UAV-UGV Rendezvous and AI Planning for Robotic Industrial Inspection Based on ROS and WARA-PS

Hicham Mohamad

LUND
UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6315
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2026

To the memory of my mother Abla Jandal
and my supervisor Professor Anders Robertsson,
for their endless support and love.
May they rest in peace.

Abstract

In the past, robots were goal-based agents and experienced a more *limited world*. These robots needed to be instructed on each individual step or action to complete a larger task. They worked in static and well-modelled environments and run predefined behaviours where their missions consisted of *repeating a single task* (e.g. assembling and vacuum cleaning). Nowadays the world of robots and autonomous vehicles is growing continuously making our environment more robotised due to advanced research and development in electronics (storage and computation power) and Artificial Intelligence (AI) algorithms.

In fact, by fully exploiting the robot capabilities and taking advantage of the AI algorithms we are changing our well-being and productivity in different sectors, particularly in industry, health-care and public safety. These *technological advances* opened up the opportunity for new applications. For instance, we have seen an increasing number of practical applications using *multi-robot systems* (or robot fleet) within industry, search and rescue (SAR), space exploration and other areas, to address specific mission scenarios where the control techniques are based on the field of *multi-agent networked control* in robotic networks. In particular, the powerful concept UAV-UGV cooperation where unmanned aerial vehicles (UAVs) and unmanned ground vehicles (UGVs) collaborate to create a more robust, efficient, and versatile solution for complex missions like disaster response, agriculture, or defence.

In this degree project, we present two distinct problems in the engineering fields of Automatic Control and AI Robotics. In the first part, we investigate how to solve a *Rendezvous landing problem* between a quadcopter (UAV) and a mobile robot (UGV). Next, in the second part we deal with the concept of *intelligent behaviour* that requires human-level cognitive capabilities of which a central problem is *selecting the action to do next* or the so-called deliberative acting. Thus, to enable such sophisticated deliberative skills, we investigate how to design the robot to properly solve an *AI planning problem* (also known as Task Planning) in Robotic Inspection domain. These two problems can be related and would complement each other in certain inspection missions, for example in construction/industrial scenarios, deploying such aerial and ground vehicles.

To this end, the goal of the project first part is to design an autopilot that can execute a Rendezvous landing algorithm, which is an important problem in *autonomous flight*. The main steps of the algorithm consist of: tracking, alignment, descent, retardation (shutting of propeller motors) and touch-down. Specifically, the autopilot's task is to force the quadrotor UAV "DJI Matrice 100" (a common research platform) to track a quadruped robot "Spot" (from Boston Dynamics) and subsequently land on it.

In this study, the dynamics of both quadrotor and quadruped systems were examined and presented, where the models are relatively simplified or necessarily approximated as first or second-order systems, in similar fashion as in various literature. Further, the Rendezvous algorithm uses *Robot Operating System* (ROS) to command and control the flight of the quadrotor UAV and the movement of the mobile ground robot. Substantially, the algorithm makes use of the *ROS interface* (including services, subscribed topics, and published topics) and runs a four-dimensional PID *position-tracking controller* implemented in Python to control the four basic movements of the quadrotor: thrust, roll, pitch and yaw. The objective of this controller is to compute and send correction commands to guide the UAV drone to track the target robot platform on the ground, as well as to accomplish the alignment of the acting unmanned vehicles in the horizontal plane and the landing operation at the end.

Simulated experiments are conducted using the ROS-based UAV simulator provided by *WARA-PS framework* (WASP Research Arena Public Safety), as well as using a two-wheeled simulated robot (TWR), which is a newly constructed URDF-based simulation model for simulating the movement of Spot robot in the rendezvous manoeuvre. The obtained results confirm the satisfactory performance of the proposed algorithm (implemented in Python) by illustrating the convergence graph using the PID position-tracking controller.

Further, for reinforcing the performance of this Rendezvous algorithm we additionally investigate other robust control strategies using the promising *Model Predictive Control* (MPC) approach, where theoretical development for three different advanced MPC-based methods is presented in this report, in which *interaction* between platforms can also be included in *cooperative manner*. However, the practical implementations of this second approach using MPC, including simulations and real-world flight tests and experiments, will be addressed in future work.

In the second work, the goal is to formulate and implement a planning model in *Planning Domain Definition Language* (PDDL) that can solve an inspection planning problem by generating a *sequence of composite (abstract) actions*. That is, as a result we obtain a *high-level* goal plan to perform certain tasks and achieve the desired objectives (like "spot-inspect" or "uav-inspect"). In fact, one possible solution plan to a small simulated inspection mission example is successfully obtained using our implemented PDDL-model, composed of domain and problem files, as input to a suitable off-the-shelf AI-based solver (task planner). Next, this generated high-level PDDL solution plan is expanded (refined) into executable *Task Specification*

Tree (TST), using TST Factory in WARA-PS, to obtain at the end a possible cooperative goal plan consisting of *detailed elementary (concrete) actions*, like "move-to" and "fly-to". Testing the applicability of the resulting PDDL plan of actions in real-world problems will be performed in future research work.

Sammanfattning

Nuförtiden växer världen av robotar och autonoma fordon kontinuerligt, vilket gör vår miljö mer robotiserad. I synnerhet är UAV-UGV-samverkan ett kraftfullt koncept, vilket avser samarbete mellan obemannade luftfarkoster (UAV) och obemannade markfordon (UGV), där drönare och markrobotar arbetar tillsammans för att övervinna individuella begränsningar. Detta skapar en mer robust, effektiv och mångsidig lösning för komplexa uppdrag som katastrofinsatser, jordbruk eller försvar. Denna avhandling adresserar två huvudsakliga problemområden: rendezvous-landning av en quadcopter på en mobil markfarkost, samt AI-planering (även känt som uppgiftsplanering eller task planning på engelska) för industriell robotinspektion.

I detta syfte är målet med projektets första del att utforma en autopilot som kan tvinga quadrotorn UAV att spåra den rörliga markroboten och därefter landa på den. Denna rendezvous-algoritm använder Robot Operating System (ROS) för att styra och kontrollera flygningen av drönaren och rörelsen av den mobila markroboten. I huvudsak kör algoritmen en fyrdimensionell PID-positionsspårnings regulator (position-tracking controller) implementerad i Python för att styra quadrotorns fyra grundläggande rörelser: dragkraft (thrust), roll, pitch och gir (yaw). Simulerade experiment utförs med den ROS-baserade UAV-simulatorn, tillhandahållen av WARA-PS ramverket (WASP Forskningsarena - Allmänna Säkerheten), tillsammans med en nykonstruerad URDF-baserad modell för simulering av en tvåhjulig robot (TWR). De erhållna resultaten bekräftar den föreslagna algoritmens tillfredsställande prestanda genom att illustrera konvergensdiagrammet. Vidare, för att förstärka prestandan hos denna rendezvous-algoritm, undersöker vi dessutom den teoretiska utvecklingen av tre olika avancerade metoder med hjälp av den lovande metoden modell-prediktiv reglerteknik (MPC). Den praktiska implementeringen av detta andra tillvägagångssätt med hjälp av MPC kommer dock att behandlas i framtida arbete.

Därefter, i den andra delen, behandlar vi konceptet intelligent robotbeteende eller så kallad avsiktligt agerande (deliberative acting). För att möjliggöra sådana sofistikerade deliberativa färdigheter undersöker vi således hur man utformar roboten för att korrekt lösa ett AI-planeringsproblem (uppgiftsplanering) inom

domänen robotiserade industriella inspektioner. Således blir målet i det här arbetet att formulera och implementera en planeringsmodell i Planning Domain Definition Language (PDDL) som kan lösa ett inspektionsplaneringsproblem genom att generera en sekvens av sammansatta abstrakta handlingar (en högnivå-målplan). Som ett resultat erhålls en möjlig lösningsplan för ett exempel på ett litet simulerat inspektionsuppdrag framgångsrikt genom att använda vår implementerade PDDL-modell som indata till en lämplig "off-the-shelf" AI-baserad lösare (uppgiftsplanerare). Därefter expanderas (förfinas) denna genererade PDDL-lösningsplan på hög nivå till ett körbart Task Specification Tree (TST), med hjälp av TST Factory i WARA-PS, för att i slutändan erhålla en möjlig kooperativ målplan bestående av detaljerade elementära (konkreta) handlingar. Att testa tillämpligheten av den resulterande PDDL-handlingsplanen i verkliga problem kommer att utföras i framtida forskningsarbete.

Acknowledgements

At the start, I gratefully acknowledge all the people that contributed directly or indirectly to this work. In particular, I am deeply indebted to my main supervisor Professor Anders Robertsson for providing constant support and advice during this project and throughout my time at the department of Automatic Control in LTH, at Lund University. Unfortunately, he passed away before completing this project work: "*May you rest in peace Super Anders, you will always be in our hearts*".

I also express my gratitude to my co-supervisors Prof. Björn Olofsson and Prof. Yiannis Karayiannidis together with the examiner Prof. Karl-Erik Årzén from the same department of Automatic Control at LTH, for their supports and for taking over the roles of supervision and examination respectively at the end of this degree project, subsequently after the lost of my main supervisor. I would also like to thank Gregory Austin for the support in setting up the platform of Spot robot.

Then, my sincere appreciation goes also to Dr. Tommy Persson, Dr. Piotr Rudol and Dr. Mariusz Wzorek, researchers at *Artificial Intelligence and Integrated Computer Science* (AIICS) division at Linköping University (LiU), for the constructive meeting sessions and discussions that helped me clarify my ideas and improve my understanding during this work. Many thanks to all of you as well !

Finally, I want to thank Kim Amadeius, Wassim Mohamad, Grazia and Massimiliano for their friendly and genuine support and encouragement during the challenging time of this degree project.

Hicham Mohamad
December, 2025

Contents

1

Introduction

By introducing the new technological advances in electronics (storage and computation power) and Artificial Intelligence (AI) algorithms, it becomes usual the generation and execution of *automated missions* and coordination of tasks using5*multi-robot systems* within the sectors of industry, search and rescue (SAR), space exploration and other similar areas of common concern.

In particular, it becomes useful and of growing interest the *UAV-UGV cooperation*, denoting Unmanned Aerial Vehicle (UAV) and Unmanned Ground Vehicle (UGV) collaboration, where drones and mobile ground robots work together to overcome individual limitations and carry out important complex tasks like *inspection and monitoring* in industry and public safety. Searching the literature we find in fact that this *multi-agent cooperation* will reduce the time to perform inspection, surveying and maintenance tasks in industry. Additionally, this cooperative system will also minimize the challenges to accomplish urgent tasks in hazardous and difficult-to-access environments, especially with the deployment of the quadrotor flying robots (UAVs) that have demonstrated great success in remote sensing and visual inspection tasks.

This thesis exhibits two interesting and practical applications of multi-robot scenarios that are one of the research area in robotics. That is, we will deal with *Rendezvous landing manoeuvre* and *AI task planning* deploying two robotic platforms, a quadrotor UAV (DJI Matrice 100) and a ground mobile robot (quadruped UGV Spot), as illustrated in Figure 1.1.

In these scenarios, the *cooperation* is an important feature to guarantee crucial safe interaction between the robots acting in this multi-agent system. Moreover, the *diversity of the operational skills* of these cooperating agents is another important feature where cooperating robots will be able to perform certain tasks a single robot would not be able to do it itself. These features, diversity of operational skills and cooperation, will improve the flexibility and robustness of the multi-agent systems when solving problems such as Rendezvous and AI planning.

