 0.6, \zeta_I = 54 \Rightarrow l_{up} = 0.29$$

$$L(t+3) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 0 \\ 0 & 0 & \mathbf{0.29} & \mathbf{0.29} & 0 \\ 0.29 & 0.59 & 0.59 & 0.29 & 0 \\ 0.29 & 0.59 & 0.30 & 0 & 0 \end{bmatrix} \qquad c = 0.9, \zeta_I = 55 \Rightarrow l_{up} = 0.28$$

$$L(t+4) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0.28} & \mathbf{0.28} & 0 \\ 0 & 0 & \mathbf{0.57} & \mathbf{0.57} & 0 \\ 0.29 & 0.59 & 0.59 & 0.29 & 0 \\ 0.29 & 0.59 & 0.30 & 0 & 0 \end{bmatrix} \qquad c = 0.8, \zeta_I = 58 \Rightarrow l_{up} = 0.30$$

$$L(t+5) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0.58} & \mathbf{0.58} & 0 \\ 0 & 0 & \mathbf{0.87} & \mathbf{0.87} & 0 \\ 0.29 & 0.59 & 0.52 & 0.29 & 0 \\ 0.29 & 0.59 & 0.30 & 0 & 0 \end{bmatrix}$$

It is of course possible to integrate the environment matrix into the path planning process. When the algorithm detects the lines in each FOV image, one could base the decision not only on the longest one, but also on the one traversing the regions that are most unknown, i.e. with the greatest density of zeros in the matrix L. In the same way, instead of exploring new areas, one can also choose to give the priority to places that are almost completely known (i.e. with a value ≈ 1), by ensuring their complete investigation, and only then move to other regions.

5 CPU SCHEDULING FOR TWO SHARED TASKS

5.1 THEORETICAL DEVELOPMENT FOR TWO GENERIC TASKS

When different tasks need to be executed during the same period, the CPU time constitutes a shared resource for the tasks to compete for: this is why is necessary to schedule its usage [11]. The goal is to assign variable sampling periods to every single task such that the overall performance of the system is optimized, subject to the schedulability constraints[1].

Let's consider n different tasks $\kappa_1, \dots, \kappa_n$ that have to be executed simultaneously and to share the same CPU. Let's also assume that each one of this tasks is robust to the change of its sampling period, i.e. it will maintain its stability; this assumption is proved to be true for a large amount of control tasks. It is clear that the sampling rate, however, cannot assume every value in $\mathbb{R}$, but needs to be constrained between a lower and an upper bound, for the reasons explained in Section 2.2: this means that the sampling period T_i in task κ_i can assume values that lays in $T_{i,min} \leqslant T_i \leqslant T_{i,max}$ (or, equally, one can say that the sampling frequency $f_i = 1/T_i$ has to be in the range $[f_{i,max}, f_{i,min}]$). The chosen time period will affect the CPU resources needed, specified by the letter ζ_i, that can assume values between 0 and 100 [%]. It is assumed that the available computing resources can be divided in the exact way [11].
We theorized two different approaches to follow in such a situation: prioritize the tasks or letting them have the same importance.

5.1.1 Tasks with different priorities

Let's assume that two task κ_1 and κ_2 have to be executed at the same time in a system with a unique processing unit. Without loss of generality, let's also assume that κ_1 has a higher priority than κ_2, i.e. the success of the former task is more important than the performance of latter. This means that, as long as κ_1 performs as it is expected, κ_2 could either be executed at the very least of its possibility or miss its deadline, based on what was a priori decided. In this case the sampling periods will be chosen following the block diagram in Figure 40.
All the functions that will be used from this point will be assumed *continuous* and *invertible*.

1 The effect of control delay and jitter are here neglected.

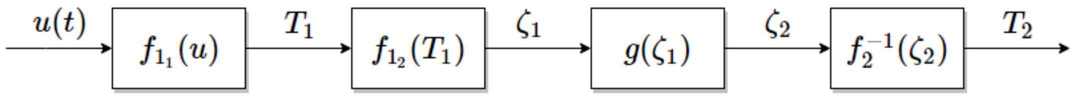


Figure 40.: Block diagram of a task scheduler with two tasks with different priorities.

As one can observe, only the first task will compute its sampling rate based on the system input, through the function $f_{1_1}(u(t))$. It will be able to use as much computational resources as it needs to have exactly the performance requested by the input at time t, without taking into account the necessity of task κ_2. Using a second function $f_{1_2}(T_1)$ it is then possible to obtain the CPU level of utilization for that first job ζ_1. It is important to specify that, while $f_{1_1}(\cdot)$ does not have any a priori restriction and can assume any form based on the input $u(t)$, function $f_{1_2}(\cdot)$ will always be *monotonic* and *strictly increasing*: the more the sampling frequency increases (or the period decreases), the more workload the processing unit will need to sustain. Once the value of ζ_1 is known, it is possible to use the task scheduler to obtain the portion of CPU ζ_2 that the task with lower priority can use. This computation is performed by a function $g(\cdot)$ that takes into account the computational resources still available and decides how much of them can be used to execute the task κ_2. Once the value of ζ_2 is known, it is possible to obtain the sampling period T_2 using the function $f_2^{-1}(\zeta_2)$. The inverse function is here used in accordance to the previous notation: in general, we will use a direct function $f(\cdot)$ that has as output the CPU portion used for the task, while the inverse function $f^{-1}(\cdot)$ will have instead the computational resources as input and the sampling period as output.
It is important to focus the attention on the the function $g(\cdot)$, that works as the actual scheduler for the tasks. This map can assume many forms, based on an a priori decision on the total resources utilization ζ_{TOT} that needs to be achieved.
For example, it may be desirable to always use all the available CPU for the two tasks: in this case it is simply:

$$\zeta_2 = g(\zeta_1) = \zeta_{TOT} - \zeta_1$$

i.e. all the remaining resources will be allocated for κ_2, that in this way can achieve performances even superior to what the system needs. This solution can be used when the optimal behaviour of the system is the main purpose, and we do not need to care about issues like battery consumption.
A second approach, on the contrary, is used when constraints on energy consumption needs to be taken into account, i.e. there is the need to use the least possible amount of resources. In this case then the function $g(\cdot)$ that works as task scheduler depends not only on the available CPU $\zeta_{TOT} - \zeta_1$, but also on some other factor, for example the remaining energy $e(t)$ or the same input $u(t)$ used previously to decide the value of T_1. Furthermore, this method is robust to the presence of noise $w(t)$: if a corruption occurs during the computation of ζ_1 or ζ_2 and their value assumes

higher numbers, the first method could allocate resources $\zeta_1 + \zeta_2 > \zeta_{TOT}$, leading to unpredictable consequences and irregular behaviour of the overall system. On the other hand the second method will have $\zeta_1 + \zeta_2 \leqslant \zeta_{TOT}$, thus having idle resources that can be used in this critical situation.
Neither of the previous approaches take into account that the function $f_{1_2}(T_1)$ could allocate by itself all the available computational resources for the first task. κ_2, having lower priority, could certainly withstand some missed deadline if this situation would only occur a few times, but it is possible to avoid this unsafe circumstance at all. In fact, since the sampling periods that the tasks can assume are constrained between a lower and an upper bound, so is the CPU portion that the task will use:

$$\zeta_{1,min} \leqslant \zeta_1 \leqslant \zeta_{1,max} \qquad \zeta_{2,min} \leqslant \zeta_2 \leqslant \zeta_{2,max}$$

The ideal situation occurs when $\zeta_{1,max} + \zeta_{2,min} \leqslant \zeta_{TOT}$, because in this case the above circumstance of not having enough resources for both tasks can never turn out. In every other situation, a simple solution consists in lowering the value of $\zeta_{1,max}$, and thus of $T_{1,min}$, to

$$\zeta'_{1,max} = \zeta_{TOT} - \zeta_{2,min} \qquad \Rightarrow \qquad T_{1,min} = f_{1_2}^{-1}(\zeta'_{1,max})$$

This will lead to a slightly degraded performance of the first task, but it will help to maintain the overall stability of the system by always providing at least the basic functionality of the second task.
It is clear that, from the beginning of Section 5.1.1, the situations where $\zeta_{i,max} > \zeta_{TOT}$, $i = 1, 2$, or $\zeta_{1,min} + \zeta_{2,min} > \zeta_{TOT}$ has never been referred to, because these are cases where neither task is schedulable, so the general scheduling problem has no solution. The only way to proceed in this case is to cancel some secondary job or to use a more powerful CPU, so that ζ_{TOT} can assume a bigger value.

5.1.2 Tasks with no priorities

The situation where no task has higher importance than any other is now explored; in this case the decision on their sampling time is taken *simultaneously* and not *sequentially* as in Section 5.1.1. The first part of the procedure does not differ from the previous case: from an input $u(t)$ the value of T_1 is calculated using $f_{1_1}(u)$, then we will obtain $\zeta_1 = f_{1_2}(T_1)$. Now, instead of determining ζ_2 based on the remaining resources, its value will be decided independently from the other task. Using the same input $u(t)$, or a different one $\nu(t)$, the value of T_2 and ζ_2 will be computed simultaneously to the first job: $T_2 = f_{2_1}(\nu)$ and $\zeta_2 = f_{2_2}(T_2)$. In this way the obtained results will be the same as if the data for the second task had been computed first, exploiting the fact that neither of them has priority over the other. This procedure can be observed in the first part of the scheme in Figure 41.

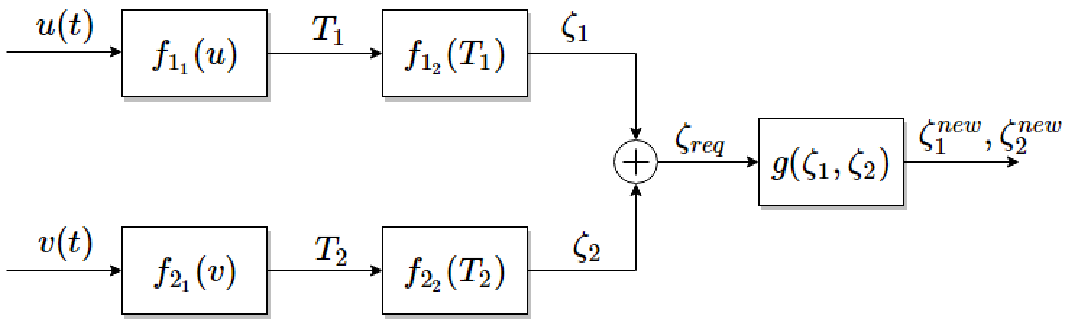


Figure 41.: Block diagram of a task scheduler with two tasks with no priorities.

In the second part of the process the quantity $\zeta_{req} = \zeta_1 + \zeta_2$ is first computed, which expresses the total CPU demand for the correct execution of both tasks, i.e. the *requested* CPU resources. If the obtained result shows that $\zeta_{req} \leqslant \zeta_{TOT}$, we do not need to perform any additional steps on the algorithm, since the situation is already optimal. On the contrary, if the requested resources exceed the total available CPU it is necessary to reallocate part of them through the task scheduler, described by the function $g(\zeta_1, \zeta_2)$, that gives as a result the new values ζ_1^{new} and ζ_2^{new}.

$$\begin{cases} \text{if} \quad \zeta_{req} \leqslant \zeta_{TOT} & \rightarrow \qquad \zeta_1^{new} = \zeta_1 \\ & \qquad\quad \zeta_2^{new} = \zeta_2, \\ \text{if} \quad \zeta_{req} > \zeta_{TOT} & \rightarrow \qquad \begin{bmatrix} \zeta_1^{new} \\ \zeta_2^{new} \end{bmatrix} = g\left(\begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix}\right). \end{cases}$$

When the second situation occurs, we need to reduce the portions of CPU for each task: this will allow, in the end, to obtain $\zeta_1^{new} + \zeta_2^{new} \leqslant \zeta_{TOT}$. It is clear that each task can be slowed down as long as it does not reach one of its lower constraints, i.e. $\zeta_{1,min}$ and $\zeta_{2,min}$.

There are many ways to perform such an operation, with some algorithms already present in the literature, e.g. [12], [3], [16]. In this project, we propose and alternative, simpler and easier approach: first, we compute how much the two tasks together exceeds the total available resources: $\zeta_{ex} = \zeta_{req} - \zeta_{TOT}$; after that, it is possible to split this excess of resources among the two tasks in two ways:

1. Split it *evenly* among the tasks:

$$\begin{cases} \zeta_1^{new} = \zeta_1 - 0.5\zeta_{ex} = \zeta_1 - 0.5(\zeta_1 + \zeta_2 - \zeta_{TOT}) = 0.5\zeta_1 - 0.5\zeta_2 + 0.5\zeta_{TOT} \\ \zeta_2^{new} = \zeta_2 - 0.5\zeta_{ex} = \zeta_2 - 0.5(\zeta_1 + \zeta_2 - \zeta_{TOT}) = 0.5\zeta_2 - 0.5\zeta_1 + 0.5\zeta_{TOT} \end{cases}$$

or, in matrix form:

$$\begin{bmatrix} \zeta_1^{new} \\ \zeta_2^{new} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 \\ -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix} + 0.5\zeta_{TOT}$$

2. Split it *proportionally* among the tasks: when the two jobs require portions of CPU that are very disproportionate, splitting the exceeding demands equally is not the optimal solution, because the performance of the one with the lower ζ will be degraded very heavily, while the other will suffer almost negligible consequences. In this case it will be better to degrade the tasks proportionally in the following way:

$$\begin{cases} \zeta_1^{new} = \frac{\zeta_1 \, \zeta_{TOT}}{\zeta_{req}} \\ \zeta_2^{new} = \frac{\zeta_2 \, \zeta_{TOT}}{\zeta_{req}} = \zeta_{TOT} - \zeta_1^{new} \end{cases}$$

   To write it in matrix form as in the previous case, it is first required to compute the two quantities a and b, which specify how to allocate ζ_{ex} :

$$a = \frac{\zeta_1}{\zeta_{req}} \qquad , \qquad b = \frac{\zeta_2}{\zeta_{req}} = 1 - a$$

   So it is possible to write:

$$\begin{cases} \zeta_1^{new} = \zeta_1 - a \, \zeta_{ex} \\ \zeta_2^{new} = \zeta_2 - b \, \zeta_{ex} = \zeta_2 - (1-a) \, \zeta_{ex} \end{cases}$$

$$\begin{bmatrix} \zeta_1^{new} \\ \zeta_2^{new} \end{bmatrix} = \begin{bmatrix} 1-a & -a \\ -b & 1-b \end{bmatrix} \begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix} + \begin{bmatrix} a \\ b \end{bmatrix} \zeta_{TOT} = \begin{bmatrix} 1-a & -a \\ a-1 & a \end{bmatrix} \begin{bmatrix} \zeta_1 \\ \zeta_2 \end{bmatrix} + \begin{bmatrix} a \\ 1-a \end{bmatrix} \zeta_{TOT}$$

Once the new values for the CPU shares are obtained, using the method that better suits the situation, it is required to go back and change the previous values on the sampling periods. This can be easily done by inverting the functions $f_{1_2}(\cdot)$ and $f_{2_2}(\cdot)$ thus obtaining:

$$T_1^{new} = f_{1_2}^{-1}(\zeta_1^{new}) \, , \qquad T_2^{new} = f_{2_2}^{-1}(\zeta_2^{new})$$

A better and more detailed scheme of this procedure is the one of Figure 42.