In the following sections we introduce an overview that outline these two problems separately. These two parts of the project will be solved using existing tools (a UAV simulator and a planning solver) provided by *WASP Research Arena Public*

Safety (WARA-PS Core System) [WARA-PS, 2024], which is a ROS-based software framework. More information about WARA-PS can be found in the paper [Andersson et al., 2021].

Figure 1.1 illustrative example of teaming the quadruped Spot robot with three quadcopters.

1.1 Part 1: Rendezvous Landing Problem

For the first part of this project, we find in literature that Rendezvous landing problem, i.e., landing of quadrotor UAV on moving platform on the ground, has been considered in different manners using advanced algorithms, in which we distinguish between cooperative, centralized, distributed and decoupled control, or other flavours. This Rendezvous landing manoeuvre is practical for UAVs that cannot remain airborne because of battery life or other limitations. In other situations, it can be that UAVs need to coordinate certain task with the ground vehicle and therefore need to land back at the selected position after completing a specific mission, for instance in inspection or search and rescue scenarios.

It was reported that the dynamics of such multi-rotor UAVs, like the quadrotor used in this project work, are inherently underactuated, unstable, and nonlinear [Mahony et al., 2012]. Hierarchical controller structure (cascade structure) is commonly suitable for this aerial vehicle which is usually separated into two cascaded controllers. For solving the Rendezvous landing problem, we implement the designed *autopilot algorithm* in the outer loop, whereas the inner-loop corresponds to the firmware low-level attitude controller provided by the manufacturer.

Research and Related Work - Rendezvous Landing

Several approaches and research trends have been involved in the development of autonomous unmanned aerial vehicles (UAVs) and much of the recent research ac-

tivity in the field of automated landing has been considered to solve Rendezvous landing problems using UAVs and robots technology. In particular, various control strategies have been applied to control the quadrotor UAV in several previous project works such as in [Bouabdallah and Siegwart, 2007; Mellinger et al., 2010; Bloesch et al., 2010; Kamel et al., 2017b; Kamel et al., 2017a; Lee et al., 2012; Muskardin et al., 2017; Persson and Wahlberg, 2019; Persson et al., 2017; Bereza et al., 2020; Persson, 2021; Fairchild and Harman, 2017; Hoenig and Ayanian, 2017]. In the following, we discuss the most relevant existing research works and connections related to our project, that we have explored and could contribute in some way to the development of this thesis and to future work. Motivating by the scientific challenge in UAV design and control and the lack of experience in the field of autonomous flying robots, we begin our project by exploring some important literature in quadrotor projects, as in [Bouabdallah and Siegwart, 2007; Mellinger et al., 2010; Bloesch et al., 2010; Kamel et al., 2017b; Kamel et al., 2017a], to firstly enhance our knowledge in modelling and control of autonomous four rotor helicopter (quadrotor) before tackling the problem of Rendezvous landing manoeuvre in our project.

A full Control system for quadrotor UAVs, structured in six different controllers, together with the simulation model are presented in [Bouabdallah and Siegwart, 2007] as parts of OS4 project at the Autonomous Systems Lab at ETHZ. In this paper, they propose the so-called *Integral Backstepping* (IB) as a single control approach for attitude, altitude and position. This IB control approach is a combination between PID and Backstepping technique. Authors argued that this control structure permits a powerful and flexible control as the Backsteppping technique is well suited for the cascaded structure of the quadrotor dynamics and guarantees asymptotic stability, while the integral action cancels the steady state errors. Moreover, other auxiliary controllers for autonomous take-off and landing, obstacle avoidance and way-point following are demonstrated in simulation.

The work in [Mellinger et al., 2012] describes a method to enable a quadrotor UAV (Hummingbird, from Ascending Technologies) to robustly perch, land or takeoff from ledges or horizontal flat pads at selected positions. This capability would be extremely useful, for example, to offer a good vantage point for environmental and military applications carrying out monitoring and surveillance missions. In brief, the design of the authors (from GRASP Lab at the university of Pennsylvania) focuses on two main contributions. First, the control algorithm to enable perching landing manoeuvre which requires the quadrotor to recognize suitable perching pads, control the *relative position and orientation* with respect to the pad and then finally grasp the pad with claws. Second, they address the design of a gripping mechanism that is used to grasp the pad with claws after landing on it . Specifically, in order to solve the primary work of perching landing, the *VICON motion capture system* is used to track the landing surfaces and the pose (attitude and position) of the flying quadrotor. A PID-based position controller, with roll and pitch angles as inputs, is integrated with this VICON system to compute the PID feedback on the position

error to control the UAV for hovering or for tracking the three-dimensional trajectories. Yet again, a nesting of control loops composed of two loops for position and attitude is the general control approach followed in this paper and the control software is accessed via ROS framework.

Other interesting research works in the field of *flight control* and *autonomous mission execution* are addressed by researchers at the Autonomous Systems Lab, at ETH Zurich. The paper in [Bloesch et al., 2010] presents a vision-based approach describing the ability of a Micro Aerial Vehicle (MAV) to navigate in unexplored and unstructured environment (GPS-denied environment) using a single camera.

A extensive survey is presented in [Kamel et al., 2017b] describing the design and implementation of various Model Predictive Control (MPC) strategies, linear and nonliner MPC, for trajectory tracking applied to both multi-rotor UAVs and fixed-wing UAVs. The general theory behind MPC is reviewed including the integration of these MPC trajectory tracking controllers into ROS framework together with an open source C++ implementation of both controllers. Additionally, in [Kamel et al., 2017a] authors perform a *detailed comparison* between the classical Linear MPC and the more advanced Nonlinear MPC that considers the *full system dynamics* of the MAV to emphasis their benefits and drawbacks in terms of performance improvement, disturbance rejection and computation effort. The experimental results of the quantitative comparison (rise time and overshoot) during hovering, step response and aggressive trajectory tracking under external disturbances clearly highlights that the Nonlinear MPC outperforms the Linear MPC. This conclusion is explained by the exploitation of the full system dynamics in Nonlinear MPC, and particularly by employing the *full thrust command* coupled with the vehicle lateral motion. The aforementioned LMPC and NMPC experiments have been implemented and evaluated on *Asctec NEO hexacopter* (equipped with an Intel 3.1 GHz i7 Core processor, 8 GB RAM).

A commonly used nonlinear controller was proposed by [Lee et al., 2012] for Rendezvous Landing manoeuvre, that is, Autonomous Landing of a vertical takeoff and landing (VTOL) UAV on a moving platform. This situation would occur for example when landing on a moving ship or truck, similar to our thesis project and to the situations described below in [Muskardin et al., 2017; Persson and Wahlberg, 2019; Persson et al., 2017; Persson, 2021; Bereza et al., 2020]. In this paper, a *Visual Servoing* algorithm using an onboard camera is integrated with the *adaptive sliding mode controller* for precision control of the quadrotor while landing on a moving target. This visual servoing technique, referred to as Image Based Visual Servoing (IBVS), involves using information from the image to directly control the degrees of freedom (DOF) of the robot. Authors demonstrate that their IBVS-guided method is capable of providing the necessary precision for tracking and landing tasks while GPS-based technique accuracy is unsuitable for miniature UAVs to perform such precision tasks. As they argued in the paper, the accuracy of GPS receivers is measured in meters, assuming the signal is not block. Moreover, an external *VICON position sensor* is used to provide position guidance to the UAV,

when the target is not detected during the patrol phase in the initial approach.

The paper in [Muskardin et al., 2017] discusses a landing System to increase payload capacity and operational availability of *High Altitude Long Endurance (HALE) UAVs*. Such HALE UAVs (fixed-wing airplanes) are intended for using in the stratosphere at altitudes in the order of 20km, and they draw increasing attention in the recent years for various applications including earth observation, atmospheric science and communication networks. In order to lower the empty weight and in turn increase the payload of the lightweight HALE UAVs the solution suggested in this paper is to eliminate the need for a *landing gear*, which is merely dead weight for most of the time, and adopt the concept of a *ground based landing gear*. Specifically, the core idea here is to have *landing platform* mounted on a car with a human driver equipped with the same real-time computer and sensor system (RTK-GPS, IMU) as present on the UAV. To increase the reliability and precision of the *relative state estimation system* a combination of RTK-GPS based networked state estimation and *optical multi marker tracking* (vision-based estimates) has been applied and tested showing the best performance. In fact, it is important to recall that the quality of the calculated *relative position* directly affects the achievable control precision. Moreover, authors propose an improved flight control system (FCS) based on energy principles to overcome the inherent coupling limitations of conventional SISO controllers. Such controller is commonly referred to as *total energy control* system (TECS) introduced in [LAMBREGTS, 1983]. Their autonomous landing manoeuvre has been divided into six distinct phases and controlled by a high-level *state machine logic*. In first step, the different aspects of the proposed cooperative control algorithms were analysed using a *realistic simulation environment* before attempting the real-world landing.

Yet, examples of Rendezvous autonomous landings related to Search and Rescue scenarios has already been studied in the scope of different research projects at the Department of Automatic Control at KTH (Royal Institute of Technology) in Stockholm. In particular, the research works presented in [Persson and Wahlberg, 2019], [Persson et al., 2017], [Persson, 2021] and [Bereza et al., 2020] consider autonomous unmanned ground and surface vehicles (UGV and USV) as moving *landing platforms* for the UAV drone in air, with close similarity to our Rendezvous problem. The project in [Persson and Wahlberg, 2019] presents a Model Predictive Control (MPC) approach for landing manoeuvre in marine environment on a autonomous boat. The other paper in [Bereza et al., 2020] considers the same marine environment to design and implement two *distributed MPC* algorithms demonstrating improved robustness towards communication breaks and the ability to use longer predicted horizons compared to the centralized MPC approach, as the authors stated. The research work in [Persson, 2021] concerns cooperative Rendezvous problem based on MPC framework in two specific scenarios: the landing of a fixed-wing drone on top of a moving ground carriage, as well as the landing of a quadcopter UAV on the deck of a boat, based on [Persson and Wahlberg, 2019; Persson et al., 2017]. Further, The performance of these two different landing systems are also

improved by employing a *variable horizon* MPC. Interestingly, these landing algorithms are subsequently implemented and analysed both in realistic simulations and in real-world outdoors flight tests through the WASP Research Arena (WARA-PS), that is composed of a large-scale *demonstrator arena* equipped with autonomous boats and drones and offer engineering support for developing and testing algorithms, as described in [Andersson et al., 2021] and [WARA-PS, 2024].

Now for completeness, after drawing inspiration from the explored research and literature discussed above, we briefly describe the main contributions commonly reviewed in the previous research projects that have been designed in our thesis in order to address the challenges of the Rendezvous landing implementation in our first work of this present thesis. Firstly, we choose to design and test the Rendezvous landing algorithm in simulation using PID-based position-tracking control with the thrust, roll, pitch and yaw rate as the control inputs. This target position-tracking controller is one of the main contributions of this algorithm. Through an *iterative process* of scanning the positions, the controller software uses the *relative position* error between the two robots to compute and send the correction commands in every iteration to the UAV to fly closer to the goal position on the ground robot. The algorithm second contribution is a *state machine* used to control the quadrotor based on its state of flight. These flight states included in the state machine are idle, takeoff, hover, flight, and descend. *Data acquisition* is the third contribution in our algorithm and consists of gathering necessary information about the other robot acting on the ground using ROS interface (topics and services).

To test the implemented algorithm and evaluate the Rendezvous performance, simulated experiments are conducted using the ROS-based UAV realistic simulator provided by *WASP Research Arena Public Safety* (WARA-PS). Moreover, a URDF-based model for a two-wheeled robot (TWR) is build for simulating the behaviour of the ground robot platform involved in the Rendezvous manoeuvre. The corresponding implementation on the physical robots system (real field experiments) will be conducted in future work.