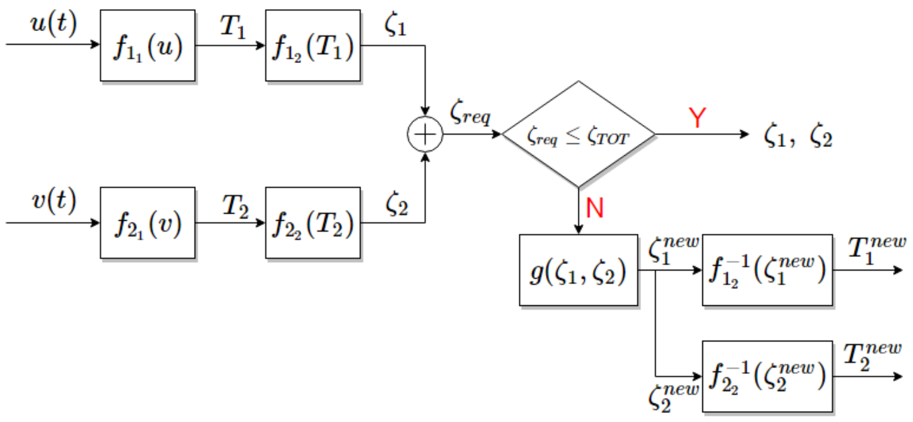


Figure 42.: Block diagram of a task scheduler with two tasks with no priorities.

Available computational resources varying with time

We always assumed the parameter ζ_{TOT} to be constant in time, and we kept it fixed once its value was decided. There are some cases in which the available CPU for the execution of the tasks changes over time. This can happen, for example, for the presence of noise and other disturbances, or for some malfunctions in the processing unit. This will lead to to a real value of available resources $\zeta_{real}(t)$ smaller than the a-priori known value ζ_{TOT}. In this case a way to measure that parameter is needed during every step of the algorithm, to make the task scheduler even more robust to dangerous situations.

$$\begin{cases} \text{Ideal scenario} & \rightarrow \quad \zeta_{real}(t) = \zeta_{TOT} = constant, \\ \text{Realistic scenario} & \rightarrow \quad \zeta_{real}(t) \leqslant \zeta_{TOT}. \end{cases}$$

In this way, every time the algorithm will be executed, a new and correct value of ζ_{TOT} will be used, guaranteeing the correct operation of the overall system.

5.2 APPLICATION TO REAL-CASE SCENARIO

The real-case scenario of this project involves the execution of two tasks: the *plane control* and the *image processing*. These tasks need to be executed at the same time and to share the same processing unit, thus we can apply the scheduling strategies presented in Section 5.1. The first task is clearly control-related, and will have as output its sampling period $T_P(t)$, while the latter involves data processing, specifically the execution of the *learning function* $l(\cdot)$ introduced in Section 4.2.1 on the images sequentially captured by the camera mounted on the air vehicle. The output will be $\tau(t)$, which is the time that has to pass between one picture and the next one captured by the camera. This variable is closely related to the execution time of such algorithm, but they are not equivalent: each time-slot τ indeed is the sum of the execution time of both tasks. This means that during the period of time $\tau(t)$ the plane dynamics will be controlled at sampling rate $T_P(t)$, and that the remaining computational resources can be used to increase the portion of "known" environment, thanks to the update of the learning function. A schematic version of how this works, time-wise, is presented in Figure 43, where it is specified that the first ϵ seconds in every time slot $\tau(t)$ are used for the actual processing of the scheduling algorithm; we can assume $\epsilon << \tau$ in general, so from now on we will neglect it.
This approach can be considered as both an *event-triggered* and a *self-triggered* control, as described in Section 2.3. The choice of the sampling period for the plane control $T_P(t)$ is made in a self-triggered fashion, because its value is computed based on the knowledge of the system and its state, which is in this case described by the complexity of the image at time t, $c(I(t))$. The length of the time period $\tau(t)$ also depends on the value of that state descriptor, the image complexity. Both of them

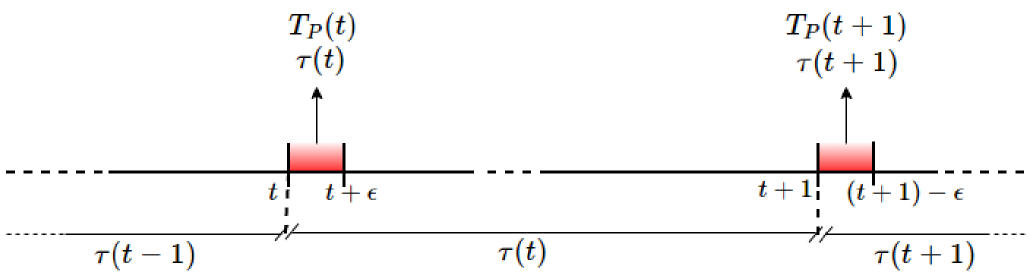


Figure 43.: Every τ seconds the scheduling algorithm computes the new values of $T_P(t)$ and the next $\tau(t)$.

are however updated when a certain event occurs, which is the expiration of time $\tau(t)$, resembling an *event-based* control.
The input of the overall process is the complexity $c(I(t))$ of the current image $I(t)$, introduced in Section 4.1: its value will determine the speed of the air vehicle $v(t)$, thus its sampling time $T_P(t)$, and the time between two consecutive pictures $\tau(t)$.

It is clear that we are working with variable sampling time: the instants of time $t-1$, t, $t+1$ and so on, are not equidistant: indeed, we have that:

$$t = (t-1) + \tau(t-1) \qquad \rightarrow \qquad t+1 = t + \tau(t)$$

So the general rule to compute the time instant t_i that we are analysing at a certain moment is:

$$t_i = t_{i-1} + \tau(t_{i-1}) = t_0 + \sum_{k=0}^{i-1} \tau(t_k) \qquad i = 1, \dots$$

5.2.1 Plane control with priority over image processing

First, the method already presented in Section 5.1.1, where the tasks can be organized by their priority, is developed. In this case, we assume that the task with higher priority is the plane control one, thus the appropriate choice for its sampling time is considered more important than the correct analysis of the current terrain picture. The overall process is presented in Figure 44.

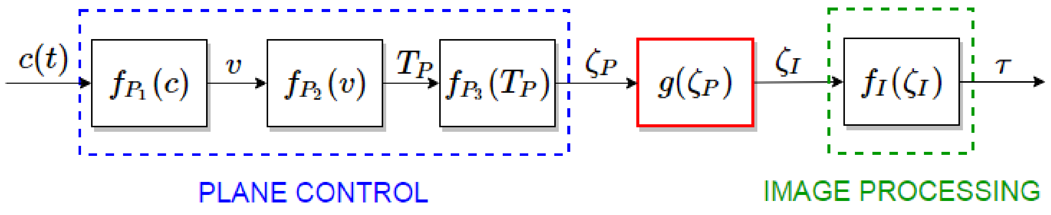


Figure 44.: Evolution of the algorithm when plane control has priority over image processing.

The first part of the algorithm involves the plane control, and its characterizing function $f_P(\cdot)$ can be summarized as the composition of the three internal function; the image processing task is in the same way characterized by the function $f_I(\cdot)$. The main difference from the previous theoretical development is that this second task is not control-related, hence its output will not be its sampling period, but the time period τ. All the functions involved in the algorithm are then the ones in the following Equation 13:

$$\begin{aligned} \zeta_P(t) &= (f_{P_3} \circ f_{P_2} \circ f_{P_1})\,(c(t)) \\ &= f_P(c(t)) \\ \zeta_I(t) &= g(\zeta_P(t)) \\ \tau(t) &= f_I(\zeta_I(t)) \end{aligned} \tag{13}$$

The input of the overall system, as mentioned before, is the complexity $c(t)$ of the current image $I(t)$: this can assume values from 0 to 1. When the complexity is low, the image is classified as not interesting, hence the plane can fly at high speed; on the contrary, when the complexity is high the image and the terrain the air vehicle is flying over are considered important, so the speed will be lower[2]. This relation is exploited through the function $f_{P_1}(c(t))$, that will be *monotonic* and *decreasing*: simulations have been made considering it a *linear*, *quadratic* or *cubic* dependency on $c(t)$; the actual form of this function will be apt to every mission's requirement. An example of how this map can be written is presented in Equation 14, with linear dependency:

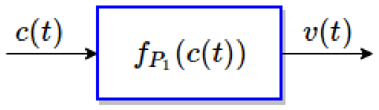
$$\begin{aligned} v(t) = f_{P_1}(c(t)) = v_{min} + \Delta v(1 - c(t)), \quad & \Delta v = v_{max} - v_{min}, \\ & 0 \leqslant c(t) \leqslant 1 \end{aligned} \tag{14}$$

As it is possible to see from Figure 45, we assumed the velocity of the air vehicle constrained between $15 \leqslant v(t) \leqslant 30$ [m/s]: the speed is maximum when the complexity is minimum, on the contrary with maximum complexity we have the slowest velocity.

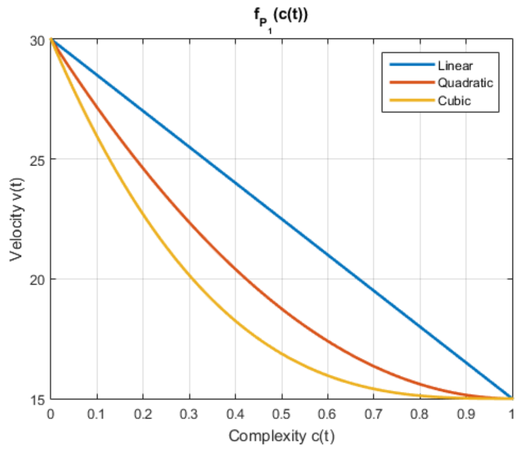
To connect the velocity of the vehicle to its control sampling periods $T_P(t)$[3] the function $f_{P_2}(v(t))$ is used. This map can assume numerous forms, but some basic hypothesis can be made: when the air vehicle flies at a medium speed, which means it is assuming its ideal cruise mode, it does not need a high control effort, because it is already in stable condition; in this case the sampling period can be maximized. On the other hand, more control is needed in situations in which stability is more difficult to maintain, i.e. in both the extremities of the allowed velocity range. In these two points the sampling period needs to be as small as possible, to provide

2 As explained in Section 3.1.1 the plane velocity is constrained between two bounds because of its dynamics, so we will have $v_{min} \leqslant v \leqslant v_{max}$.

3 A complete notation for this variable would be $T_P(v(t)) = T_P(v(c(t)))$. To avoid this complex and hardly legible notation, we maintained only the dependency on time.



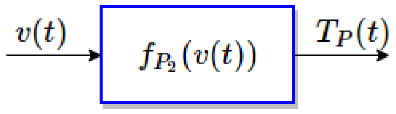
(a) Function block.



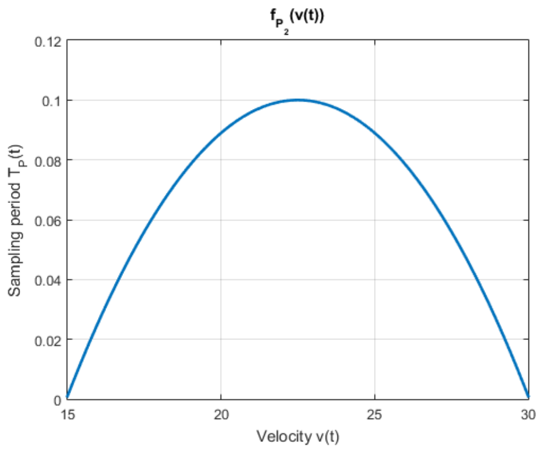
(b) Simulation results.

Figure 45.: Examples on how to obtain the UAV's velocity from the image complexity.

robustness. The conclusion is that the function $f_{P_2}(v(t))$ will have two minima at the extremes of its domain, and a maximum in the middle of its domain, as represented in Figure 46.



(a) Function block.



(b) Simulation results.

Figure 46.: Example on how to obtain the plane's sampling period from its complexity.

Here we assumed that the plane control sampling period could assume values in the range $0.001 \leqslant T_P(t) \leqslant 0.1$ [s]. The equation used in Figure 46 is the one of a *parabola* with vertex in $(v_{middle}, T_{P,max}) = (22.5[m/s], 0.1[s])$, passing through the points $(v_{min}, T_{P,max}) = (15[m/s], 0.001[s])$ and $(v_{max}, T_{P,max}) = (30[m/s], 0.001[s])$, that will also be the extremities of its domain. In general, the equation of a parabola is $y = ax^2 + bx + c$ and, by substituting the values listed before, one can obtain the three parameters a, b and c. In this specific case we obtained:

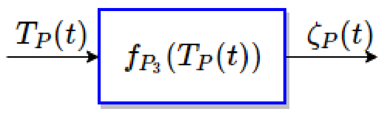
$$T_P(t) = f_{P_2}(v(t)) = -0.0018\, v^2(t) + 0.0792\, v(t) - 0.7910$$

It is clear that using this specific function will use the lowest values of T_P only if the speed is very high or very slow: another possible solution consists in using a *Gaussian* function, with its characteristic *bell* shape, that will allow to use the smaller values of T_P more often. It is necessary to remark that nevertheless an optimal solution does not exist, because everything depends on the needs of each mission.

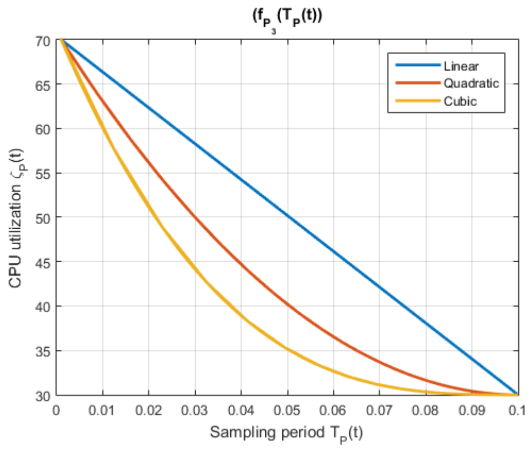
The third function characterizing the plane control task is $f_{P_3}(T_P(t))$, which establishes the relationship between the sampling period and the computation resources $\zeta_P(t)$ that need to be reserved for it. In this case too the function will be *monotonic* and *decreasing*: its maximum value is reached when the sampling period is the smallest, while the minimum will correspond to a larger T_P. An example for this map is the following, showing linear dependency:

$$\zeta_P(t) = f_{P_3}(T_P(t)) = \zeta_{P,min} + \Delta\zeta_P(1 - T_P^{'}(t)), \qquad \Delta\zeta_P = \zeta_{P,max} - \zeta_{P,min}$$

To use the previous equation the data T_P needs to be normalized in the range $0 \div 1$, producing $T_P^{'}$. We assume to know also the values of the minimum and maximum CPU requested, $\zeta_{P,min}$ and $\zeta_{P,max}$ respectively; their values have here been fixed to 30 and 70 [%] [4]. A graphical representation of how the function can be chosen is presented in Figure 47.