Moreover, investigating the related works discussed previously, especially in [Persson and Wahlberg, 2019; Persson et al., 2017; Persson, 2021; Bereza et al., 2020], we can figure out that the promising Model Predictive Control (MPC) framework must be suited for a *robust Rendezvous manoeuvre*. In fact, when formulating the corresponding *optimal control problem* (OCP) in MPC we can add *robustness* to our algorithm thanks to the ability of representing necessary *constraints* and likely *disturbance* in the formulation. This functionality would be very useful in real-world experiments as well. Hence, we discuss a second attempt to reinforce the Rendezvous algorithm using another position-tracking control approach with MPC techniques. This MPC-based controller would robustly improve the rendezvous performance by rejecting disturbances and taking in consideration necessary constraints. The algorithm development and simulation of this second approach will be implemented in future work. For this optimization-based MPC approach, a simplified dynamic model of the multi-robot system is needed. The dynamics of the

internal attitude controller (inner loop controller in the cascaded structure) is also included in this model.

1.2 Part 2: AI Planning Problem

In the second part of the project work, we deal with an *automated planning problem* (also known as AI planning or task planning) in an autonomous inspection scenario to be carried out using UAVs and robots technology without human intervention. The approach we describe in this thesis is sufficiently general (high-level task planning) where modelling the environment for the robotic inspection task must be simple enough to allow the planner to manage the high-level reasoning and plan a variety of other tasks in the same setting.

Planning is an Artificial Intelligence technology that seeks to select and organise activities in order to achieve specific goals [Ghallab et al., 2004]. In other words, the purpose of planning is to synthesize *an organized set of actions* to carry out some activity [Ghallab et al., 2016a]. In this setting, robots need to "think first before acting" or "acting deliberately", that is, what action to do next?

Having said that, it is worth noting that this AI planning problem is in essence a *control problem*. In fact, the controller, that prescribes the "action to do next", is not learned from experience (*learning-based approach* as in Reinforcement Learning) or given by a programmer (*programming-based approach*) but is derived automatically from a model of the actions, sensors and goals, i.e., *model-based approach*. As stated in [Geffner and Bonet, 2013], three different approaches in Artificial Intelligence (AI) have been used to address this *action selection* problem. More details about this will be found in another chapter.

In general, with AI planning we encapsulate a *deliberative function* to make robots more intelligent and understand what they are doing. In this way, we may give different level of definitions to "robot control" represented by a hierarchical internal structure [1], as illustrated in Figure 1.2.

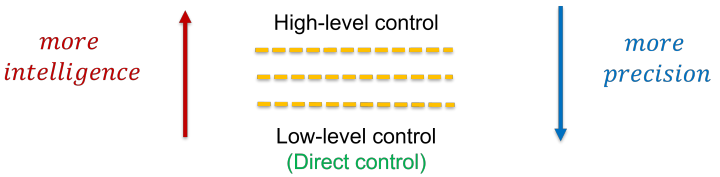


Figure 1.2 Different level of definitions may be given to robot control: Control system unit has a hierarchical internal structure: different but cooperating models, objectives, methods are used at the various control layers . (Courtesy: Alessandro De Luca, DIAG Sapienza)

[1] http://www.diag.uniroma1.it/~deluca/rob2_en/material_rob2_en.html

Yet, the motivation of this second part is to create a deliberative *planning model* for autonomous robots to achieve some level of *autonomy* in industrial inspection domain. Having a model of the planning problem written in *Planning Domain Definition Language* (PDDL-model) enables the usage of a wide variety of solvers (task planners), that would be in charge of ordering and executing the tasks (mission specification) by obtaining a *high-level solution plan*.

Research and Related Work - AI Automated Planning

For the past several years, the topic of AI Task Planning has captured more interest of researchers within the planning community especially when industry has been moving towards conducting remote monitoring, surveillance and inspection using drones and robots technology. First, to identify the best methodology and boost our knowledge in this evolving research field, various useful literature were explored like in [Russell and Norvig, 2010; Haslum et al., 2019; Geffner and Bonet, 2013; Ingrand and Ghallab, 2014; Ghallab et al., 2016a; LaValle, 2006]. In the literature, we distinguish between two main planning paradigms: *Single-Agent* and *Multi-Agent* planning problems. Then, There are two main approaches for solving Multi-Agent Planning (MAP) tasks: *centralized* and *distributed*.

Overall, the use of AI Planning in autonomous robots mission control is not new. Over the last years, several relevant research projects in the field of automated planning have been performed to solve planning problems using drones and robots technology in various application domains (with different mission specifications), as in [Cashmore et al., 2013; Cashmore et al., 2014; Steenstra, 2019; Torreño et al., 2017; Muñoz et al., 2016; Ulam et al., 2007; Kvarnström, 2011; Doherty et al., 2013]. In the following, we discuss the most important existing research works and connections related to our planning problem in this project, that could contribute to the development of this thesis and future work.

In similar fashion to our planning problem in this thesis, the core task for solving all the planning problems in the aforementioned previous works requires the formulation of the corresponding high-level planning model interpreted and encoded in PDDL. The resulting output plans, generated by a planner in a standard form, are *symbolic structures* (sequential or partial-order plans) constructed from the actions in the domain model (the first input) and instantiation with *symbolic values* from the problem description (the second input).

Previous work of AI Planning for underwater inspection tasks is carried out by *autonomous underwater vehicles* (AUVs) in [Cashmore et al., 2013; Cashmore et al., 2014] in order to inspect a collection of structures on the oilfield, a topic particularly interesting for the oil and gas industry. In this paper, authors describe their advanced approach of combining task planning in temporal domain with controllers in a feedback loop (success or failure). First, for defining an abstraction of the environment, a Probabilistic Roadmap (PRM) is generated using an OMPL *motion planner* in an environment model provided by PANDORA architecture [Lane et

al., 2012]. In particular, this motion planner takes into account the vehicle dynamics and checks for collision-free trajectories. Furthermore, this PRM ensures a extended connecting structure that links additional strategic waypoints, necessary for having sufficient view-points for each inspection point. Then, for generating a set of waypoints (an inspection trajectory) a *path planning* is also integrated. To solve their inspection problem, interpreted and encoded in PDDL, the planner Forward-Chaining Partial-Order Planning (POPF) [Coles et al., 2010] is used, of which the obtaining plan may contain waypoints where multiple "observe" and re-orienting actions take place. The implemented planning system is integrated in ROS to send the sequence of actions in the mission plan to the controllers. To test the behaviour of the planning system a simple scenario was simulated using UWSim, an open source tool for simulation and supervision of underwater intervention missions. Integrating such techniques of path planning and PRM into our work would be an interesting direction for future work.

In PANDORA, the interesting EU-funded research project presented in [Lane et al., 2012; Maurelli et al., 2016], the focus is to enhance the long-term (persistent) autonomy of AUV missions, through increased cognition, at all the levels of abstraction - Persistent Autonomy through learning, adaptation, observation and re-planning. To achieve smart inspection, an *ontological knowledge base* (KB) is incorporated, in order to cope with unexpected events and environments, to deal with faults and to make smart decisions in response to changes in the real-world. Robot Operating System (ROS) framework is used to integrate planning and all of the components of this project, and ROSPlan is used to execute the planned activities. Mission plans, to achieve a set of inspection and maintenance goals, are constructed using the planner POPF [Coles et al., 2010]. The results of this research project were demonstrated successfully in 3 different experimental scenarios: structure inspection, chain inspection and cleaning and valve turning.

Another interesting related work using autonomous underwater vehicles is explored in the Master thesis work in [Steenstra, 2019] which deals with the task of underwater surveying using AUVs. This bottom surveying technique is beneficial for detecting objects and mapping the seafloor which is fundamental in typical applications like Mine Countermeasure (MCM) and Search and Recovery (SAR) operations. To this end, the author builds a PDDL-based *task planning system* to control the AUV in performing *submarine survey missions* autonomously where the main objective is to plan a *coverage path* for such survey tasks while minimizing operation time. Hence, equipped with its *Side-Scan Sonar (SSS) system*, the AUV should navigate in a *lawnmower pattern* (Boustrophedon pattern) over the area of interest while scanning the entire area with its SSS system. This work takes into account underwater environment challenges for navigation and perception, namely "location uncertainty" and "limited communication". Specifically, to cope with these challenges the vehicle navigates using an *Inertial Navigation System (INS)* (IMU and gyroscope instead of GPS) to estimate its location, orientation and velocity, and in addition the acoustic communication is the usual choice for underwater activities

but these signals are often noisy, lossy and low-bandwidth in comparison to electromagnetic signals. It is worth recalling that the location uncertainty returns to the fact that GPS is not available underwater (because electromagnetic waves are heavily damped underwater) and consequently the vehicle has no access to its absolute position. Whereas limited communication is due to the high conductivity of seawater. Furthermore, as it is argued in [Steenstra, 2019], planning a *coverage path* is different from conventional path planning, as the focus is not on a destination but rather on the path itself, which is often referred to as *coverage planning*. Finally, the planning system is evaluated successfully by simulating the resulting survey plans in different scenarios using the ROS-based *UUV Simulator* [Manhães et al., 2016], a realistic simulation environment which provides the underwater world with different scenarios and seafloors. This is achieved through extending Gazebo (the open-source robotics simulation platform) with additional implemented plugins [2] in order to model actuators and sensors needed to simulate the interaction between the vehicle and the underwater world. In addition, the PDDL planning model is validated mathematically by means of the Event-B method.

In [Torreño et al., 2017] a comprehensive survey reviews the most relevant techniques in the research field of *Cooperative multi-agent planning (MAP)*, developed by the AI Planning and Multi-Agent Systems (MAS) communities. Authors discuss the advances in deterministic *cooperative and distributed MAP* methods that integrate planning and coordination. Additionally, they present the MAP solvers (planners) that participated in the 2015 Competition of Distributed and Multi-Agent Planning (CoDMAP) [3] and classify them according to their key features and relative performance. In general, solving such a MAP problem requires various features such as computational or information distribution, specialized agents, coordination or privacy. In fact, the need for *information distribution* stresses the issue of privacy, which is also one of the reasons to adopt a multi-agent applications. In order to emphasize the basic aspects that characterize a MAP task, this paper also introduces an illustrative modelling example, related to the transportation task in the logistics domain (adapted from [Torreño et al., 2014]), that would describe the key elements of modelling planning domains using the multi-agent version (MA-PDDL) of the specification language PDDL (the Planning Domain Definition Language), which is considered as the *de facto* standard language for modelling our single-agent planning problem as well. Yet, in this single-agent language, the user specifies the *domain* of the task (planning operators, types of objects, and functions) and the *problem* to be solved (objects of the task, initial state, and goals) [Torreño et al., 2014]. From a conceptual point of view, MAP approach generalizes the concept of a *single planning agent* treated in our thesis to solve the inspection planning problem, that is, executed by a single actuator. Whereas under the MAP paradigm multiple intelligent agents should work cooperatively to synthesize a *joint plan* that solves

[2] `https://github.com/uuvsimulator/uuv_simulator`
[3] `http://agents.fel.cvut.cz/codmap/`

the common goal of the group. Integrating such MAP techniques into our work, could contribute to interesting future development and research directions of extending our robotic inspection planning work in a broader perspective by involving more drones and robots and providing *agent-based system mechanisms* like, among others, information distribution, multiple agents coordination (in planning and execution) and privacy model.