(a) Function block.



(b) Simulation results.

Figure 47.: Example on how to obtain the necessary computational resources for the plane control from its sampling period.

It is possible to see in Figure 48 the complete function $f_P(c(t))$ for the plane control task and observe how the final variable ζ_P changes as the input $c(t)$ grows.
If a linear function is used in f_{P_1} and f_{P_3} the minimum usage of CPU for this first task will be found when the image has medium complexity, because the vehicle would be travelling at its ideal cruise velocity, thus it will require minimum control

4 Clearly the value of $\zeta_{P,max} \leqslant \zeta_{TOT}$, otherwise the overall scheduling problem will have no meaning.

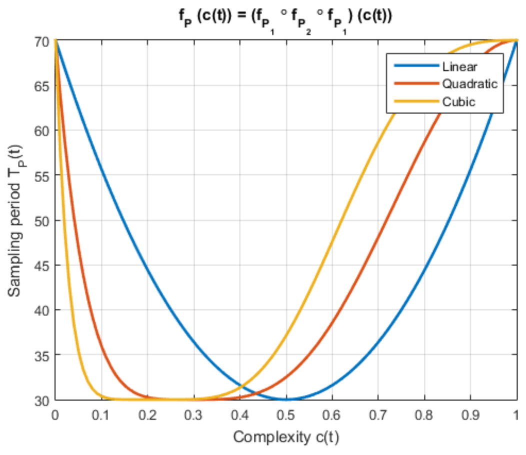


Figure 48.: Complete function $f_P(c(t))$ for the plane control.

effort; for the opposite reason, the CPU usage will be found at the extremes of the complexity range. When a cubic or a quadratic function is used we can see that the minimum utilization is reached on a different position, nearer to small values of the complexity: this happens because these functions are *convex*. On the contrary, if *concave* functions are used, the minimum will be near higher values of complexity.

Once the value of ζ_P is known, the task scheduling function $g(\zeta_P)$ can be used to determine the computational resources that the image processing task can utilize. In this case also the map will be *monotonic* and *strictly decreasing*. We can decide to use all the available CPU to complete this second task, in which case the scheduling function will simply be:

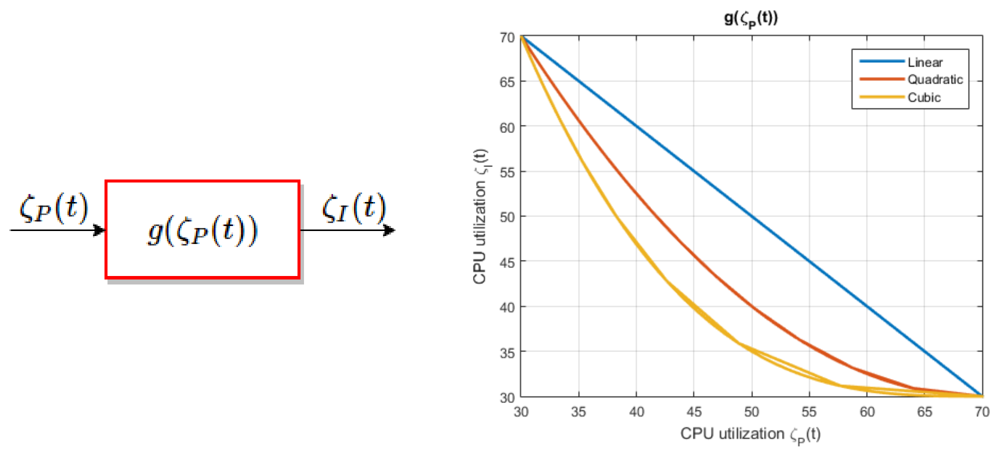
$$\zeta_I(t) = g(\zeta_P(t)) = \zeta_{TOT} - \zeta_P \tag{15}$$

or it can assume other forms, in which case the condition $\zeta_P + \zeta_I \leqslant \zeta_{TOT}$ always needs to be checked. Another simple function that can be used exploit a linear dependency between the two variables:

$$\zeta_I(t) = g(\zeta_P(t)) = \zeta_{I,min} + \Delta\zeta_I(1 - \zeta_P'(t)), \qquad \Delta\zeta_I = \zeta_{I,max} - \zeta_{I,min}$$

where ζ_P' is the value of ζ_P normalized between $0 \div 1$. For all the following simulation it has been assumed $\zeta_{I,min} = 30$ and $\zeta_{I,max} = 70$, so that $\zeta_P(t) + \zeta_I(t) = \zeta_{TOT}$ in all time instants, thus reflecting the situations of Equation 15. In Figure 49 we can see examples of how the task scheduling function works. If observed carefully, it is possible to observe a double line regarding the quadratic and the cubic case: this actually exist also for the linear case, but it not visible because the two lines are perfectly overlying. This is due to the fact that the function $f_P(c(t))$ is not *bijective*, because not *surjective*.

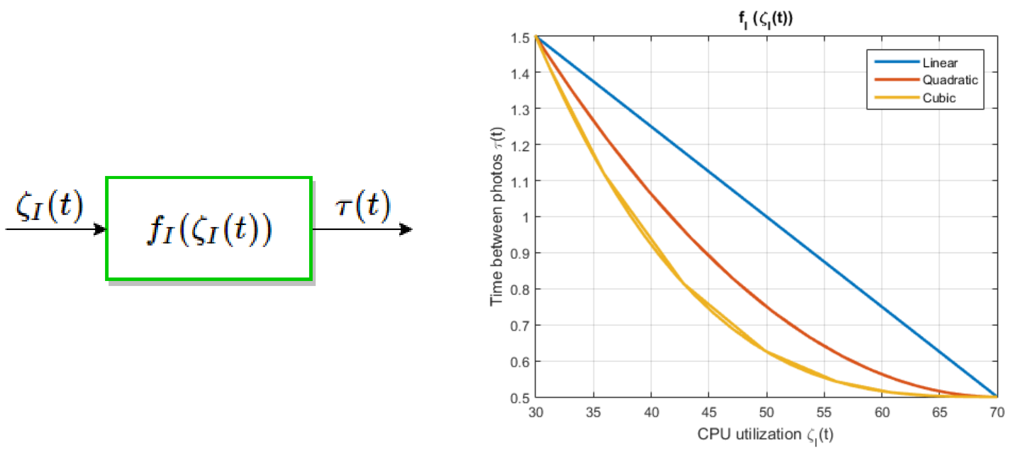
The final step considers the achievement of the value of $\tau(t)$ through the function $f_I(\zeta_I(t))$, that correlates the time that has to pass between two consecutive shots



(a) Function block. **(b)** Simulation results.

Figure 49.: Example on how to obtain the computational resources for the image processing task from the CPU portion required for the plane control.

of the camera (and thus influencing the update of the learning function $l(\cdot)$) to the remaining resources. It need to be *monotonic increasing*, as one can observe from Figure 50, because the less CPU it has available, the smaller number of operations can be performed. In this case to update in a significant way the map of the known environment, it is necessary to have a longer τ period. We imagined the value of $\tau(t)$ to be included in the range $[0.5,\ 1.5]$ seconds.



(a) Function block. **(b)** Simulation results.

Figure 50.: Example on how to obtain the time period $\tau(t)$ from the remaining available CPU.

If we plot the value of $\tau(t)$ with respect to the complexity as in Figure 51, although, we can see how this result is not satisfying: when the image $I(t)$ presents a low complexity, the time $\tau(t)$ assumes its maximum value, as one can expect (since the current terrain is not interesting, there is not need to capture a great number of photos). The same situation is however present also when the complexity is high, in which case we would like τ to be small. Indeed, the relation between the complexity and τ should be *strictly decreasing*, and this is why this approach where the plane control task is prioritized over the image processing one is not suitable for this scenario.

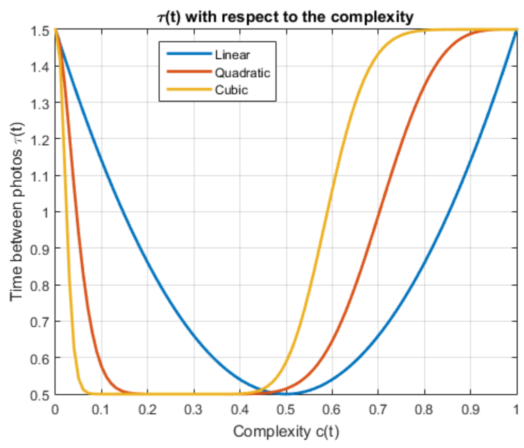


Figure 51.: Simulation results for the prioritized case.

We observe eventually the total consumption of CPU resources from Figure 52: using only linear function guarantees a complete use of the computation resources, while quadratic or cubic functions will maintain part of the CPU idle, except for some critic points. This can be useful in case some other task of secondary importance needs to be performed, or if battery saving is important for the specific mission. Of course the results of using 100% of the CPU in the linear case is obtained only because of the choices for ζ_P and ζ_I range of values, that always sum up to 100. If those values are changed, also the slope of the line representing the total resources consumption will vary.

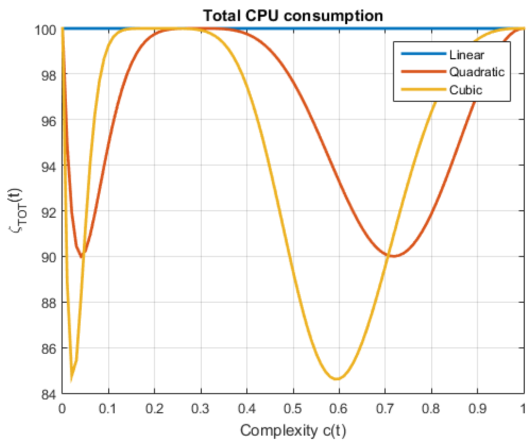


Figure 52.: Simulation for the total CPU used in the prioritized case.

5.2.2 Plane control and Image Processing with equal importance

When this approach is used, the two task are not prioritized, and their correct accomplishment has the same importance: the partition of the CPU's resources is conducted simultaneously and independently. If, at the end of this process, the total resources needed ζ_{req} result higher than the actual available resources ζ_{TOT}, the algorithm will proceed with their re-distribution using one of the methods presented in Section 5.1.2. Following the same procedure, the block diagram that summarizes the process is the one of Figure 53.

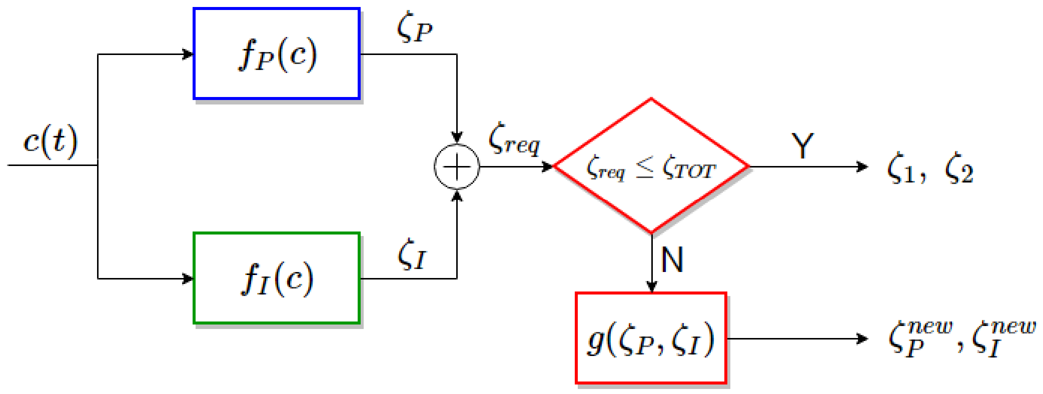


Figure 53.: Block scheme of the complete process when the tasks do not have priorities.

The algorithm computes the two values $\zeta_P(t)$ and $\zeta_I(t)$ through the functions $f_P(c(t))$ and $f_I(c(t))$: they both use the same input $c(t)$, the complexity of the image $I(t)$. Specifically:

- The function $f_P(\cdot)$ that regulates the computational resources needed to perform an appropriate air vehicle control has the same characteristics of the one used in Section 5.2.1:

$$\begin{aligned}\zeta_P(t) &= (f_{P_3} \circ f_{P_2} \circ f_{P_1})\,(c(t))\\ &= f_P(c(t))\end{aligned}$$

  with the same properties of the inner functions already disclosed in the same Section. The block scheme follows in Figure 54.

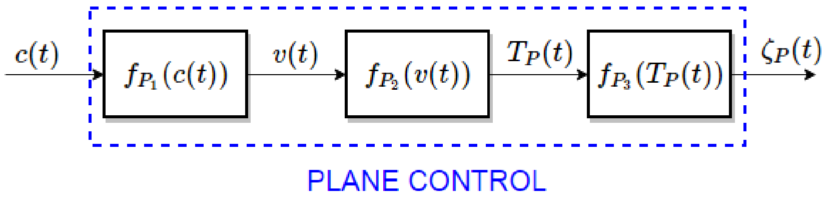


Figure 54.: Block scheme of the function $f_P(\cdot)$ for the plane control.

In this case the overall behaviour of function $f_P(\cdot)$, with respect to the complexity $c(t)$, can be observed in Figure 48 on page 69.