Interestingly, there is also another important recent work in [Muñoz et al., 2016], that focus on mobile robots applications in two classical exploration domains: pictures acquisition and samples collecting/ delivering (after drilling operations). There, authors have developed and evaluated an AI planning system that combines the capabilities of high-level PDDL-based *task planning* and *path planning* for the control of a mobile robot in processing these exploration missions to be performed in experimental scenarios at specific waypoints in different locations. This new implemented planning system, called *up2ta* (Unified Path-Planning and Task-Planning Architecture), has been integrated using the well-known *Fast Forward (FF) planner* [Hoffmann and Nebel, 2001]. At the end, this up2ta system provides an ordered list of tasks, constituting the plan generated by the PDDL planner to achieve a set of goals , and the feasible paths between them obtained by the path planning algorithm.

Finally, a more extensive study in [Doherty et al., 2013], at the UASTech Lab [4] at Linköping University, focus on the problem of specifying, generating and executing the *high-level collaborative missions* deploying several autonomous Unmanned Air Vehicles (UAVs) or other unmanned systems in complex scenarios. To this end, authors propose to increase the degree of autonomy and the complexity of the collaborative systems by constructing a *conceptual framework and architecture* composed of a collection of functionalities and capabilities. In particular, their generic agent-based software architecture, based on the concept of *delegation*, integrates the use of the high-level mission specification language based on the use of *Temporal Action Logic (TAL)*, Task Specification Trees (TSTs), and an automated planner with a formal semantics based on the use of a temporal action logic (TAL) instead of PDDL. This planning system TFPOP [Kvarnström, 2011] combines interesting properties of *temporal partial-order* and *forward-chaining* planning , capable of generating both single- and multi-platform mission specifications in the high-level mission specification language that are translatable into executable TSTs. These TSTs, distributable and extensible structure for defining single- and multi-platform tasks and missions, are also required by the UAV platform deployed in our project during the achievement of mission goals. In this framework, missions can be defined in TAL or as TSTs and can be translated in either direction.

In all these previous works discussed above and in our project as well, we refer to a PDDL-model of a planning problem as a *domain description* and a connecting *problem description* (two distinct files), and eventually this is the input to a planner software (usually domain-independent AI planner), which aims to solve the given

[4] `www.ida.liu.se/divisions/aiics/`

planning problem via some appropriate AI planning algorithm. Mainly, given a domain description, we can express several *problem descriptions* (of different sizes, initial states and goals) connected to the same domain description. In fact, the plan creation aspect of AI planning deals mainly with *state transformation* problems. These problems are characterized by an initial state, a desired goal, and a set of possible actions that can be taken to change the state. For completeness, recalling another important aspect of plan creation using PDDL, it is also worth noting that, according to the slogan of the originator of PDDL Dew McDermott "Physics no advice", we need to formulate a *neutral specification* of the planning problems and focus on expressing the physical properties of the world without worrying to advise the AI planner on how to search for solutions.

Now for completeness, after exploring the research and literature discussed above, we briefly describe the primary aspects commonly reviewed in these previous research projects that we have made use of them in our thesis in order to solve the planning problem in our second work of this present thesis. Specifically, we investigate how we can model the planning problem and formulate the necessary actions in the inspection domain. We suppose that we have a world that consists of a team of a quadruped UGV (Spot) and a number of quadrotor UAV (DJI Matrice 100 drones, one and may be more). The goal is to navigate an industrial / construction site and performing inspection on predefined inspection points. In the PDDL-model we consider a problem description of small scenario with 5 inspection points. One possible *high-level* task plan is obtained using an off-the-shelf *generic solver* (task planner) based on suitable AI planning Algorithms able to be run on WARA-PS framework or on various academic research sites available online. This high-level solution plan can be viewed as a *mission specification* consisting of a *sequence of composite (abstract) actions*, like "spot-inspect" or "uav-inspect". Furthermore, the generated "high-level" inspection plan, expressed in the high-level domain-independent language PDDL, is then translated (expanded) into executable *Task Specification Tree* (TST), created through a TST Factory in the aforementioned WARA-PS framework. The resulting TST uses expanded elementary action nodes corresponding to *detailed elementary actions*, like "move-to" and "fly-to". In general, tasks are assigned based on criteria such as robot type, task duration and reachability, and battery level.

1.3 Motivating Example: Visual inspection scenario

In future work, field experimental tests on the real robotic platforms may be conducted to evaluate the simulation results of this project work in real-world experiments by executing certain inspection missions in a small practical scenario, executed by Spot together with a quadrotor UAV (DJI Matrix 100).

For instance, an examplar case of such an *industrial inspection scenario* consists of visual inspection tasks in several different points inside a building construction

environment (or industrial sites, generally speaking). These inspections would be executed by taking pictures and/or video live-streaming using the available cameras and sensors on-board. In case Spot robot is incapable to reach some difficult-to-access inspection points, the DJI M100 will intervene and accomplish such unreachable tasks having the ability of flying and "seeing" using remote sensing not only in horizontal plane but also in altitude.

In particular, such these inspection missions are useful for construction companies, by which they can perform revision / inspection of the specifications required of newly construction works, at some predefined inspection points, and verify their correctness in accordance with the *construction drawings* (Blueprints), which usually indicate arrangement of building components, detailing, dimensions and so on.

1.4 Contribution and Development

In general, this degree project contributes to benefit the transition from Industrial Automation to Industrial Autonomy. In particular, it will contribute to the development and evolution of the fields of *collaborative robot systems* and AI Automated Planning in industrial inspections and other applications including environmental monitoring and assisting emergency services (in the domain of public safety) in scenarios such as earthquakes, flooding or forest fires. In Figure 1.3, we can see several illustrative examples of industrial inspections from Yokogawa Robot and Drone Technology [5], which supports *plant control* under the transition process from Industrial Automation to Industrial Autonomy (IA2IA).

According to study conducted by Boston Dynamics, *automated robotic inspections* would improve safety, efficiency and predictability in industrial scenario, in which the benefits can for example include removing personnel from dangerous environments (electrified, radioactive, explosive), collecting data remotely and identifying risks from a far, as well as reducing time to inspection.

[5] `https://www.yokogawa.com/solutions/featured-topics/digital-transformation/robot-and-drone-technology/#Overview`

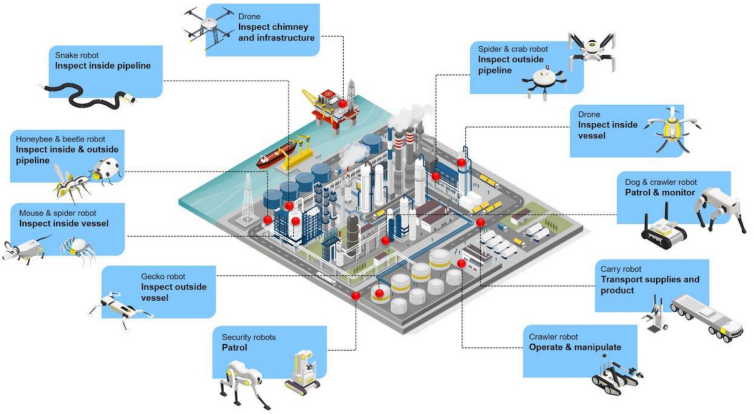


Figure 1.3 An overview of different collaborative robot systems necessary to carry out important inspections and monitoring tasks in industry. (Courtesy: Yokogawa Automation)

1.5 Thesis Organization

The thesis is organized as in the following outline:

Chapter 2: Preliminaries

In this chapter 2 we start the thesis with a short summary of the underlying theory and background. The chapter introduces the basic theoretic concepts that we rely on throughout this work. First, we illustrate the basic concept of *layered architecture* and its application on multi-rotor system using hierarchical control approach, with a particular focus on *quadrotor* control and its principal movements. Then, it describes the main notions of *AI planning* (task planning) and its applications, as well as the related AI algorithms and modelling languages.

Chapter 3: WARA-PS Core System

This chapter 3 introduces/describes the basic components of WASP Research Arena Pubclic Safety (WARA-PS) framework. It covers how to interact with WARA-PS simulator and setting up the main environments necessary for execution and development of UAV flight missions on our computer system. Then, the chapter illustrates the structure of DJI SDK API and the WARA-PS DJI simulator upon which we build our work to control the flight operations of our quadrotor UAV (DJI Matrice 100) deployed in this project. Moreover, we include an example illustrating the ROS communication underlying one DJI simulation using WARA-PS simulator configured with a trajectory tracking controller using MPC.

Chapter 4: Rendezvous Problem: Simulation and Control using ROS

In chapter 4 we present the simulated Rendezvous landing system and its components. Its core part is how to use ROS interface together with the PID-based *position-tracking controller* to build the proposed Rendezvous algorithm in simulated environment using the ROS-based WARA-PS simulator (djisim). In addition, it describes how to use *Gazebo plugin control* to model a two-wheeled mobile robot to simulate the behaviour of the ground robot. Finally, the chapter reports evaluation results in simulated experiments using ROS bags to plot the convergence graph and illustrate the satisfactory performance of the algorithm.

Chapter 5: Physical Modelling of the Robotic Platforms: Spot and DJI Matrice 100

This chapter 5 covers modelling of the quadruped and quadrotor systems that are deployed in this project. In particular, it describes the practical approximation techniques used to model the quadruped Spot process as a *first-order system* when we cannot follow the classic Newton-Euler methods to extract the kino-dynamic model. Then, it discusses how the nonlinear dynamics of the quadrotor UAV is represented by the common ordinary differential equations (ODE) . A brief overview of the main characteristics and functionalities of these two deployed platforms, i.e., the legged robot Spot and the quadrotor DJI M100 is given in section 9.1 and section 9.2 in Appendix.

Chapter 6: Rendezvous Landing using MPC Design: Theoretical Development for Future Work

In chapter 6, we present the theoretical development of solving Rendezvous landing problem using a second approach to highlight and motivate the use of MPC framework on this problem, previously solved using PID approach. The aim is to extend the building of the initial PID-based algorithm presented previously in section 4.6 to the case of model- and optimization-based control approach MPC to reinforce the performance of the algorithm also in real-world experiments.

First this chapter introduces the concept of receding horizon control (RHC). Then, it describes the relevant *optimal control problem* (OCP) formulation related to our Rendezvous landing algorithm based on Model Predictive Control (MPC).

Finally, three advanced MPC-based methods are proposed to increase robustness on this Rendezvous performance. The intention would be to implement them and conduct experimental evaluation on real robotic platforms in future work.

Chapter 7: AI Planning Problem - Automated Robotic Inspections

In this chapter 7, we provide a detailed description of solving the AI automated planning problem in the second part of the project. The goal of this contribution

is to supply mobile robots with *higher autonomy* when operating in in the field of industrial inspection.

First the chapter describes how to model *autonomous planning capabilities* in the robotic inspection scenario using Planning Domain Definition Language (PDDL) to plan efficiently critical inspection tasks to be performed by a team of cooperative mobile robots, while respecting their own *sensor limitations* and *motion constraints*.

Next, it present the encoding of our *planning model* formulated in two files, (1) the inspection *domain* and (2) the *problem instance*, implemented in the high-level mission specification language PDDL. This problem file includes the actual planning problem that needs to be solved by the PDDL-solver to obtain a candidate solution plan at the end.

Finally, the chapter introduces the contribution of a variety of the off-the-shelf planners applying various AI algorithms. It also presents a deeper insight in the concept of *planner selection*.

Chapter 8: Summary: Conclusions and Future Work

This final chapter 8 summarizes the thesis by recalling its main contributions and achievements. It then criticizes the thesis work current limitations to point out towards new future work directions. Finally, it discusses the benefits of Robotic Industrial Inspections in the field of Industry Digitalization (Industry 4.0).