- The image processing function $f_I(\cdot)$ can be decomposed as in the block scheme of Figure 55. The first inner function $f_{I_1}(c(t))$ uses the image complexity to

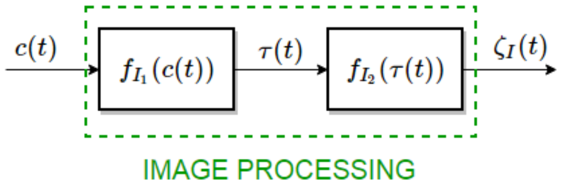
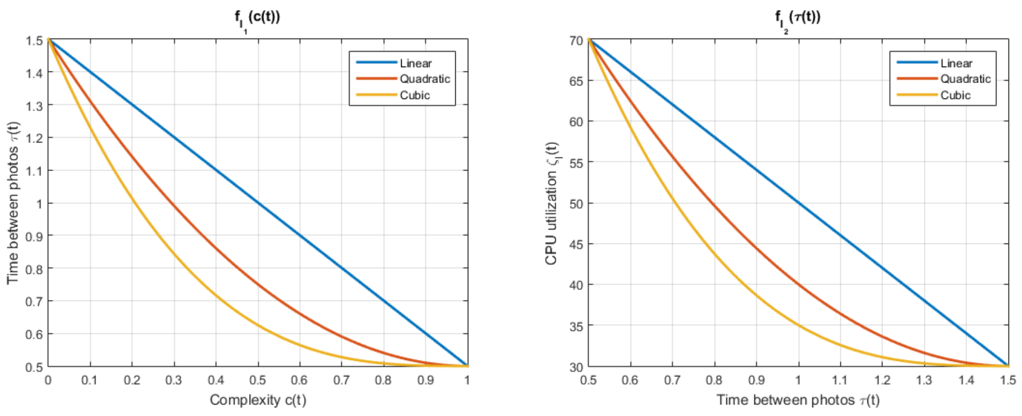


Figure 55.: Block scheme of the function $f_I(\cdot)$ for the image processing.

determine the time $\tau(t)$ that has to pass between two consecutive photos captured by the vehicle's camera. This map needs to be *monotonic* and *strictly decreasing*: when an image is classified as non-interesting, i.e. $c(t)$ is low, more time can pass until the next image is processed; on the contrary, if an image is considered important, i.e. its $c(t)$ is high, the value of $\tau(t)$ needs to be small, so that there are more possibilities to study and process the area over where the vehicle is flying. This function can, for example, have the behaviour presented in Figure 56a.



(a) Simulation results for $f_{I_1}(\cdot)$. **(b)** Simulation results for $f_{I_2}(\cdot)$.

Figure 56.: Examples on how the image processing functions can be implemented.

This may seem as a contradiction: we are reserving a longer time for the algorithm when it is not really needed, because when $c(t)$ is low the learning function $l(\cdot)$, that is being used to generalize an image processing application, increases its values much faster, thus making the area easily "known". It is necessary to remember although that such function is also influenced by the available resources $\zeta_I(t)$ in a *decreasing* manner, i.e. the fewer resources are reserved in such period $\tau(t)$, the harder it is to learn about the area, because the operations are processed slower. Exploiting the situation, is possible to come up with some requirements for function $f_{I_2}(\cdot)$, which controls how $\tau(t)$ is connected to $\zeta_I(t)$. A longer $\tau(t)$ will allow more time to conduct the required operations of the image processing task, thus demanding less computational resources; a small $\tau(t)$ requires the same image processing algorithm to be exploited in a smaller amount of time, i.e. it will be more demanding. Observing Figure 56b one can see examples of how the function can be implemented. The overall function $f_P(c(t))$ has therefore the following form:

$$\begin{aligned}\zeta_I(t) &= (f_{I_2} \circ f_{I_1})\,(c(t)) \\ &= f_I(c(t))\end{aligned}$$

The CPU resources requested by the two task can be summarized in Figure 57.

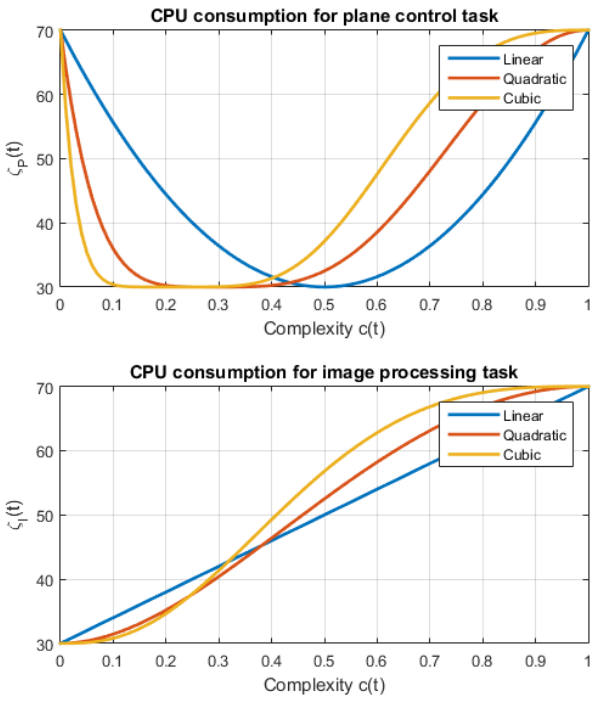


Figure 57.: Computational resources used by the plane control task and by the image processing task in relation to the complexity of the image, supposing both of them constrained in the range [30, 70] %.

The plane control task requires the maximum use of CPU when the complexity of the image is either very low or very high, because those are the most difficult situations to control: the air vehicle has to fly using a velocity which is at the two extremes of its allowed range. On the contrary the image processing task uses all of its reserved resources only when the image complexity is $\simeq 1$. It is now clear that this will be the most critical condition, since both tasks will use all of the resources held for them.

Once the two values of ζ_P and ζ_I are known, we can compute the total CPU required by computing their sum: if $\zeta_{req} < \zeta_{TOT}$, the algorithm can be executed with the current values of T_P and τ; otherwise, the resources demanded by the two tasks are too high, hence they need to be adjusted using the task scheduler function $g(\zeta_P, \zeta_I)$.
From Figure 58 it is possible to see that the situation is ideal, i.e. no CPU overload

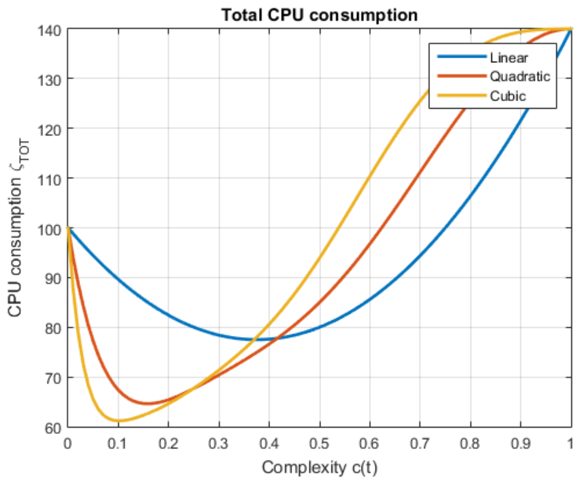
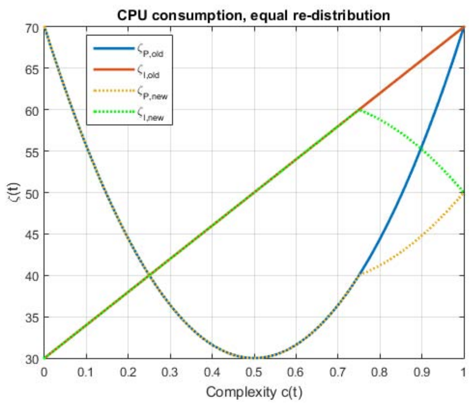
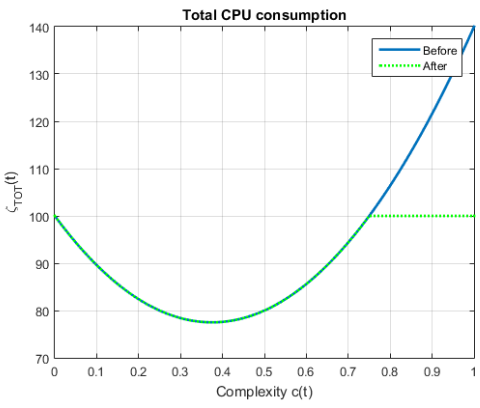


Figure 58.: Total CPU requested by the two tasks simultaneously.

occurs, when the value of complexity is lower than 0.75, assuming the use of linear functions and $\zeta_{TOT} = 100\%$. Over this value, it is necessary to adjust ζ_P and ζ_I using some techniques such as the ones of Section 5.1.2.
When the first procedure is used, i.e. splitting the excess of demand *evenly* between the tasks, the result is the one presented in Figure 59: in particular Figure 59a shows how the CPU is adjusted for each individual task, while Figure 59b shows that, after applying the task scheduling function $g(\zeta_P(t), \zeta_I(t))$, the condition $\zeta_{req} \leqslant \zeta_{TOT}$ holds, which was the result we expected.



(a) Effect of the scheduling function for each task.



(b) After the re-distribution, $\zeta_{req} \leqslant \zeta_{TOT}$ holds.

Figure 59.: Simulation of of the scheduling algorithm with even re-distribution between tasks.

The second method considered a *proportional* division of the exceeding CPU, that degrades the performances of each task accordingly to their effective use of CPU, so that one task is not more disadvantaged than others from this re-allocation of resources. The results that are obtained resemble the ones in Figure 59, obtained using the previous method.
To observe a more evident difference between the two approaches it is necessary to be in a more complex situation, e.g. modifying the ranges in which ζ_I and ζ_P can take values. For the results in Figure 60 for example the values $40 \leqslant \zeta_P(t) \leqslant 100$ [%] and $30 \leqslant \zeta_I(t) \leqslant 80$ [%] have been used.

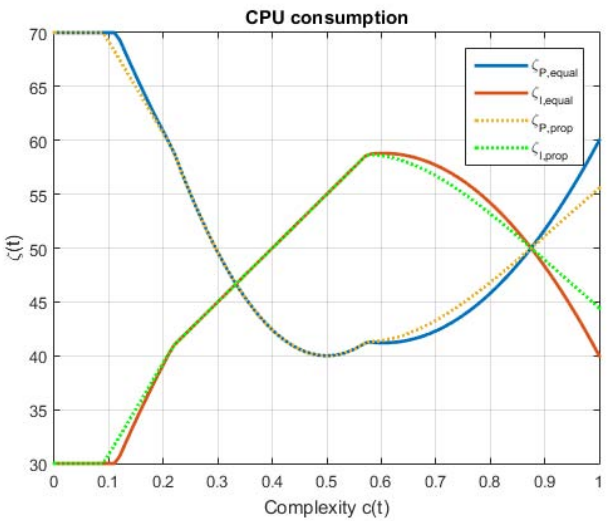


Figure 60.: Re-distribution of resources using the two methods of even and proportional partitioning of the excess.

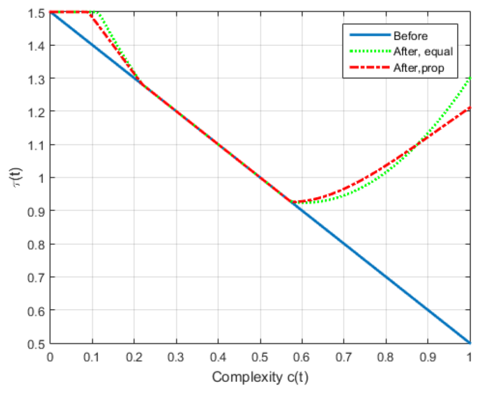
The new values ζ_P^{new} and ζ_I^{new} clearly have consequences on the other data involved: $\tau(t)$ and $T_P(t)$ (and also $\nu(t)$[5]) are modified, and will assume a new value, obtained inverting the functions $f_{P_3}(\cdot)$ and $f_{I_2}(\cdot)$. Using the same data of the previous example, it was possible to obtain their new value, as reported in Figure 61.
It is clear that $\tau_{new}(t)$ and $T_{P,new}(t)$ are very far from their ideal values (although this example has been designed with the goal of emphasizing the difference between the old and new values, by taking the algorithm to its extremes. Indeed, this is not

5 Regarding the inversion of $f_{P_2}(\cdot)$ to obtain the velocity of the vehicle: the function is not bijective, so formally its inversion in impossible. There is however a way to solve the problem, exploiting the fact that the value of the input $c(t)$ do not change in the time interval τ when the task scheduling algorithm is executed. From Figure 46 on page 67 it is possible to see that with each value of T_P we can choose between two different velocities, ν_1 (on the left, the lowest) and ν_2 (on the right, the highest). The choice between the two is simple:

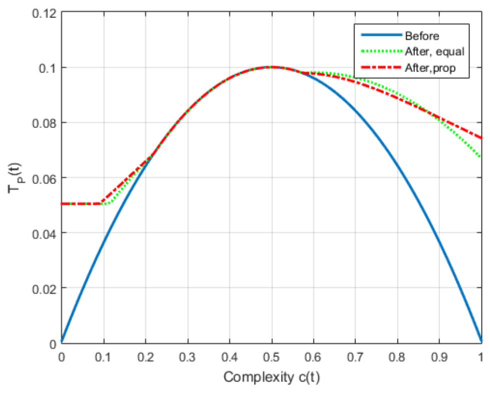
$$\begin{cases} \text{If } c(t) \leqslant 0.5 & \rightarrow \quad \nu(t) = \nu_2 = max(\nu_1, \nu_2), \\ \text{If } c(t) > 0.5 & \rightarrow \quad \nu(t) = \nu_1 = min(\nu_1, \nu_2). \end{cases}$$

This means that it is possible to write the inverse function not as $f_{P_2}^{-1}(T_P)$, but more precisely as $f_{P_2}^{-1}(T_P, c)$.

an ideal situation), but still it allows the two tasks to keep on functioning without causing damages to the processing unit.



(a) New values of $T_P(t)$.



(b) New values of $\tau(t)$.

Figure 61.: New values assumed by T_P and τ after the re-distribution of CPU resources.

6 SIMULATION RESULTS ON THE REAL-CASE SCENARIO

6.1 THEORETICAL RESULTS ON MEANINGFUL EXAMPLES

We try now to make the scenario of Section 5.2.2 more realistic, using data that can be applied to some real plane control and image processing task.
First, we exploit some data that are known:

- The *update rate of the sensors* mounted on the air vehicle: the sensors gives new data with a fixed frequency, around the value of $f_s = 25\text{Hz}$ ($T_s = 40ms$). Since the inner loop dynamics are usually $5 \div 10$ times faster, this means that it is possible to obtain the maximum value for the plane sampling period, that will be $T_{P,max} = 8ms$.

- The *sampling frequency of the on-board CPU*, optimized for high-speed computations. It runs at a maximum rate $f_P = 1k\text{Hz}$, which means that the smaller period for the plane control could be $T_{P,min} = 1ms$ [43]. This value is although not acceptable, because it does not take into account that a small part of such controller always need to be shared with the second task, which leads to the actual minimum value $T_{P,min} = 1.1\ ms$. This means that the sensors can be *over-sampled*, giving the controller a better statistical measure: the additional control is therefore *redundant*, providing the necessary robustness for more precarious flight conditions. Indeed, this gives the vehicle a very high control bandwidth, making it very stable in flight. We assume also that the execution time of the control is $1\ ms$.

- The *maximum rate of the camera* mounted on the UAV, i.e. the smaller time that has to pass before the camera will be able to take another picture. Usually this value is $f_I = 5\text{Hz}$, or $T_I = 200ms$. It is easy to see that this parameter coincides with the variable τ, which means that $T_I = \tau_{min} = 0.2\text{s}$. The value of τ_{max} is here also fixed to $1s$.

- The *Worst Case Execution Time of the Path Planning algorithm*, that we will fix to $T_{pp} = 30ms$ and the *Worst Case Execution Time of the Task Scheduling algorithm*, fixed also to $T_{ts} = 30ms$.

We study how the task are scheduled in three meaningful examples. In the strategy that will be used, all functions uses a linear (proportional) rescaling, making it computationally efficient, thus possible to use on-line.

6.1.1 Medium Complexity Image

When $c(t) = 0.5$ it is possible to obtain, using function $f_{I_1}(c(t))$[1], the value of τ, in this case equal to 0.6 s. All this period $\tau(t)$ is interposed by the smaller time periods $T_s = 40$ ms, when the vehicle's sensors supply new measurement for the plane control[2].
The execution time for the path planning algorithm and the scheduling algorithm need to be reserved only once in each time segment τ, respectively at the beginning and at the end. This leaves, in each one of those time slots, 10 milliseconds available for the plane control: this means that for these two time slots $(0 \div T_s$ and $(\tau - T_s) \div \tau)$ the sampling period for the plane control will be fixed in any condition and equal to $T_P = 4$ms, since $T_P = T_s/t_{av}$, where t_{av} is the available time in each time slot. Figure 62 offers a graphical representation of the situation.

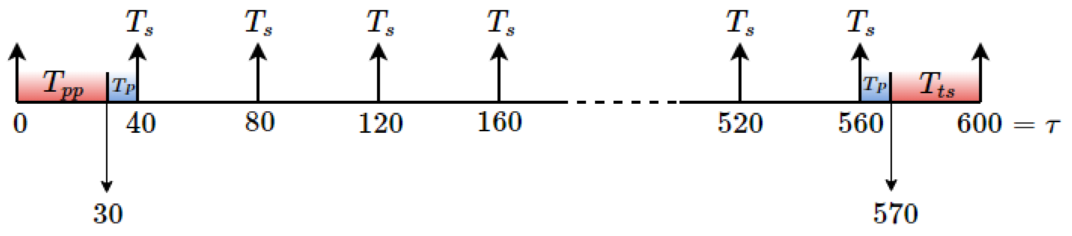


Figure 62.: Times partitioning for the $c(t) = 0.5$ situation.

Having $c(t) = 0.5$ means that the air vehicle is flying in an ideal cruising mode, where only a minimum amount of control is needed, leading to a value $T_P(t) = T_{P,min} = 8$ms. In this case, to avoid the great amount of spare CPU that will result by using the functions of Section 5.2, it will be more suitable to use a different kind of function $f'_P(\cdot)$, in which the value of $T_P(t)$ will depend not only on the complexity but also on the *tracking error* $e(t)$, i.e. on how far the vehicle current position is from the desired trajectory. On the other hand, even if the tracking error would turn out the highest possible, being the UAV very far, it will not be possible to use $T_P(t) = T_{P,min} = 1.1$ ms, because the image processing task also requires part of each time slot T_s. Using the function $f_I(\cdot)$ as defined in Section 5.2.2, it is possible to recover that data: assuming $10 \leqslant \zeta_I(t) \leqslant 60$ [%], for $c(t) = 0.5$ we obtain $\zeta_I(t) = 35$ %, i.e. the time required for image processing task t_{ip} will be equal to 14 ms during every time slot T_s [3], and 182 ms in the entire slot $\tau(t)$. This leaves a

1 The function is assumed to have a linear form, i.e.

$$\tau(t) = \tau_{min} + \Delta\tau(1 - c(t)), \qquad \Delta\tau = \tau_{max} - \tau_{min}$$

2 Of course if the value of T_P will result higher than 8 ms, i.e. $1/T_s$, an access to the sensors value will be made more often.

3 Assuming that the whole time slot of 40 ms can be used for these two tasks, this value corresponds to $\zeta_{TOT} = 100$ %. Then $t_{ip}(t) = \zeta_I(t) \cdot 40/100$

total of 26 ms for the plane control, hence $T_{P,min} = 1.5$ ms. In the end, therefore, we will be left with a choice of $T_P(t)$ in the range $1.5 \leqslant T_P(t) \leqslant 8$ $[ms]$, depending on the tracking error $e(t)$. Let's imagine for the example that it will give as a result 3 ms: then in every time slot ~ 13 ms will be reserved for the plane control task, meaning $\zeta_P(t) = 33$ %, as in Figure 63. There will remain 32 % of the computational resources available for other purposes: this idle CPU can be used simply to save energy, to fulfil some other secondary task, e.g. communication with a base station, or to improve the performances of the tasks.

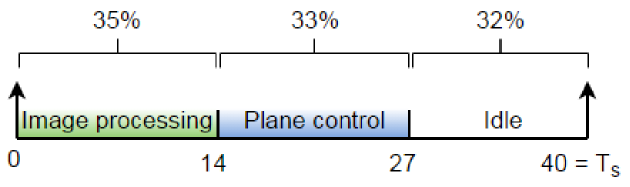


Figure 63.: Partitioning of a time-slot T_s.

An observation needs to be made about the actual distribution in each time slot T_s of t_p and t_i: the representation of Figure 63 (and later of 66 and 69) is just a simplification of how they are actually allocated. It is clearly not true that the image processing task is all executed first and the plane control task is executed all together only after that, or vice-versa. The plane control indeed needs to be executed periodically, with $T_P = 3$ ms in this case, and we supposed it to have an execution time of 1 ms; the image processing task on the contrary does not have that need, but it will be *preempted*. A more realistic representation is presented in Figure 64: the green blocks represent the image processing task; the blue blocks represents the plane control part, and each block is 1 ms long, which is the execution time of the control algorithm. The white part represents the idle time for the CPU.

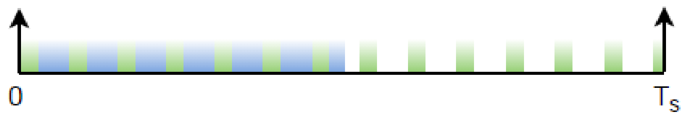


Figure 64.: Realistic situation of the plane control being executed periodically and the image processing being preempted.

6.1.2 Low Complexity Image

When the complexity is low, i.e. $c(t) \approx 0$, the corresponding value of τ will be the highest, so $\tau(t) = 1$ s. Such as in the previous case, the first and final time slots are reserved for the path planning and the task scheduling algorithm, while the remaining time $\tau(t) - 2T_s$ can be used for the tasks we are analysing.

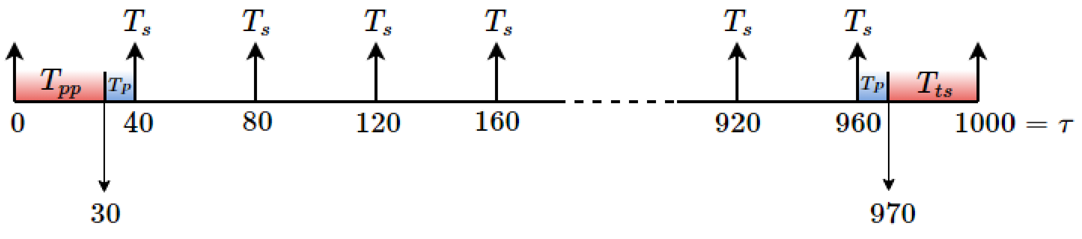


Figure 65.: Times partitioning for the $c(t) \approx 0$ situation.

In this situation the speed of the air vehicle will be the greatest possible, so the minimum sampling period for the plane control will be very small, since the plane is working on an area where the control is harder: function $f_P(c(t) = 0)$ will give as a result $T_P(t) = T_{P,min} = 1.1$ ms. This value stems from the fact that it is not possible to use the real minimum value of 1 ms, because a portion of each time segment T_s always needs to be reserved for the image processing task, as mentioned at the beginning of Section 6.1. The smallest share for that task occurs when it needs only $\zeta_{I,min} = 10\%$, i.e. exactly this case in which $c(t) = 0$ (indeed using $f_I(c(t) = 0)$ the obtained result is $\zeta_I(t) = 10\%$). That result corresponds to $t_{ip} = 4$ ms, leading to an available time for the plane control equal to $t_{pc} = 36$ ms $\Rightarrow T_{P,min} = 1.1$ ms. In this case the necessary computational resources for the plane control task are $\zeta_P(t) = \zeta_{P,max} = 90\%$, as in Figure 66.

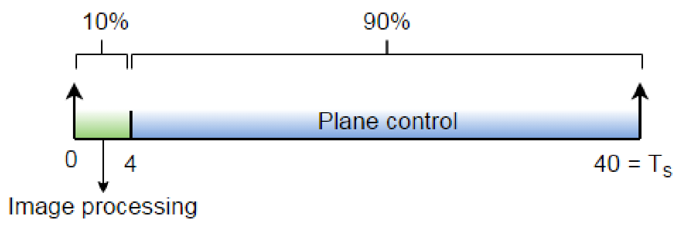


Figure 66.: Partitioning of a time-slot T_s.

In this example, to retrieve the value of $T_P(t)$ the function $f_P(\cdot)$ of Section 5.2 has been used. The application of $f'_P(\cdot)$, function that also take into account the tracking error, will have given the same result, because in this case there are no free resources that can be used to raise the sampling rate.

6.1.3 High Complexity Image

When the image has high complexity $c(t) \approx 1$, it means that the area the plane is flying over is very interesting, so, according to function $f_{I_1}(c(t))$ the time between two consecutive images will be small, i.e. $\tau(t) = 0.2$ s, as depicted in Figure 67. According to function $f_{I_2}(\tau(t))$, the computational resources that needs to be reserved for the image processing task are $\zeta_I(T) = 60$ % of the CPU, with $t_{ip} = 24$ ms,

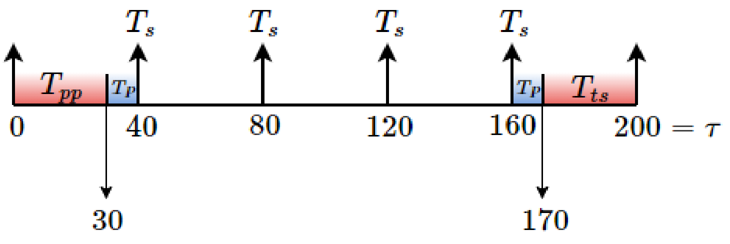


Figure 67.: Times partitioning for the $c(t) \approx 1$ situation.

with only 16 ms remaining for the plane control task in each time slot T_s. This value of t_{pc} leads to a minimum value $T_{P,min} = 2.5$ ms, that is nevertheless not enough for the required control: when the complexity is high the UAV flies at a minimum speed, thus requiring a sampling period $T_P(t) = T_{P,min} = 1.1$ms and a CPU portion of $\zeta_P(t) = 90\%$. Unlike the previous examples, in this case the condition $\zeta_{req}(t) \leqslant \zeta_{TOT}$ is violated: a re-distribution of available resources is necessary. Applying the scheduling function $g(\cdot)$ in the *proportional* fashion we will obtain the new values $\zeta_P^{new}(t) = 60\%$ and $\zeta_I^{new}(t) = 40\%$. Proceeding with the scheme of Figure 41 and inverting functions $f_{P_3}(T_P(t))$ and $f_{I_2}(\tau(t))$, it is possible to obtain, respectively, the new values $T_P^{new}(t) = 3.8$ ms and $\tau^{new}(t) = 0.52$ ms [4]. The new situation will be the one in Figure 68.

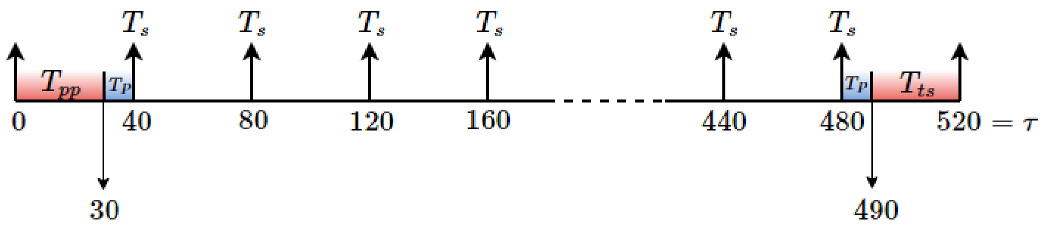


Figure 68.: Times partitioning for the $c(t) \approx 1$ situation, after being correctly scheduled.

The allotment of resources in a single time slot T_s is the one presented in Figure 69, and clearly does not allow for any idle time for the processing unit.

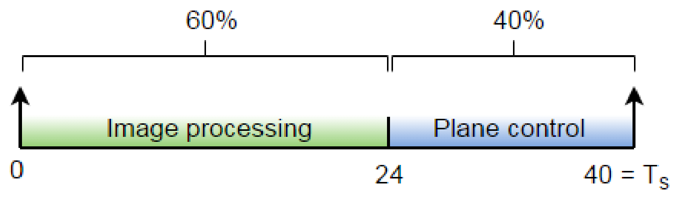


Figure 69.: Partitioning of a time-slot T_s after the re-distribution of resources.

In general, one can see how all of the resources are re-allocated in Figure 70: for

4 The results that we would have obtained with an *equal* re-distribution of resources are quite similar: $\zeta_P^{new}(t) = 65$ and $\zeta_I^{new}(t) = 35$, leading to $T_P^{new}(t) = 3.2$ ms and $\tau^{new}(t) = 0.6$ ms.

low values of complexity the behaviour of the complete system is ideal, in the sense that it never needs to resort to the scheduling function. The minimum of the CPU consumption is reached when $c(t) \approx 0.43$, and it is equal to $\approx 44.5\%$. The necessity for the re-distribution appears only when $c(t) \geqslant 0.84$.

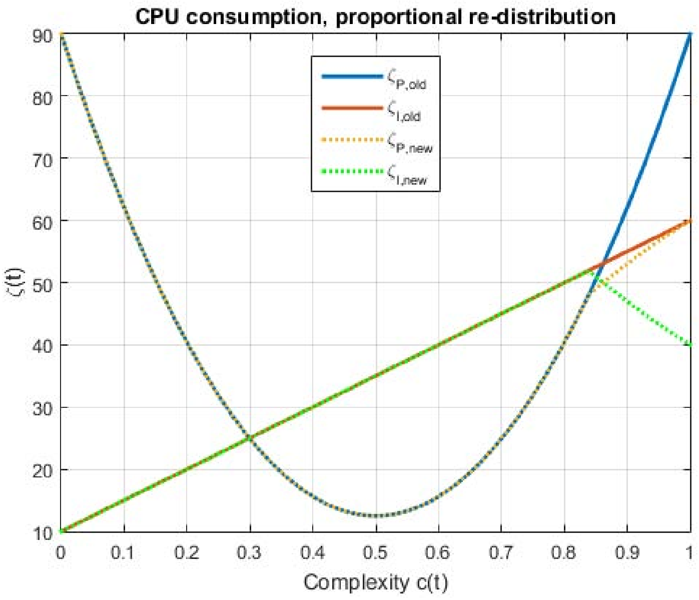


Figure 70.: The re-allocation of resources starts when $c(t) \geqslant 0.84$.