Chapter 9: Appendix

This chapter 9 includes the technical information about the robotic plaforms Spot and DJIM100, as well as, the PDDL domain and problem files necessary for solving the AI task planning problem in the automated robotic inspection domain. Moreover, it contains useful examples about implementing nonlinear MPC using Python and ACADO.

2

Preliminaries: Background and Related work

In this section, we recall the background and basic notions that contributes to our thesis from different literature and research studies in Automatic Control, Robotics and Artificial Intelligence (AI). In particular we review the common hierarchical control approach for quadrotors, and additionally we get a general feeling for what AI planning is and how to specify planning problems.

2.1 Layered Architecture - Hierarchical Control Approach

For completeness, we briefly outline the basic concepts of the modern "layered decomposition" of control systems, drawn and reviewed from the literature in [Åström and Murray, 2020]. This layered control systems are common in a large class of control problems which consist of *planning and following a trajectory* in the presence of noise and uncertainty, including robotics, self-driving cars, and flight control. Usually, the specific "abstraction layers" in a control architecture depend on the problem domain, however, here we are going to deal with the decomposition layers that are common in controlling the motion of many mobile robot systems, including the quadruped Spot robot and the "vertical take-off and landing" (VTOL) quadrotor drone deployed in this thesis project.

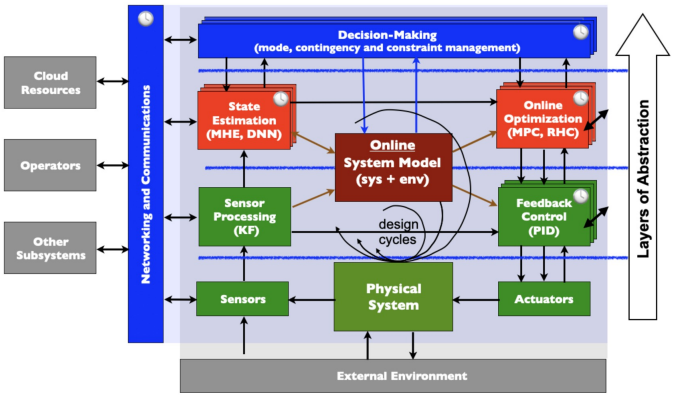


Figure 2.1 Layered control systems. The system consists of four layers: the physical layer (lowest), the feedback regulation layer (green), the trajectory generation layer (red), and the decision-making layer (blue). Networking and communications allows information to be transferred between the layers as well as with outside resources (left). The entire system interacts with the external environment, represented at the bottom of the figure [Murray, 2022].

In general, for complex control systems, it is often useful to break down the control problem into a hierarchy of control problems, each solved at a different layer of abstraction [Åström and Murray, 2020]. The resulting layered control system consists of four layers, as illustrated in Figure 2.1:

1. *Physical layer* (lowest), representing the physical process being control as well as the sensors and actuators. This layer is often described in terms of "input/output dynamics" that can describe how the system evolves over time, where an "ordinary differential equation" model is one of the most common representations, as shown in Figure 5.5 that corresponds to the quadrotor model in this work.

2. *Feedback regulation layer* (green), sometimes also called the "inner loop", in which we use feedback control to track a reference trajectory. This layer commonly represents the abstractions used in classical control theory. This layer could also use Kalman filtering to process signals (to try to minimize the effects of noise) and also perform "sensor fusion".

3. *Trajectory generation layer* (red), sometimes also called the "outer loop". In this layer we attempt to find trajectories for the system that satisfy a commanded task taking in account nonlinearities and constraints on the inputs and states. we assume that the effects of noise, disturbances, and unmodeled dynamics have been taken care of at lower levels. However, this layer can also have an "observer" function (state estimation), as in the feedback regulation

layer, that can be used for "perception" and "prediction" tasks, to identify other agents in the environment and their predicted trajectory for example. This information will be used by the trajectory generation algorithm to avoid collisions or to maintain the proper position relative to those other agents. This could correspond to our problem of tracking Spot robot by the UAV quadrotor in the landing task.

4. *Decision-making layer* (blue), sometimes also called the "supervisory" layer. This supervisory control module performs "higher-level tasks" such as mission planning and contingency management (if a sensor or actuator fails). At this layer we often reason over "discrete events" and "logical relations". For example, we may care about "discrete modes of behavior", which could correspond to different phases of operation (takeoff, cruise, landing) or different environment assumptions (highway driving, city streets, parking lot). This layer could correspond to our AI planning problem.

Another important element of modern control systems is their "distributed" and "interconnected" nature. Networking and communications allows information to be transferred between the layers as well as with outside resources (left). The entire system interacts with the external environment, represented as the block at the bottom of the diagram in figure [Murray, 2022]. This block represents many things, including noise, disturbances, unmodeled dynamics of the process, and the dynamics of other systems with which our system is interacting.

2.2 Hierarchical Control Approach for Quadrotors

The important observations reviewed above in the previous section, explained in detail in [Åström and Murray, 2020] and [Murray, 2022], would be useful to figure out the hierarchical control approach we choose in this project to solve our VTOL quadrotor autonomous landing problem, where we reason about the control system at different layers of abstraction, which is itself a "nested" set of control systems. We follow this "cascaded" control structure based on several articles work developed at ETH Zurich and well described in [Bloesch et al., 2010], [Kamel et al., 2017b] and [Kamel et al., 2017a], which is also very similar to the general control structure used in WARA-PS framework.

In fact, a hierarchical control approach is common for quadrotors according to many articles. As illustrated in Figure 2.2, found in [Mahony et al., 2012], the lowest level, the highest bandwidth, is in control of the rotor rotational speed. The next level is in control of vehicle attitude, and the top level is in control of position along a trajectory .

As we can notice in this figure, these levels form nested feedback loops and make use of an "inner/outer" loop desing methodology. Similar block diagrams related to our landing task in this project, with the same "cascade" fashion, will be dis-

cussed in the following sections and illustrated in Figures 6.4 and 6.6. These block diagrams shows the process dynamics and controller divided into two components: an inner loop consisting of the attitude dynamics and controller and an outer loop consisting of the position controller using Model Predictive Control (MPC).

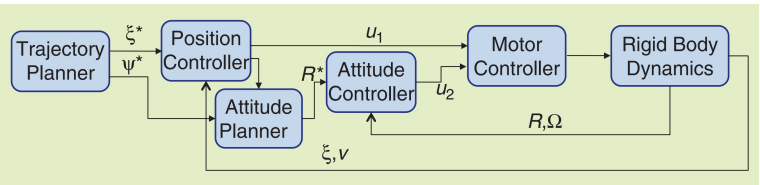


Figure 2.2 A common hierarchical control approach for VTOL quadrotors: The innermost motor control loop, the intermediate attitude control loop, and the outer position control loop. These levels form nested feedback loops. **Rotor speed** drives the dynamic model of the vehicle, so high-quality control of the motor speed is fundamentally important for overall control of the vehicle; "high bandwidth control" of the thrust T_Σ, denoted by u_1, and the torques (τ_x, τ_y, τ_z) denoted by u_2, lead to high performance attitude and position control [Mahony et al., 2012].

2.3 Quadcopter Principal Movements and Characteristics

In this section, it is worthwhile to outline the principal characteristics that could affect the quadrotor control system and, furthermore, help readers to conceptualize and figure out the working principals of the quadcoptor used in this project.

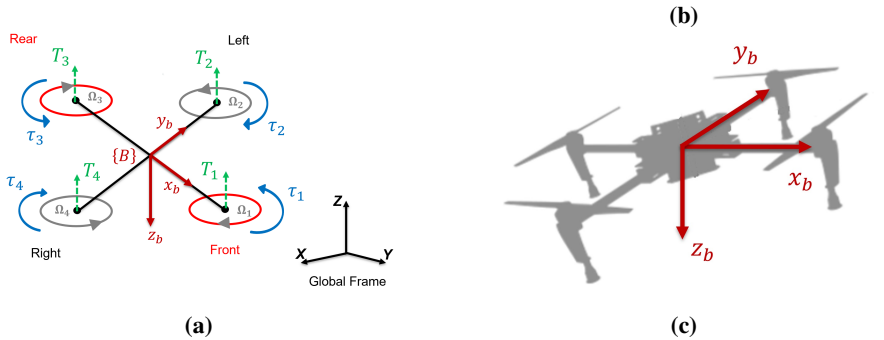


Figure 2.3 The inertial and the body coordinate frames of the quadcopter: The most common multirotor aerial platform, the quadrotor vehicle, consists of four individual rotors attached to a rigid cross airframe. (a) Variables and notation for the multirotor aerial vehicle model. Quadrotor uses 4 rotors for lift and propulsion. As shown in figure, rotor i rotates anticlockwise (positive about the z axis) if i is even and clockwise if i is odd. Yaw control is obtained by adjusting the average speed of the clockwise and anticlockwise rotating rotors. (b) Illustrative picture depicting the quadcopter.

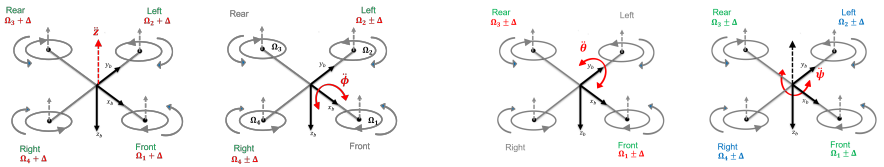
The quadrotor vehicle, the most common multirotor aerial platform in robotics research and industry worldwide, consists of four individual rotors attached to a rigid cross airframe: Rotor 1 is on the positive x_B-axis, 2 on the positive y_B-axis, 3 on the negative x_B-axis, 4 on the negative y_B-axis, as sketched in [Mellinger et al., 2010], [Mahony et al., 2012] and [Powers et al., 2015]. The main properties of these vehicles can be summarized by the following points:

- Because of four inputs and six outputs in a quadrotor, the quadcopter has six degrees of freedom (6 DoF), six outputs $\mathbf{y} = \begin{pmatrix} x & y & z & \phi & \theta & \psi \end{pmatrix}$ but there are only four control inputs $\mathbf{u} = (T_\Sigma, \tau)$. Such quadrotor is considered an *underactuated nonlinear complex system*, i.e., there are two degree of freedom that are left uncontrollable. Quadrotor motion is obviously caused by a series of forces and moments coming from different physical effects.

- Depending on the speed rotation of each propeller (four motors) it is possible to identify the *four basic movements* of the quadrotor: thrust, roll, pitch and yaw, denoted respectively by T, ϕ, θ and ψ, as illustrated in Figure 2.4. As such, attitude and position of the vehicle can be controlled to desired values by changing the speeds of the four motors.

- *Attitude control* is the heart of the control system, it keeps the 3D orientation of the helicopter to the desired value [Bouabdallah and Siegwart, 2007]. Its performance is directly linked to the performance of the actuators.

- A position error results in a required *translational velocity*. To achieve this translational velocity it requires appropriate pitch and roll angles so that a component of the vehicle's thrust acts in the horizontal plane and generates a force to accelerate the vehicle. As it approaches its goal the airframe must be rotated in the opposite direction so that a component of thrust decelerates the motion.
- To achieve the pitch and roll angles requires *differential propeller thrust* to create a moment that rotationally accelerates the airframe.



(a) **(b)**

Figure 2.4 Characteristics of the quadrotor movement: Depending on the speed rotation of each propeller it is possible to identify the four basic movements of the quadrotor. In hovering, all the propellers rotate in the same speed Ω to counterbalance the gravity acceleration g. (a) **Throttle input** (left): This command is provided by increasing (or decreasing) all the propeller speeds by the same amount. It leads to a vertical force w.r.t. body-fixed frame B which raises or lowers the quadrotor. **Roll movement** ϕ (right): obtained by increasing (reducing) the speed of the left motor while reducing (increasing) the speed of the right motor. (b) **Pitch movement** θ (left): obtained by increasing (reducing) the speed of the rear motor while reducing (increasing) the speed of the front motor. **Yaw movement** ψ (right): obtained by increasing (decreasing) the speed of the front and rear motors while decreasing (increasing) the speed of the lateral motors (left-right couple).