6.2 SCHEDULING ALGORITHM ON THE REALISTIC SCENARIO

We tested in conclusion the outcome of the overall algorithm, by integrating all the components of the simulated scenario described in the previous chapters. In Chapter 3 the first task has been developed, with the UAV control, path planning and path following algorithms. The second task, i.e. the learning function, has been explained in Chapter 4, following the development of the complexity function. In Chapter 5 ultimately the scheduling algorithm has been built, which sets all the variable parameters ($\nu(t)$, $T_P(t)$ and $\tau(t)$) based on the complexity of the current FOV and the available resources $\zeta_P(t)$ and $\zeta_I(t)$.
After the decision of an arbitrary starting point on the environment map and the initial heading of the air vehicle, the result is the one of Figure 71, over a time period of 50 s. The yellow plane represents the initial position, the light blue one final position, the blue line the trajectory that has been followed and the green circle the time instants $t_0 = 0$ s, $t_1 = t_0 + \tau(t_0)$ s, ..., in which the variables are updated.

Before analysing the behaviour of the UAV, it is necessary to point out that the trajectory that it has to follow is not ideal. We programmed the path planning

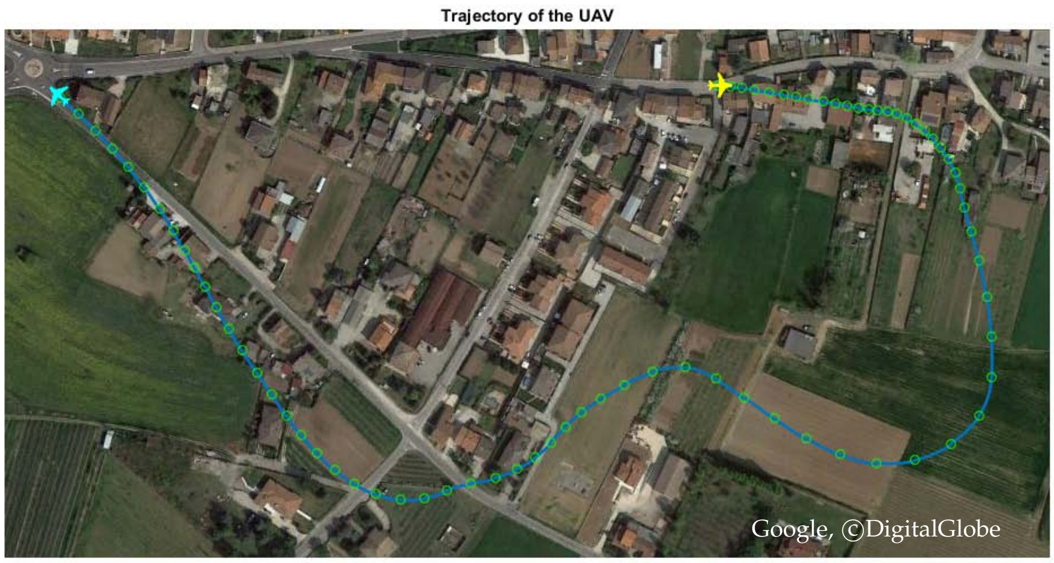


Figure 71.: Trajectory of the air vehicle.

algorithm so that it follows the longest line in each image captures at the time instants t_i. This line can vary in a significant way from one image to the next, thus making the trajectory not very smooth. To avoid, or to reduce, this problem, a small adjustment to the path planning algorithm has been made. The length of the line has been considered as its *weight*, and in each new image the line to follow is being updated only if its weight, θ_c, is bigger than the previous weight, θ_p (i.e. if the new line is longer than the last one). If this does not happen, the straight line to follow remains the one of the previous FOV. Furthermore, in order to avoid that, when a line has the maximum length, the UAV keeps on following the same one, its weight is repeatedly decreased of a certain value.

$$\begin{cases} \text{if} \quad \theta_c < (\theta_p - 10) & \rightarrow \qquad \theta_c = \theta_p - 10, \quad l_c = l_p \; ; \\ \text{if} \quad \theta_c \geqslant \theta_p & \rightarrow \qquad \theta_c, l_c \quad \text{unchanged.} \end{cases}$$

Once the adjustment has been made, the results will be the one of Figure 72, with a smoother trajectory.
It is possible to observe how, in the areas with a greater number of buildings, the time instants are much closer among them, as a result of a higher complexity. This will lead to low velocities, small sampling periods and small time periods τ. On the contrary, when the plane is flying over fields, e.g. in the right section of the image, the complexity is low and the time instants are more diluted. This correct behaviour is summarized in Figure 73, which shows how the parameters vary when the complexity change. Also associated to the complexity are the portions of CPU associated to each task, defined by the functions described in Section 5.2.2. We can see how during certain time periods the total usage of CPU reaches 100 %, but it never evolves into a re-allocation of resources. The mean values of the parameters

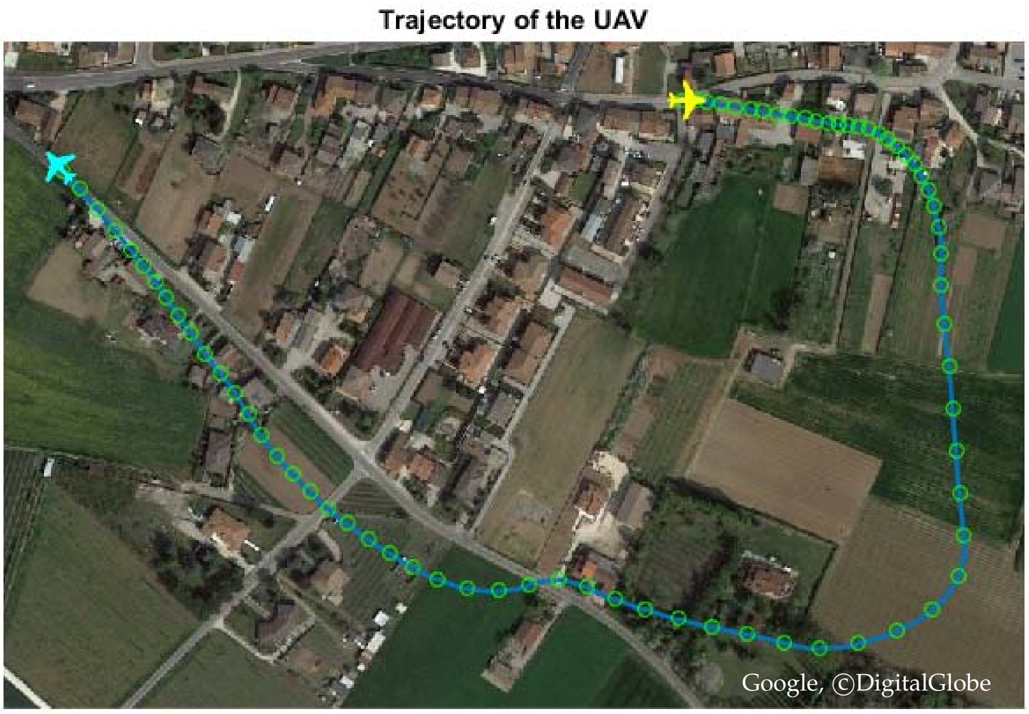


Figure 72.: Trajectory of the UAV with the optimized path planning algorithm and NLGL guidance law.

are summed up in Table 8: in particular the value of $\overline{\zeta_{TOT}}$ is positive, leaving a great amount of free resources for other tasks.

Table 8.: Mean values of the parameters for the NLGL guidance law.

$\bar{v}$ [m/s]	$\overline{T_P}$ [s]	$\bar{\tau}$ [s]	$\overline{\zeta_P}$ [%]	$\overline{\zeta_I}$ [%]	$\overline{\zeta_{TOT}}$ [%]
24.3	0.006	0.7	36.2	29.0	65.1

We would also like to see how the image processing task has behaved, even if it is not possible to give an absolute verdict, i.e. to decide whether it was "good" or not. We can nevertheless study how much of the complete map has been visited, and with which grade of accuracy. The UAV fled over 349037 pixels, i.e. it studied 1.1222% of the environment. The mean degree of knowledge[5] is 0.7839. Of this visited points, 182438 are completely known, i.e. their value in the environment matrix L is equal to 1 (0.5865% of the complete map and 52.269% of the total visited cells). In Figure 74 it is possible to observe what has just been described: the areas that belong to the FOV for the longer time are entirely known, while the ones that have been analysed for a shorted period, e.g. the external edges of the FOV, are only partially known.

5 As defined in Section 4.2.1, the knowledge of an area can assume values in the interval $[0, 1]$, where 1 means that the air vehicle has complete acquaintance with the area.

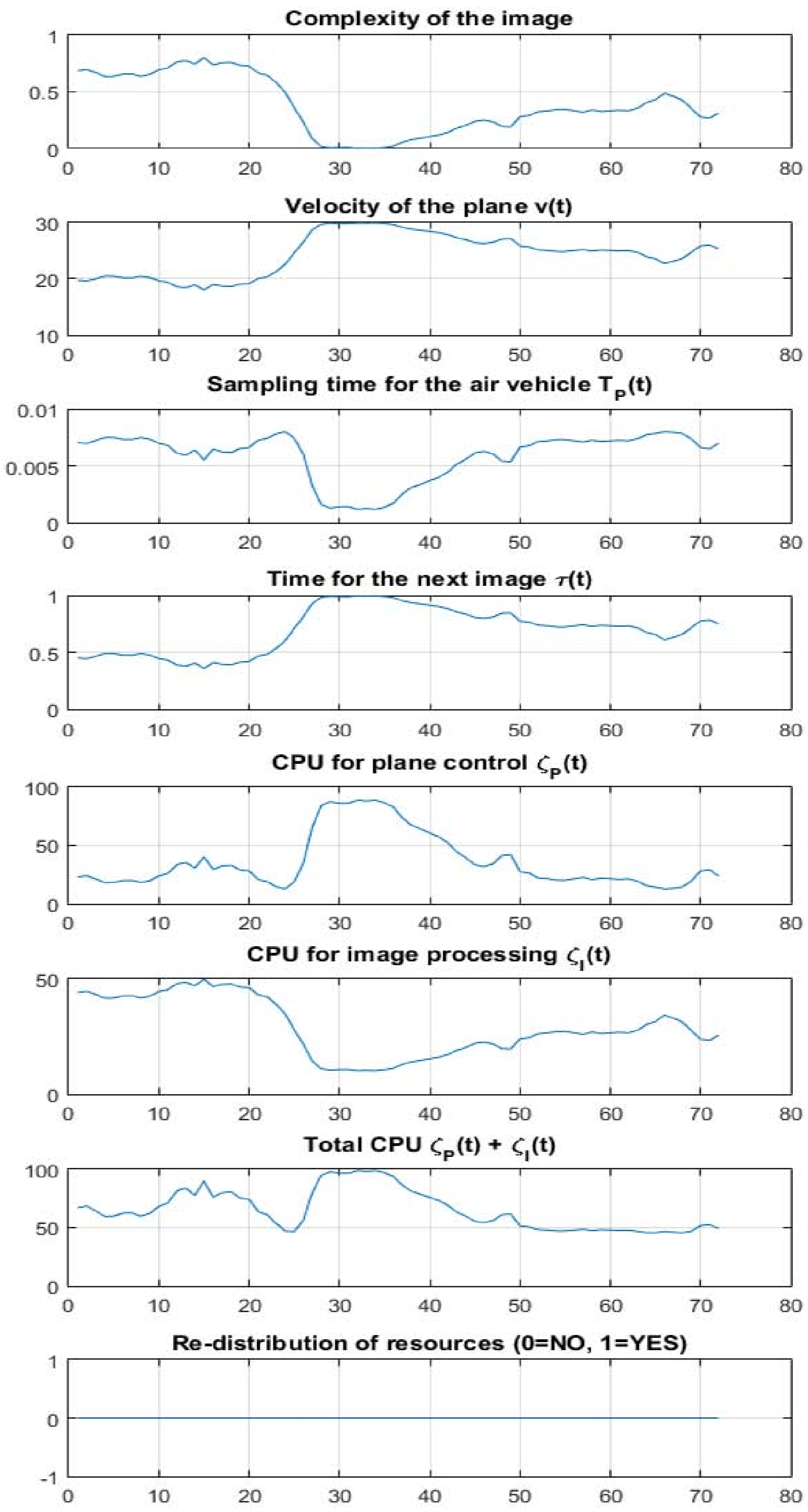


Figure 73.: Variations of the parameters according to complexity.

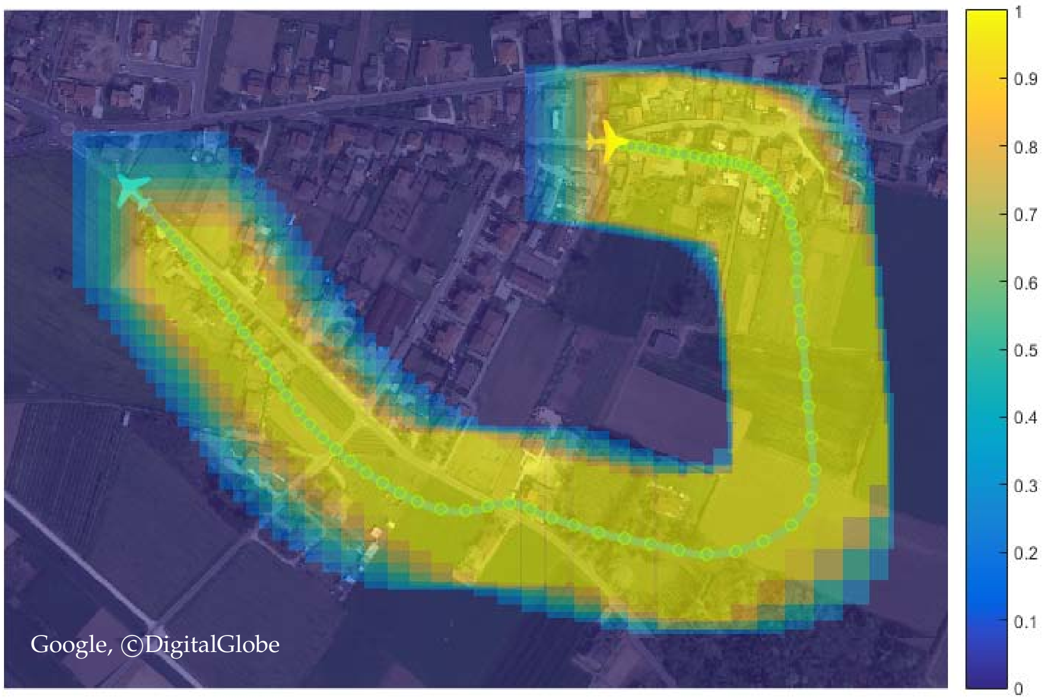


Figure 74.: Output of the learning function $l(\cdot)$: areas in yellow are completely known, while the ones in blue are still unknown.

Other important informations to take into account:

- The total distance covered by the UAV is $d = 1272$ m;
- The number of times that the scheduling and path planning algorithm has been performed is $N = 72$: clearly this data is not fixed, given that the parameter τ is variable. This value also corresponds to the number of photos of the environment captured by the plane.

We also wanted to study the case with fixed parameters, i.e. where v, T_p and τ do not vary with the complexity. The chosen fixed values are[6]

$$v = 22.5 \text{ m/s} \qquad T_p = 0.004 \text{ s} \qquad \tau = 0.6 \text{ s}$$
$$\zeta_P = 40\% \qquad \zeta_I = 35\ \%$$

that gives as a result the trajectory of Figure 75.
In this situation the UAV will travel 1134 m and the path-planning algorithm will be performed 84 times (but there is no need to run also the scheduling algorithm). Regarding the output of the image processing task, the UAV will explore 308404 pixels, i.e. 0.9915% of the original map, with a mean value of 0.7839. Of them, 52.222 % are completely known (with a value equal to 1), and that corresponds to 0.5178% of the total environment.

6 The values have all been chosen, for uniformity, in the middle of their interval.

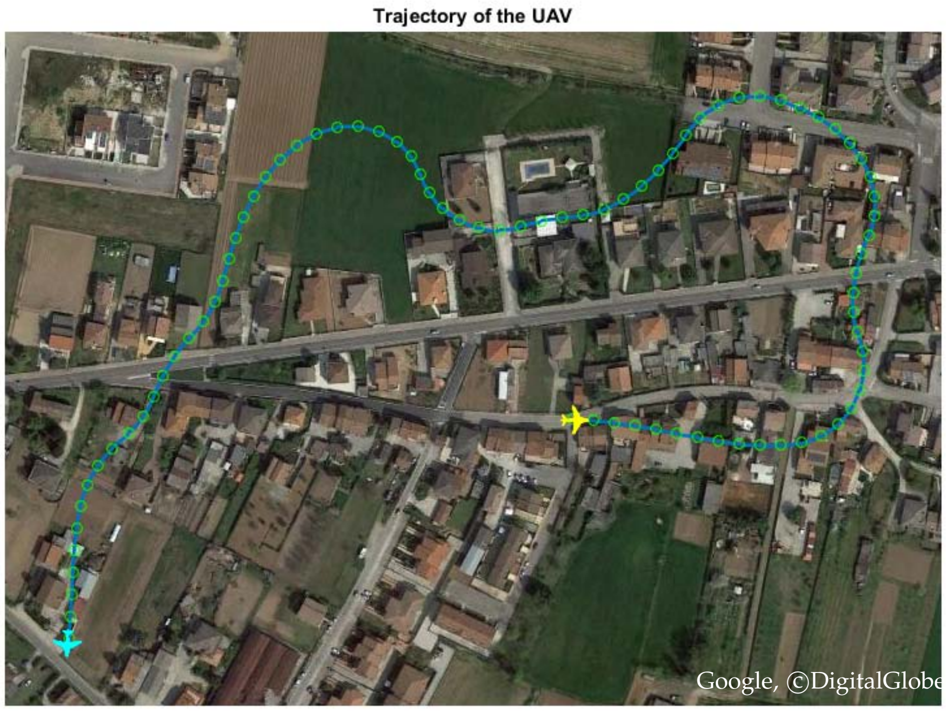


Figure 75.: Trajectory of the UAV with fixed sampling.

It is clearly not possible to make a direct comparison between the two methods presented, because the path planning algorithm designs different routes for the two of them. It is besides inappropriate to try and apply the same path on the two situations, since there would not be consistency with the scheduling algorithm: during every test run, a new line to follow, hence a new path, is chosen every τ seconds, a value that differs from simulation to simulation. To force a specified path, which has been previously designed keeping in mind some particular values of τ, means going against its possibility to vary, thus compromising the goal of RT scheduling. Even so, we can observe that the distance travelled by the UAV and the percentage of explored territory are both higher in the first case, which uses the designed scheduling algorithm, instead of the one with fixed parameters. This behaviour consistently occurs during various simulations ($\simeq 150$), with the scheduling algorithm that allows the air vehicle to travel, on average, 150 meters more every 30 seconds, and to explore 0.1% more of the environment. Furthermore, and most notably, these better results are achieved by using an equal or smaller amount of computational resources. This is why it is safe to say that a scheduling algorithm that allows for variable sampling rate in the UAV control, and an image processing algorithm with variable execution time, performs *on average* better of one with fixed quantities.

To end this chapter, one last example is taken into account, in which a re-distribution of resources due to the scheduling algorithm does occur. Clearly, for this to happen, the UAV must be travelling over areas with high complexity, as seen in Section 6.1, e.g. a city centre. This is the case of Figure 76, where the area is classified as highly complex, as one can appreciate from the fact that distance from two consecutive dots is always very small, meaning a small value of τ.

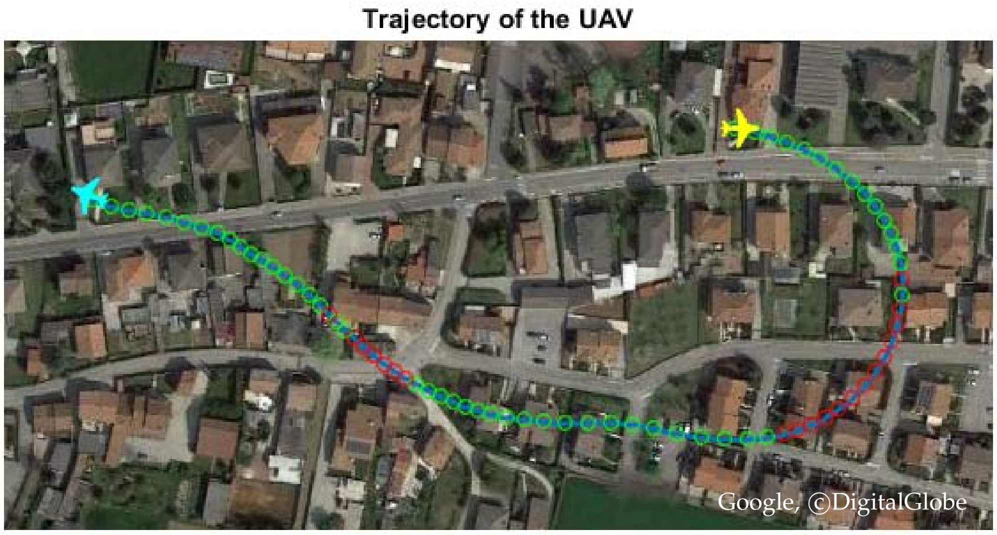


Figure 76.: Trajectory of the UAV in a city centre, i.e. in an area of high complexity; the red dots represent the time instants in which a re-allocation of resources happens.

The mean values of the parameters in this example are reported in Table 9. As one could expect from the theoretical development, an environment with a high complexity degree shows lower velocities, smaller sampling periods[7] and shorter periods τ, when compared with Table 8. The CPU's portion dedicated to the plane control is similar, but the most significant difference appears in the resources allocated for the image processing algorithm, that needs more CPU only when c is high (on the contrary, the maximum amount of CPU is requested for the plane control when the velocity is both low or high).

Table 9.: Mean values of the tasks' parameters in an area of high complexity.

$\overline{v}$ [m/s]	$\overline{T_P}$ [s]	$\overline{\tau}$ [s]	$\overline{\zeta_P}$ [%]	$\overline{\zeta_I}$ [%]	$\overline{\zeta_{TOT}}$ [%]
18.6	0.0059	0.41	35.9	46.7	82.6

As we just mentioned, in this case the total amount of computational resources needed sometimes exceeds ζ_{TOT}, which is why a re-arrangement of the parameters' value is necessary. Their new ones are obtained by using the formulas specified on the scheduling function $g(\cdot)$, widely explained in Section 5.2.2, and are summarized in Figure 77. This example is also useful to understand the different length of the time instants: in this case the simulation was only 30 s long, but it had $t = 74$ time instants, almost as much as in the previous example of Figure 72. This happens

7 The difference between the two values of $\overline{T_P}$ is not very accentuated because of the peculiar shape of its defining function.

because in this last situation the time instants are so much closer, since the UAV is traversing an area with complexity, while in the previous case they were on average more far from each other.

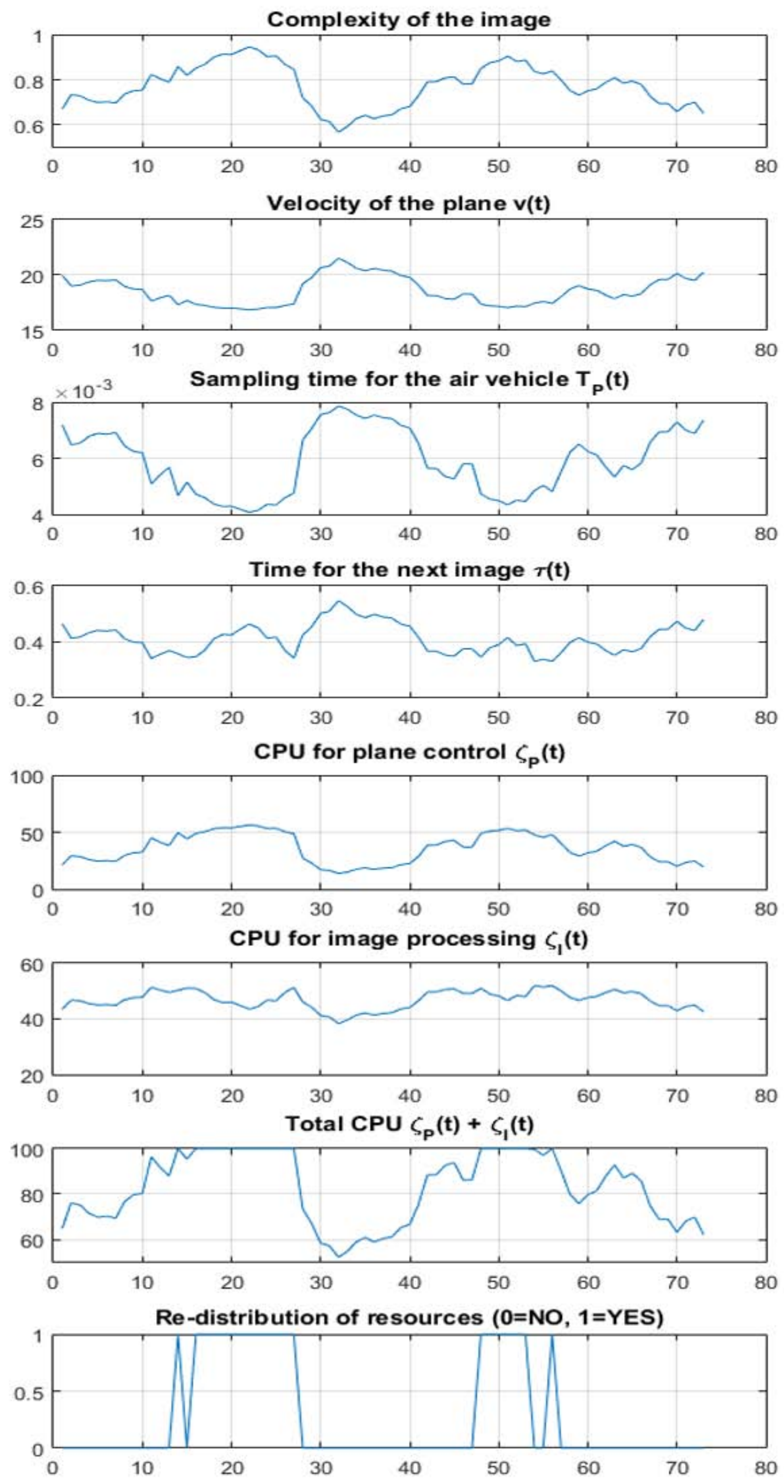


Figure 77.: Parameters' variations according to complexity, with re-allocation of resources.

7 CONCLUSION

The goal of this thesis was to develop a scheduling algorithm that allows two tasks with variable parameters to share the same processing unit, so with a limited amount of computational resources. The partitioning process had to be solved and adapted to the condition of every new time instant in real-time, based on a certain input signal, that influenced also the sampling rate of the two tasks. The process was first solved in a theoretical way, providing two solutions, one with prioritized tasks and one for jobs with the same importance, with the design of a scheduling algorithm.
The solution was eventually applied to a real-case scenario, concerning the flight of a UAV over the map of a territory with different kind of environments. The first task to be scheduled was the plane control itself, in which the sampling period $T_P(t)$ had to be adjusted every new time instant. The second task dealt with the image processing of the pictures taken from the underlying territory, and had to show variable execution time. This is strictly correlated to the period that had to pass between two consecutive photos $\tau(t)$, that has been chosen as the second variable parameter; during this time period the image processing of the current photo has to be concluded. This highlights one of the main peculiarities of this project: usually in literature, when dealing with real-time scheduling, the variable parameters taken into account are sampling frequencies, because the tasks are control-related. In this case only one task fall under that category (the UAV control), while the other is data processing-related, thus the characterizing parameter is unusual. The input signal that influenced both tasks is the image's complexity $c(I(t))$, i.e. its degree of interestingness. Based on this value, the scheduling function decides the air vehicle's velocity (and hence its sampling rate), and the value of τ, in a *self-triggered* fashion. At the expiration of the period τ new values are computed, in an *event-triggered* fashion. The variation of the parameters influenced the computational resources their related tasks needed, ζ_P and ζ_I. When their predicted values exceeded the maximum amount of available CPU, they had to be re-distributed through a function $g(\cdot)$. This causes a small deterioration of the performances of the tasks, but still guaranteeing that the overall process would function and operate without missed deadlines or system failures.
The realistic scenario has then been implemented: the air vehicle was modelled through its kinematics equations (Chapter 3), and the image processing algorithm though a learning function (Chapter 4). We ran simulations of the system with fixed parameters, then added the scheduling function to highlight the differences in the

two models. Unfortunately, it is not possible to draw definitive conclusions from a direct comparison, because the path that the UAV follows in the two situations is not the same, and it influences the global performance. The imposition of an a-priori specified path for both is furthermore inappropriate, as it will defeat the goal of the scheduling algorithm. Nevertheless, some observations from a statistical point of view are possible: on most cases, indeed, a UAV equipped with the scheduling algorithm will travel 150 meters more every 30 seconds, and will explore 0.1% more of the total map. Moreover, a smaller or equal amount of computational resources is usually requested, making this algorithm very useful in situations in which the utilization of fewer resources is a critical point, e.g. the flight of an air vehicle.