2.4 AI Planning Framework: Functionalities, Main Features, Algorithms

In this section we outline the conceptual approach to AI Planning and the related main features and algorithms, as well as the common planning description language PDDL and other representation techniques to represent and encode the task-related information defined by the mission specification for the planning problem.

A good question can be emerged here: *What is AI Planning?* It is also known as Task Planning. This clarification is important for not confusing it with the concepts of Path Planning and Motion Planning in Robotics. Primarily, we argue that "Planning and Problem Solving" is one of the main areas in Artificial Intelligence (AI). In fact, *"Intelligence" is associated with the ability to "plan actions" needed to accomplish a desired goal and solve problems with those plans* [Russell and Norvig, 2010; Poole and Mackworth, 2017; Murphy, 2019].

Moreover, Planning is a core *deliberation function* for an autonomous robot. It can be viewed as the coupling of a *prediction problem* and a *search problem*. The former refers to the capability to predict the effects of an action in some context. The latter searches through all possibilities in order to choose *feasible actions* and to organize them into a "plan", which is foreseen, at planning time, to be feasible and to achieve a "desired goal" or optimize a performance criterion [Ingrand and Ghallab, 2014].

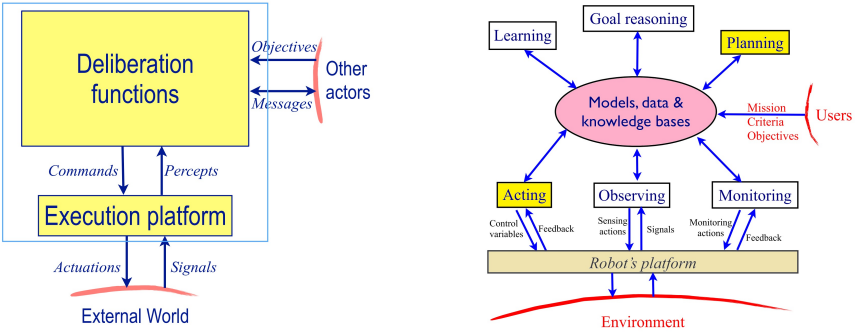


Figure 2.5 Deliberation functions implement the "reasoning" needed to choose, organize and perform actions that achieve the actor's objectives, to react adequately to changes in the environment, and to interact with other actors, including human operators. (a) Conceptual view of an actor [Ghallab et al., 2016a]. the actor has two main modules: a set of deliberation functions and an execution platform. (b) Schematic view of deliberation functions: interactions between functions is oversimplified. [Ingrand and Ghallab, 2014]. it is conceptually clarifying to distinguish between the following six deliberation functions: Planning, Acting, Observing, Monitoring, Goal Reasoning and Learning.

Deliberation Functions As discussed in [Ghallab et al., 2016a; Ingrand and Ghallab, 2014], there is more to "deliberate action" than just planning in an abstract space. Several distinct functions can be required for *acting deliberately*, of which the most relevant can be listed in the following six deliberation functions, as shown in Figure 2.5: Planning, Acting, Observing, Monitoring, Goal Reasoning and Learning.

Deliberation is a very active and critical research field in *intelligent robotics*. It covers a wide spectrum of problems and techniques at the intersection of Robotics and Artificial Intelligence. Deliberation has to closely interact with sensing and actuating. It does not stop at planning and is not reducible to a single function. It critically requires the coherent integration of several deliberation functions.

To highlight the difference between planning and acting, it is useful recalling, as stated in [Ingrand and Ghallab, 2014], that in planning we combine "prediction" and

"search" *to synthesize a trajectory in some abstract action space* based on predictive models of the environment and feasible actions in order to achieve some purpose. Whereas acting refines planned actions into "commands" appropriate for the current context and reacts to events. Moreover, the techniques used in planning and acting can be compared as follows. Planning can be organized as an *open-loop search*, whereas acting needs to be a *closed-loop process*. Planning relies on *descriptive* models (know-what); acting uses mostly *operational* models (know-how).

For the rest of deliberation functions the reader is invited to review the literature in [Ingrand and Ghallab, 2014; Ghallab et al., 2016a]. "Automated Planning" textbook of [Ghallab et al., 2016a] would be excellent on all aspects of planning and deliberation.

NOTE *Motion planning versus task planning:* In motion planning in Robotics, robots have to plan precise movements of their wheels, legs, arms, and joints. Motion planning and task planning are different problems that use distinct representations and search techniques. The latter has been mostly considered by the *AI community*, while the former has been studied by the robotics and *computational geometry* communities. In simple cases one can decouple the two problems: task planning produces a "high level plan" whose steps requiring motions can be handled by a specific motion planner [Ingrand and Ghallab, 2014]. LaValle's textbook [LaValle, 2006] "Planning Algorithms" (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

Applications in Real World: Automation versus Autonomy

Why AI planning/Task Planning is needed? When dealing with *autonomous robots* such as spacecrafts, autonomous underwater vehicles (AUV), unmanned aerial vehicles (UAV) or self-driving cars, it is important to consider and integrate several *deliberative functionalities* for use in the deployed systems. From the perspective of robotics, deliberation is any "computational function" required to act deliberately, as mentioned in [Ingrand and Ghallab, 2014].

With AI planning we enable "automated reasoning" capability in the nowadays robots, which usually rely on "handcrafted" knowledge modelled by continuous human guidance, i.e., an operator. Using deliberative functionalities, each robotic platform becomes an agent, and perhaps "Agentifying" could be a good term to use in this context, as used in [Doherty et al., 2013]. As such, using *sensory-motor capabilities*, these robotized systems can interact with their environment and may have to *act deliberately* in order to achieve their intended objectives.

Having said that, it is important to highlight that autonomous robots may or may not require explicit deliberation. Deliberation is not needed for autonomous robots working in fixed, well-modelled environments, as for most fixed manipulators in manufacturing applications. Neither would deliberation be required if all feasible missions consist of repeating a single task, e.g., as for a vacuum cleaning

or a lawn-mowing robot. In these and similar cases, deliberation takes place at the "design stage". Its results are directly implemented in the robot controller [Ingrand and Ghallab, 2014]. In short, we can argure that "autonomy plus diversity" entail the need for deliberation or AI planning.

In general, Robotics is manifesting a paradigm shift, from "automated tools" to "autonomous agents", i.e. to intelligent robotics or AI robotics. It is specifically shifting from a "closed world" of highly repetitive actions to an "open" real world" where the environment and tasks are uncertain, partially observable or even unknown. AI robotics has concentrated on how a mobile robot should handle unpredictable events in an unstructured world [Murphy, 2019]. For industrial robots as well, production is currently experiencing a paradigm shift from "mass production" to "mass customization" of products, as stated in [Pedersen et al., 2015]

Artificial agents that are able to "autonomously decide" *what to do*, *when and how to do it*, is one of the most fascinating and appealing challenges, that humanity is facing in this century. Researchers in the field of Robotics and Artificial Intelligence (AI) are pursuing such an ultimate goal by analysing every aspect of human psychology, locomotion and cognitive capabilities in order to develop agents that are able to process information and reach human-level performance [Riccio, 2018].

Model-Based Representation of AI Planning Problems

In this section, we give a short review of the principal aspects that characterize the model-based representation of AI planning problems that is necessary for encoding a particular domain into AI planning systems, i.e., AI planners. Specifically, in the model of these problems, agents or Robots need to have representations of their world and, importantly, of the actions they can perform to affect the world, as well as description of their objectives to achieve. The resulting problem specification, "model description", will be the input of the planner software, a "generic solver", which aims to solve the given planning problem [Geffner and Bonet, 2016].

Figure 2.6 Planner is generic solver for instances of a particular model: A planner is a solver over a class of models (classical, conformant, contingent, MDP, POMDP); it takes a model description, described in compact form by means of "planning languages", and computes the corresponding controller. Models and Solvers: Research work in AI has increasingly focused on the formulation and development of solvers for a wide range of models. A solver takes the representation of a model instance as input, and automatically computes its solution in the output [Geffner and Bonet, 2016].

First, we argue that the concept of planning conceived as *the model-based approach to action selection*, as stated in [Geffner and Bonet, 2016], is an important

view that defines more clearly the role of planning in intelligent autonomous systems. This in addition to the common definition, related to the branch of AI, concerned with the "synthesis of plans of action to achieve goals".

Next, for completeness, we recall that the problem of "selecting the action to do next" is at the center of *Intelligent Autonomous Behaviour*. In Artificial Intelligence (AI), three different approaches have been used to address this problem [Geffner and Bonet, 2016]

- *Programming-based approach* - Specify control by hand: the controller that prescribes the action to do next is given by the programmer, usually in a suitable high-level language.

- *Learning-based approach* - Learn control from experience: the controller is not given by a programmer but is induced from experience as in reinforcement learning .

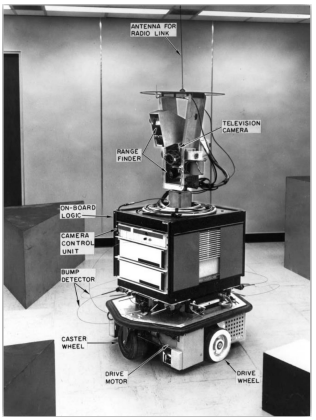
- *Model-based approach (Planning)* - Derive control from model: the controller is not learned from experience but is derived automatically from a "model" representing the initial situation, the actions, the sensors, and the goals.

AI Planning Framework: Historical Background According to historical notes in [Russell and Norvig, 2010], AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. The Stanford Research Institute Problem Solver or STRIPS, the first major planning system (automated planner) developed by Richard Fikes and Nils Nilsson in 1971 at SRI International[1] [Fikes and Nilsson, 1971], illustrates the interaction of these influences. STRIPS was designed as the planning component of the software for the Shakey robot project [2] at Stanford Research Institute (SRI). Shakey the robot is shown in Figure 2.7a and some of the important characteristics of Shakey and its planner STRIPS are summarized in the same figure caption.

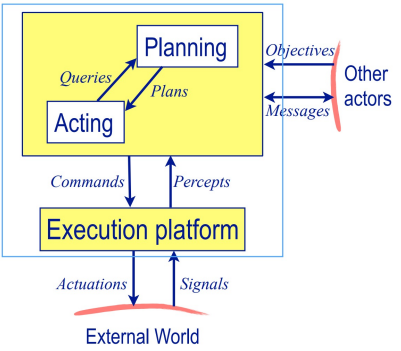
The *representation language* used by the planner STRIPS has been far more influential than its algorithmic approach; the same name is also used to refer to the *formal language* of the inputs to this planner. This language is the base for most of the languages for expressing automated planning problem instances in use today; what we call the conventional "classical planning" language is close to what STRIPS used. The *Action Description Language*, or ADL [Pednault, 1986; Pednault, 1988], relaxed some of the STRIPS restrictions and made it possible to encode more realistic problems.

[1] https://en.wikipedia.org/wiki/Stanford_Research_Institute_Problem_Solver
[2] https://en.wikipedia.org/wiki/Shakey_the_robot

The *Problem Domain Description Language*, or PDDL [Ghallab et al., 1998], was introduced as a computer-parsable, standardized syntax for representing planning problems and has been used as the de facto standard language for the *International Planning Competition* (IPC) since 1998. PDDL was derived from the original STRIPS planning language, which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals. So far, there have been several extensions to PDDL.