7.1 FUTURE DEVELOPMENTS

There are of course many ways in which this project can be enhanced.
The first and more important improvement consists certainly in the introduction of noises, that can drastically change the output of the scheduling algorithm. The principal sources of disturbances are due to the presence of wind, which can consistently modify the velocity of the plane. In this case the velocity v of the air vehicle no longer corresponds to its ground velocity v_g, and also the wind direction has to be taken into account. This will influence the kinematic equation of the UAV and, consequently, the way the path following algorithm operates, that has to be modify in order to be robust to wind. Furthermore, wind estimation is usually not reliable, because of inferior sensors and the necessity of high computational power to run advanced estimation algorithms. Another noise that can occur is due to corruptions when the sampling time is low: redundancy is not present, hence it does not provide the necessary robustness to the sensors reading. In addition, a user may experience more noises when the CPU usage is over 70 %, with the presence of lags.
Another advances could be application of the scheduling algorithm to n tasks, in order to make it usable in more general contexts. In this way it will be possible to take into account other periodic tasks, that can enhance the overall performance of the scenario.
A small device that could however improve the global flight experience of the UAV consists in introducing a feedback in the plane control[1]. This will allow for a real-time change of the control parameters of the plane, that will be adapted to the change of the sampling period, making the real-time scheduling problem even more complete.

1 This is possible only in the case the dynamic of the air vehicle are completely modelled in the scenario, and not disregarded as in this project.

A USE OF A CLASSIFIER

In this first method, we classify every new image that the vehicle takes into three categories: high, medium or low interest. The decision is made by extracting a series of features from the image and then comparing them to the features of every category: if they assume values that are closest to a high interest image, then it is classified in this way etc. To develop this method we need first to have a data set of images belonging to every category. This has been made by an a-priori classification from a human operator, having many images that resembles a real photograph taken by the air vehicle. It is now already clear why this algorithm do not always give the best results: this a-priori classification is made by a human operator, and so we need to withstand to what the operator decided and evaluated as interesting or not. All the test images that we had were evaluated into three categories: *High interestingness*, *medium interestingness* and *low interestingness*. We used this a-priori classification as training and validation data sets by splitting the images that we had.

The first step of this procedure consists in a preliminary operation, i.e. adjust the number of images in each category so that they will all have the same number of images and the training set will be balanced.
Eventually, we separated each set into a *training set*, that contains 30% of the images, and a *validation set*, with the remainder 70%. We also randomized the split to avoid biasing the results.
We extracted features from the training set by using the Matlab function `bagOfFeatures.m`, that can help us accomplish two tasks:

1. extracts *SURF features* from all images in all image categories, creating a "vocabulary" of SURF features representative of each image category;
2. constructs the visual vocabulary by reducing the number of features through quantization of feature space using *K-means clustering*.

SURF features

For a long time, keypoint detection and description was made through the SIFT algorithm, that guaranteed good results but long computational time. In 2006 Bay et al. [ref] published the paper "SURF: Speeded Up Robust Features", introducing indeed the SURF algorithm, a speeded-up version of SIFT. The main difference between the two algorithm is the way they approximate Laplacian of Gaussian and the use of wavelet transforms for orientation assignment. Another important improvement is the use of sign of Laplacian (trace of Hessian Matrix) for underlying interest point, that adds no computation cost since it is already computed during detection. Analysis shows it is 3 times faster than SIFT while performance is comparable to SIFT.

K-means clustering

K-means clustering is a method of vector quantization, that aims to partition n observations into k clusters, in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. By default, the Matlab function divides all the features in 500 clusters.

Next, encoded training images from each category are fed into a classifier training process, that relies on the multiclass linear SVM classifier.
Having now the trained classifier we can evaluate its performance. First, we can make a *sanity check* by testing it with the training set. We obtain the following confusion matrix in Table 10:

Table 10.: Confusion matrix for the Training Set.

	High	Medium	Low
High	1.00	0.00	0.00
Medium	0.08	0.92	0.00
Low	0.00	0.00	1.00

We can see that we have ones, or values really close to one, on the main diagonal, i.e. a near perfect confusion matrix.
In the next step we evaluate the classifier on the Validation Set, which was not used during the training. We use the `evaluate` function to test how good the classification performed, observing the returned confusion matrix, shown in Table 11.

Table 11.: Confusion matrix for the Validation Set.

	High	Medium	Low
High	0.96	0.04	0.00
Medium	0.46	0.29	0.25
Low	0.04	0.13	0.83

It is clear that the results are not good, in particular the images belonging to the category of *"medium interestingness"* are often confused and classified into the other two categories. The average accuracy is 69%, which is unsatisfactory and inadequate for our goal.
We made another attempt with this method, this time using five categories of images

instead of only three, each one containing images of decreasing complexity, to check if the performances of the classifier improved. Unfortunately, this did not happened and the results were even worse, returning the confusion matrix in Table 12 for the Validation Set. In this case the average accuracy is 51%, and this disappointing results are probably caused by the difficulty of the a-priori classification into the five categories, as mentioned previously, that has to withstand the personal opinion of the human operator who is in charge if this classification.

Table 12.: Confusion matrix for the Validation Set, using five categories of images.

	High	High/Medium	Medium	Medium/Low	Low
High	0.89	0.11	0.00	0.00	0.00
High/Medium	0.33	0.33	0.11	0.11	0.11
Medium	0.22	0.11	0.22	0.00	0.44
Medium/Low	0.11	0.00	0.22	0.56	0.11
Low	0.04	0.13	0.83	0.44	0.56

BIBLIOGRAPHY

[1] Seto D., Lehoczky J.P., Sha L. and Shin K.G. (1996). "On task schedulability in real-time control systems". In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 13-21, Los Alamitos, CA.

[2] Wang X. and Lemmon M.D. (2009). "Self-Triggered Feedback Control Systems With Finite-Gain $\mathcal{L}_2$ Stability". In *IEEE Transactions on Automatic Control*, Volume:54 , Issue: 3, pages 452 - 467

[3] Velasco M., Martì P. and Fuertes J.M. (2003). "The self triggered task model for real-time control systems". In *24th IEEE Real-Time Systems Symposium*, pages 67-70.

[4] Heemels W.P.M.H., Johansson K.H. and Tabuada P. (2012). "An introduction to event-triggered and self-triggered control". In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 3270 - 3285, Maui, HI.

[5] Årzèn K.E., Cervin A., Eker J. and Sha L.(2000). "An introduction to control and scheduling co-design". In *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, Volume 5, pages 4865 - 4870, Sydney, NSW.

[6] F. Lindh, T. Otnes and J. Wennerström. "Scheduling Algorithms for Real-Time Systems". Mälardalens University, Sweden.

[7] Singh A., Jeffay K. (2007). "Co-Scheduling Variable Execution Time Requirement Real-Time Tasks and Non Real-Time Tasks". In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 191-200, Pisa.

[8] M. Caccamo, G. C. Buttazzo and D. C. Thomas (2005). "Efficient reclaiming in reservation-based real-time systems with variable execution times". In *IEEE Transactions on Computers*, Volume 54, no. 2, pages 198-213.

[9] M. Hu, J. Luo, Y. Wang, M. Lukasiewycz and Z. Zeng (2014). "Holistic Scheduling of Real-Time Applications in Time-Triggered In-Vehicle Networks". In *IEEE Transactions on Industrial Informatics*, Volume 10, no. 3, pages. 1817-1828.

[10] J. Eker, P. Hagander and K. Årzén (2000). "A feedback scheduler for real-time controller tasks". In *Control Engineering Practice*, Volume 8, pages 1369-1378.

[11] A. Cervin, J. Eker, B. Bernhardsson and K. Årzén (2002). "Feedback–Feedforward Scheduling of Control Tasks". In *Real-Time Systems*, Volume 23, Issue 1/2, pages 25-53.

[12] G. C. Buttazzo, G. Lipari, M. Caccamo and L. Abeni (2002). "Elastic Scheduling for Flexible Workload Management". In *IEEE Transactions on Computers*, Volume 51, Issue, pages 289-302.

[13] H. Kushner and L. Tobias (1969). "On the stability of randomly sampled systems". In *IEEE Transactions on Automatic Control*, Volume 14, no. 4, pages 319-324.

[14] K. E. Årzén (1999). "A Simple Event-Based PID Controller". In *14th IFAC World Congress*.

[15] A. Anta and P. Tabuada (2010). "To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems". In *IEEE Transactions on Automatic Control*, Volume 55, no. 9, pages 2030-2042. 2010.

[16] T. Gommans, D. Antunes, T. Donkers, P. Tabuada and M. Heemels (2014). "Self-triggered linear quadratic control". In *Automatica*, Volume 50, Issue 4, Pages 1279-1287.

[17] D. Jung, J. Ratti and P. Tsiotras (2009). "Real-time Implementation and Validation of a New Hierarchical Path Planning Scheme of UAVs via Hardware-in-the-Loop Simulation". In *Unmanned Aircraft Systems*, pages 163-181, Springer Netherlands.

[18] H. Castañeda, O. S. Salas-Peña, J. de Leòn-Morales (2013). "Robus Autopilot for a Fixed Wing UAV Using Adaptive Super Twisting Technique". In *6th International Conference on Physics and Control PHYSCON 2013*, San Luis Potosí, México.

[19] Y. Kang and J. K. Hedrick (2009). "Linear Tracking for a Fixed-Wing UAV Using Nonlinear Model Predictive Control" .In *IEEE Transactions on Control Systems Technology*, Volume 17, No. 5, pages 1202-1210.

[20] H. Chao, Y. Cao and Y. Chen (2007). "Autopilots for Small Fixed-Wing Unmanned Air Vehicles: A Survey". In *International Conference on Mechatronics and Automation*, Harbin, 2007, pages 3144-3149.

[21] P. B. Sujit, S. Saripalli and J. B. Sousa (2014). "Unmanned Aerial Vehicle Path Following: A Survey and Analysis of Algorithms for Fixed-Wing Unmanned Aerial Vehicles". In *IEEE Control Systems*, Volume 34, Number 1, pages 42-59.

[22] M. Z. Shah, R. Samar and A. I. Bhatti (2015). "Guidance of Air Vehicles: A Sliding Mode Approach". In *IEEE Transactions on Control Systems Technology*, Volume 23, Number 1, pages 231-244.

[23] J. Fang, C. Miao, Y. Du (2012). "Adaptive nonlinear path following method for fix-wing micro aerial vehicle". In *Industrial Robot: An International Journal*, Volume 39, Issue 5, pages 475 - 483.

[24] D. Chaudhuri, N. K. Kushwaha and A. Samal (2012). "Semi-Automated Road Detection From High Resolution Satellite Images by Directional Morphological Enhancement and Segmentation Techniques". In *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, Volume 5, Issue 5, pages 1538-1544.

[25] A. Mukherjee, S. K. Parui, D. Chaudhuri, B. B. Chaudhuri and R. Krishnan (1996). "An efficient algorithm for detection of road-like structures in satellite images". In *Pattern Recognition, Proceedings of the 13th International Conference*, Vienna, Volume 3, pages 875-879.

[26] S. Y. Arafat, A. Y. Butt and N. Liaqat (2011). "Automatic road detection using MCSC". In *Multitopic Conference (INMIC), 2011 IEEE 14th International*, Karachi, pages 126-131.

[27] V. Hoang, D. Caceres Hernandez, A. Filonenko and K. Jo (2015). "Path Planning for Unmanned Vehicle Motion Based on Road Detection Using Online Road Map and Satellite Image". In *Computer Vision - ACCV 2014 Workshops*, pages 433-447, Springer.

[28] R. Cunha, D.J. Guerreiro Tomé Antunes, P. Gomes and C.J. Silvestre (2006). "A path-following preview controller for autonomous air vehicles". In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, Keystone, Colorado, pages 1-21.

[29] `http://wiki.paparazziuav.org/wiki/Main_Page`

[30] S. Park, J. Deyst, and J. P. How (2007). "Performance and Lyapunov Stability of a Nonlinear Path Following Guidance Method". In *Journal of Guidance, Control, and Dynamics*, Volume 30, No. 6, pages 1718-1728.

[31] A. Ralescu and J. Shanaham (1999). "Perceptual Organization for Inferring Object Boundaries in an Image". In *Pattern Recognition*, Volume 32, pages 1923-1933.

[32] I. Mario, M. Chacon, D. Alma, and S. Corral (2007). "Image complexity measure: A human criterion free approach". In *Proc. Annual Meeting of the North American Fuzzy Information Processing Society*, pages 241–246.

[33] J. Zou and C. Liu (2010). "Texture Classification by Matching Co-occurrence Matrices on Statistical Manifolds". In *10th IEEE International Conference on Computer and Information Technology*, pages 1-7.

[34] Y. Chen, J. Duan, Y. Zhu, X. Qian and B. Xiao (2015). "Research on the image complexity based on neural network". In *2015 International Conference on Machine Learning and Cybernetics*, Guangzhou, pages 295-300.

[35] J. Perkiö and A. Hyvärinen (2009). "Modelling Image Complexity by Independent Component Analysis, with Application to Content-Based Image Retrieval". In *Proceedings of the 19th International Conference on Artificial Neural Networks: Part II*, pages 704-714.

[36] `http://www.astro.cornell.edu/research/projects/compression/entropy.html`

[37] T. Lindeberg (2001). "Edge detection". In Hazewinkel, Michiel, *Encyclopedia of Mathematics*, Springer.

[38] J. Matas, O. Chum, M. Urban and T. Pajdla (2002). "Robust Wide Baseline Stereo from Maximally Stable Extremal Regions". In *Proceedings of the British Machine Conference*, pages 36.1-36.10, David Marshall and Paul L. Rosin, editors.

[39] `http://www.micc.unifi.it/delbimbo/wp-content/uploads/2011/03/slide_corso/A34%20MSER.pdf`

[40] E. Rosten and T. Drummond (2006). "Machine learning for high-speed corner detection". In *European conference on computer vision*, pages 430-443.

[41] S. Leutenegger, M. Chli and R. Y. Siegwart (2011). "BRISK: Binary Robust invariant scalable keypoints". In *2011 International Conference on Computer Vision*, Barcelona, pages 2548-2555.

[42] Polycarpou M., Passino K., Yang, Y. and Liu Y. (2003). "Cooperative Control Design For Uninhabited Air Vehicles". In *Cooperative Control: Models, Applications and Algorithms*, pages 283-321, Springer US.

[43] S. G. Ahrens (2008). "Vision-Based Guidance and Control of a Hovering Vehicle in Unknown Environments". Master of Science in Mechanical Engineering, Massachusetts Institute of Technology.