(a)



(b)

Figure 2.7 Automated Planning (a) **Shakey the Robot** (1966 - 1972) was the first general-purpose mobile robot able to reason about its own actions. While other robots would have to be instructed on each individual step of completing a larger task, Shakey could analyze commands and break them down into basic chunks by itself. **STRIPS planner**: Shakey had a short list of available actions within its planner, STRIPS. These actions involved travelling from one location to another, turning the light switches on and off, opening and closing the doors, climbing up and down from rigid objects, and pushing movable objects around. The STRIPS automated planner could devise a plan to enact all the available actions, even though Shakey himself did not have the capability to execute all the actions within the plan personally. (b) Conceptual view of a computational agent or an actor, as shown in [Ghallab et al., 2016a]: **Automated Planning** is the field devoted to studying the "reasoning" side of acting. The actor has two main modules: a set of deliberation functions and an execution platform. Deliberation functions implement the "reasoning" needed to choose, organize, and perform actions that achieve the actor's objectives, to react adequately to changes in the environment, and to interact with other actors, including human operator. This module is restricted to "planning" (which actions) and "acting" (how to perform them).

Virtually all work in automated planning has been concerned with ways of representing and solving what might be called the *classical planning problems*. The so-called "classical planning problem" is normally considered the simplest form of planning problem, the restricted conceptual model, which assumes a determinis-

tic, discrete and essentially non-temporal world model [Haslum et al., 2019]. This framework has typically been used to model real-world problems and is usually composed of the following three components, as stated in [Geffner and Bonet, 2016]

1. the *models* that express the dynamics, feedback and goals of the agent. From a more practical perspective, planning brings together AI *knowledge representation* and AI *problem solving* techniques. To apply a particular AI method to a problem, we must formulate a "problem representation" [Haslum et al., 2019].

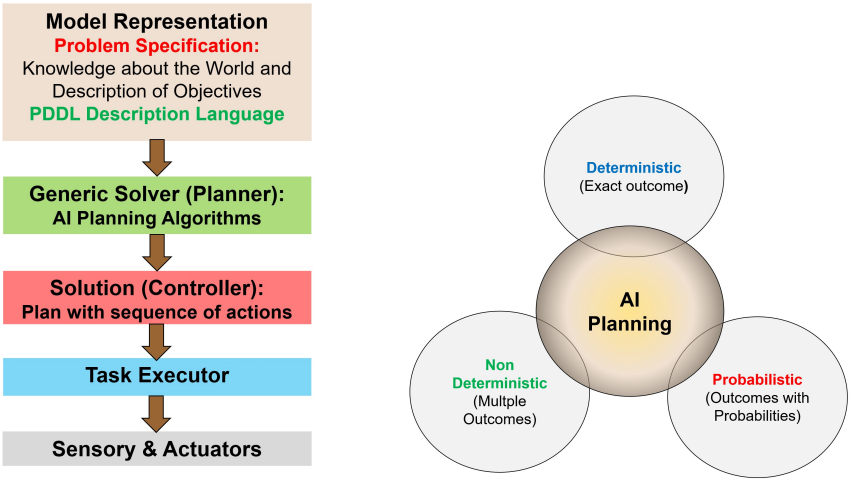
2. the *languages* that express these models in compact form. The general language features for describing states and actions can be summarized as follows [Russell and Norvig, 2010]

    - *Representation of states*: Decompose the world in *logical conditions* and represent a state as a *conjunction* of positive literals (where a "literal" is an atomic sentence which consists of a single proposition symbol that can be true or false).

    - *Representation of goals*: Partially specified state and represented as a *conjunction of positive ground literals* (where "ground" is a term with no variables). A goal is satisfied if the state contains all literals in goal.

    - *Representation of actions*: Action = PRECOND + EFFECT

3. the *algorithms or solvers* that use the representation of the models for generating the solution plan/behaviour. Planning systems (planners) are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions. These representations make possible the derivation of effective "heuristics" and the development of powerful and flexible algorithms for solving problems [Russell and Norvig, 2010].



(a) **(b)**

Figure 2.8 Making plans is considered a sign of intelligence, and for this reason *automated planning* has been one of the goals of research in Artificial Intelligence (AI) since its beginning [Haslum et al., 2019]. (a) **AI Planning basic structure**: An oversimplified planning process primarily consists of a model representing the world and objectives to achieve, a general solver/planner with AI algorithms, a solution plan with actions to perform and an executor. (b) **Planning in Real-World**. There is a variety of models which is the result of several orthogonal dimensions: uncertainty in the initial system state (fully known or not), uncertainty in the system dynamics (deterministic or not), the type of feedback (full, partial or no state feedback), and whether uncertainty is represented by sets of states or probability distributions. Thus, we distinguish between Standard Classical Planning (Exact outcome), Non-Deterministic Environment (Multiple outcomes), and Probabilistic Environment (Outcomes with probabilities).

Conceptually, as shown in the first block, in the oversimplified planning structure in Figure 2.8a, planning problems are defined and represented before solving them using some algorithmic approaches. It is usually possible to specify planning problems using various representation techniques and description languages.

Since the 90s, increasing attention has been placed on planning over "non-classical models", the extended models, such as MDPs and POMDPs where *action effects are not fully predictable*, and the *state of the system is fully or partially observable*. An overview about these broader classes of planning problems can be summarized in the table in Figure 2.9. More about these extended models can be explored in the literature.

Defining Planning Models - Basic State Model For better understanding the standard classical planning and the extended models we briefly review from

[Geffner and Bonet, 2016] the mathematical definition and the main characteristics of the essential models used in planning.

As stated previously, classical planning is concerned with the selection of actions in environments that are *deterministic* and whose initial state is *fully known*. In this type of problem, the "*world*" is regarded as being in one of a potentially infinite number of states. Performing an action causes the world to jump from one state to the next [Pednault, 1988].

Indeed, a wide range of models used in planning can be understood as variations of a *basic state model* featuring $\mathscr{S} = \langle S, s_0, S_G, A, f, c \rangle$. In this way, the model underlying classical planning can be described as this state model $\mathscr{S} = \langle S, s_0, S_G, A, f, c \rangle$, in which each components has the following meaning, where it is normally assumed that action costs $c(a,s)$ do not depend on the state, and hence $c(a,s) = c(a)$:

1. a finite and discrete state space S
2. a *known initial state* $s_0 \in S$
3. a non-empty set $S_G \subseteq S$ of goal states
4. actions $A(s) \subseteq A$ applicable in each state $s \in S$
5. a *deterministic* state transition function $f(a,s)$ such that $s' = f(a,s)$ stands for the state resulting of applying action a in s, $a \in A(s)$
6. positive *action costs* $c(a,s)$

In the *problem specification* of classical planning, we are given a set of acceptable goals, a set of allowable actions, and a description of the world's initial state. We are then asked to find a "sequence of actions" $\pi = a_0, \ldots, a_{n-1}$ that will transform the world from any state satisfying the initial-state description to one that satisfies the goal description, generating a state sequence $s_0, \ldots, s_n$ such that $a_i \in A(s_i)$, $s_{i+1} = f(a_i, s_i)$ and $s_n \in S_G$, for $i = 0, \ldots, n-1$. This framework has typically been used to model real-world problems [Pednault, 1988].

As an input, **classical planners** accept a *compact description* of models of this form in *languages featuring "variables"* (Strips, PDDL, PPDDL, ...), where the **states** are the possible "valuations" of the variables. The resulting classical solution plan $\pi = a_0, \ldots, a_{n-1}$ represents an *open-loop controller* where the action to be done at time step i depends just on the step index i

However, the solution of models that accommodate **uncertainty and feedback**, produce *closed-loop controllers* where the action to be done at step i depends on the "*actions and observations*" collected up to that point. These models can be obtained by "relaxing the assumptions" in the model above displayed in italics. The model for **partially observable planning**, also called *planning with sensing* or *contingent planning*, is a variation of the classical model that features both uncertainty and feedback, namely uncertainty about the initial and next possible state, and partial

information about the current state of the system. Mathematically such a model can be expressed in terms of the following ingredients:

1. a finite and discrete state space S
2. a *non-empty set* S_0 of *possible* initial states, $S_0 \subseteq S$
3. a non-empty set $S_G \subseteq S$ of goal states
4. a set of actions $A(s) \subseteq A$ applicable in each state $s \in S$
5. a *non-deterministic* state transition function $F(a,s)$ for $s \in S$ and $a \in A(s)$, where $F(a,s)$ is non-empty and $s'' \in F(a,s)$ stands for the possible successor states of state s after action a is done, $a \in A(s)$
6. a set of observation token O
7. a sensor model $O(s,a) \subseteq O$, where $o \in O(s,a)$ means that token o may be observed in the (possibly hidden) state s if a was the last action done
8. positive *action costs* $c(a,s)$

Similarly, a **partially observable planner** is a program that accepts as inputs compact descriptions of instances of the model above and automatically outputs the control.

The models above are said to be **logical** as they only encode and keep track of "what is possible" or not. In **probabilistic models**, on the other hand, each possibility is weighted by a probability measure. A probabilistic version of the partially observable model above can be obtained by replacing "the sets" S_0, $F(a,s)$ and $O(s,a)$ by "probability distributions", i.e., the set of possible initial states S_0, the set of possible successor states $F(a,s)$ and the set of possible observation tokens $O(s,a)$, as follows:

- a *prior probability* $P(s)$ on the states $s \in S_0$ that are initially possible
- *transition probabilities* $P_a(s'|s)$ for encoding the likelihood that s' is the state that follows s after a
- *observation probabilities* $P_a(o|s)$ for encoding the likelihood that o is the token that results in the state s when a is the last action done

The model that results from changing the sets S_0, $F(a,s)$ and $O(s,a)$ in the partially observable model, by the probability distributions $P(s)$, $P_a(s'|s)$ and $P_a(o|s)$ is known as a *Partially Observable Markov Decision Process* or **POMDP** [Kaelbling et al., 1998]. The advantages of representing uncertainty by probabilities rather than sets is that one can then talk about the "*expected cost*" of a solution as opposed to the cost of the solution in the *worst case*.

A **fully observable model** is a partially observable model where the state of the system is fully observable, i.e., where $O = S$ and $O(s,a) = \{s\}$. In the logical setting such models are known as *Fully Observable Non-Deterministic* models, abbreviated **FOND**. In the probabilistic setting, they are known as *Fully Observable Markov Decision Processes* or **MDP**s [Bertsekas, 1995].

Finally, an **unobservable model** is a partially observable model where no relevant information about the state of the system is available. This can be expressed through a sensor model O containing a single dummy token o that is "observed" in all states, i.e., $O(s,a) = O(s',a) = \{s\}$ for for all s, s' and a. In planning, such models are known as **conformant**, and they are defined exactly like partially observable problems but *with no sensor model*. Since there are no (true) observations, the solution form of conformant planning problems is like the solution form of classical planning problems: a fixed action sequence. The difference between classical and conformant plans, however, is that the former must achieve the goal for the given initial state and "unique" state-transitions, while the latter must achieve the goal in spite of the uncertainty in the initial situation and dynamics, "for any possible initial state" and "any state transition" that is possible.

	Non-Observable Environment: No information gained after action	**Fully Observable Environment:** Exact outcome known after action
Non-Deterministic Multiple outcomes	**NOND Conformant Planning** *(Planning with NO Sensing)*	**FOND Conditional / Contingent Planning** *(Planning with Sensing)*
Probabilistic Outcome with probabilities	**Probabilistic Conformant Planning** **Partially Observable Marcov Decision Processes (POMDP)**	**Probabilistic Conditional / Contingent Planning** **Marcov Decision Processes MDP**

Figure 2.9 Classic Planning is extended to cover Non-deterministic and stochastic environments. A planner takes a compact representation of a planning problem over a certain class of models (classical, conformant, contingent, MDP, POMDP) and automatically produces a controller. For fully and partially **observable models**, the controller is closed-loop, meaning that the action selected depends on the observations gathered. For **non-observable models** like classical and conformant planning, the controller is open-loop, meaning that it is a fixed action sequence [Geffner and Bonet, 2016].

Classical Planning as Path Finding In planning languages, as PDDL or STRIPS, a planning task is represented through a finite number of *situations* or *states* that are described by a set of atoms or propositions/sentences, as explained in [Russell and Norvig, 2010] and shown in Figure 2.10. These states are changed by the execution of planning actions, i.e., dealing mainly with *state transformation problems*.

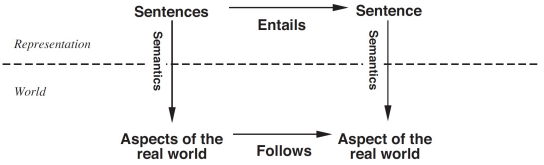


Figure 2.10 Correspondence between Real World and Representation: A compact description of models (planning problems) is usually expressed in languages featuring "variables" (Strips, PDDL, PPDDL, ...), where the **states** are the possible "valuations" of the variables. Sentences are physical configurations of the agent, and **reasoning** is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent. **Entailment**: Sentences $P_1, P_2, \cdots, P_n$ **entail** sentence P iff the truth of P is implicit in the truth of $P_1, P_2, \cdots, P_n$. In other words, if the world is such that it satisfies the P_i then it must also satisfy P [Brachman and Levesque, 2004]. Figure courtesy: AIMA book [Russell and Norvig, 2010]

Another important characteristic to recall is that there is a direct correspondence between *classical planning models* and "directed graphs", and between *classical plans* and certain "paths" over these graphs [Geffner and Bonet, 2016]. Classical planning, the simplest form of planning where actions have deterministic effects and the initial state is fully known, can be easily cast as a "path-finding problem" over a *directed graph*. Indeed, a classical planning model $\mathscr{S} = \langle S, s_0, S_G, A, f, c \rangle$ defines a weighted directed graph $G = (V, E, w)$, where the nodes in V represent the states in S, the edges (s, s') in E represent the presence of an action a in $A(s)$ such that s' is the state that follows a in s and the weight $w(s, s')$ is the minimum cost over such actions in s.

It follows from this correspondence that an action sequence $\pi = a_0, \ldots, a_m$ is a plan for the state model $\mathscr{S}$ iff π generates a sequence of states $s_0, \ldots, s_{m+1}$ that represents a directed path in the weighted digraph $G = (V, E, w)$, as illustrated in the Block World example in Figure 2.11. Thus, in principle any "path-finding algorithm" over weighted directed graphs can be used for finding plans for the classical planning model.

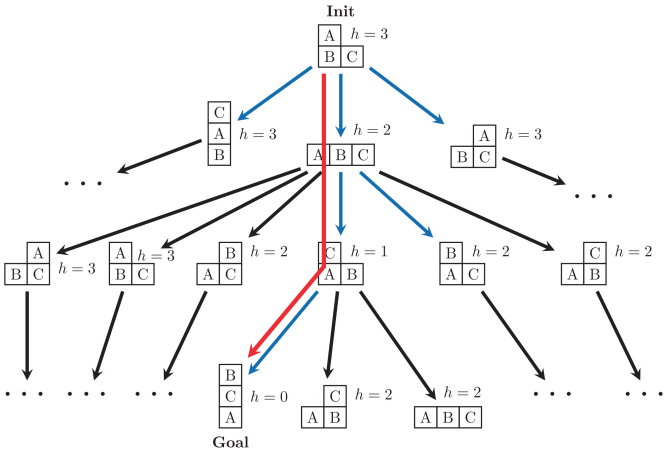


Figure 2.11 There is a direct correspondence between **classical planning models** and "directed graphs", and between **classical plans** and certain "paths" over these graphs: A fragment of the graph corresponding to a simple **Blocks World planning problem** involving three blocks with initial and goal situations as shown. The automatically derived heuristic values shown next to some of the nodes. A plan for the problem is shown by the path in red. [Geffner and Bonet, 2016].

AI Planning Algorithms and Resources

In terms of automated planning, different valuable literatures, like [Russell and Norvig] [Russell and Norvig, 2010], Ghallab [Ghallab et al., 2004], Lavalle [LaValle, 2006] and some tutorials available online [Geffner and Bonet, 2016; Ghallab et al., 2016b], address the important "planning algorithms" and applications. Additionally, for the planning community, International Planning Competition (IPC) became not only a standard way to compare the perfo rmance of planners, but also a source of a wide variety of "benchmarks" motivated by both real-world problems and challenging fundamental features of the planners [Komenda et al., 2016].

Furthermore, a large variety of useful web-pages/repositories of *planning benchmark models* can be found on-line. Like for example, "planning.domains" [3], hosted and developed by Andrew Coles and Christian Muise [4], provides a PDDL Editor with an integrated planning solver (planner) available to use on internet and could facilitate a successful planning resulting in a sequence of actions [5]. It also provides a collection of tools for working with planning domains and a programmatic access to a wide collection of PDDL benchmark domain and problem files.

Planning is the art and practice of thinking before acting [Haslum et al., 2019]. In fact, as mentioned by [Ghallab et al., 2016a] and illustrated in Figure 2.7b, Auto-

[3] `http://planning.domains/`
[4] `https://github.com/AI-Planning/classical-domains`
[5] `http://editor.planning.domains`

mated Planning is the field devoted to studying the reasoning side of acting. Specifically, we recall that, before acting, the central goal of AI planning is that of finding solution plans or an organized "set of actions" to carry out some activity.

We should also note the we usually distinguish between two distinct categories in planning models. The *restricted conceptual model* assumed in “Classical Planning" and the *extended planning models* that address temporal planning, on-line planning or planning in partially-observable and non-deterministic domains. From the classical planning to the extended models, the field of Automated Planning has experienced huge advances [Ghallab et al., 2016a]. Moreover, Planning can be done on-line or off-line, and agents and environments can correspond to types mentioned in Chapter 2 in Russell and Norvig AIMA book [Kovacs, 2012].

There are many approaches to solving planning problems, and many implementations of these approaches exist as well. Here it is enough to review the main characteristics of the following planners that are research prototypes developed by various university research groups [6]. More details about them can be found in different literatures and universities as in AIMA book [Russell and Norvig, 2010], in [Ghallab et al., 2004; LaValle, 2006] and others.

- *FF (Fast Forward)*. FF is a very successful *forward-chaining heuristic search planner* producing sequential plans. Although it does not guarantee optimality, it tends to find good solutions "very quickly" most of the time, and it has been used as the basis of many other planners, such as LAMA. FF does a variation of *hill-climbing search* using a non-admissible heuristic, which is also derived from *plan graphs*, like that used by Graphplan, IPP and others.

- *LAMA (Land Marks)*. LAMA is the winner of the *sequential satisficing* track of the 2008 International Planning Competition (IPC), where "satisficing" means that the plans are not necessarily optimal. One heuristic used is based on a domain analysis phase extracting landmarks, variable assignments that must occur at some point in every solution plan. The *landmark heuristic* measures the distance to a goal by the number of landmarks that still need to be achieved.

- *IPP (Interference Progression Planner)*. IPP is a descendant of *Graphplan*. *It produces optimal parallel plans*. Graphplan-style planners perform "iterative deepening A^* search" using an admissible heuristic, and use a special graph structure to compute the heuristic and store updates during search.

- *VHPOP (Versatlile Heuristic Partial Order Planner)*. VHPOP is a *partial-order causal-link* planner. It can work with either ground or *partially instantiated actions*, and it can use a wide variety of *heuristics* and *flaw selection*

[6] `https://www.ida.liu.se/~TDDC17/info/labs/lab4/lab4_planning.en.shtml`

strategies and different search algorithms (A*, IDA* and hill-climbing). Depending on the heuristic and search algorithm used, the solution may be optimal or not.

By executing some of these most popular and effective AI planning algorithms, we usually can generate "sequence of actions" for performing tasks and achieving the objectives, i.e. a plan for the goal. At the end, we need to use long-term strategies and generate a complete solution in advance, not just limited horizon lookahead. A variety of planners/solvers of different algorithms are available to run on various academic and research centres as LiU and Toronto university, such that one could solve its own planning problem and obtain a useful plan.

General purpose planners enable AI systems to solve many different types of planning problems. However, many different planners exist, each with different strengths and weaknesses, and there are no general rules for which planner would be best to apply to a given problem [Jiang et al., 2019].

		Planning agents	
		1	n
Execution agents	1	Single-agent planning **Fast Downward (FD)** [Helmert 2006]	Factored planning **Agent Decomposition-Based (ADP)** [Crosby et al. 2013]
	n	Planning for multiple agents **Threaded Forward Partial-Order (TFPOP)** [Kvarnstrom 2011]	Planning by multiple agents **Forward Multi-Agent Planning FMAP** [Torreno et al. 2015]

Figure 2.12 Conceptual planning schemes according to the number of planning and execution agents, with example MAP solvers that apply them. Planning by multiple-agents scheme distributes the MAP task among several planning agents where each is associated with a local task T_i . Thus, planning-by-multiple-agents puts the focus on the coordination of the planning activities of the agents. Unlike single-planner approaches, the planning decentralization inherent to this scheme makes it possible to effectively preserve the agents' privacy. [Torreño et al., 2017].

For completeness, it is worth noting that according to the relation between the number of planning agents and execution agents, we can have a general categorization for multi-agent planning (MAP) (by-multiple-agents) as summarized in the table in Figure 2.12, with example of solvers that are applied to them. This has a broad range of applications and more details about multi-agent planning can be found in [Kvarnström, 2011; Kovacs, 2012; Torreño et al., 2017; Komenda et al., 2016].

Domain-Independent Modelling of Planning Problems with PDDL

In this section, we will briefly review how to build the domain-independent "world model" with *Planning Domain Description Language* (PDDL), decomposed into two component files: *domain definition* and *problem definition*, in order to synthesize a plan that achieves a desired goal.

The adoption of a *common language* for modelling planning domains allows for a direct "comparison" of different approaches and increases the availability of shared planning resources, thus facilitating the scientific development of the field [Fox and Long, 2003].

As mentioned above, an automated planner could devise a plan to enact a list of available actions in a mission specification and achieve some objective at the end. However, before experimenting with any planner, trying to solve our planning problem, we need to write our own "model" in which we decide "what actions" to perform and "how" to perform them.

AI planning is model-based: an AI planning system ("planner", for short) takes as input a compact description (or model) of the initial situation, the actions available to change it, and the goal condition to output a plan composed of those actions that will accomplish the goal when executed from the initial situation [Haslum et al., 2019]. The planners we will use all accept the same form of input in PDDL, a "de facto standardized" planning language. This language provides a common syntax and semantics for planning formalisms with varying *degrees of expressivity*. The language is briefly discussed in section 10.1 of the Russell and Norvig textbook [Russell and Norvig, 2010], and though the book does not use the standard syntax.

PDDL describes the initial and goal states as *conjunctions of literals*, and actions in terms of their *preconditions and effects* [Russell and Norvig, 2010]. The PDDL model consists of two files: a "domain" definition and a "problem" definition. Mainly, given a domain, we can express different problems, that is, of different sizes, initial states and goals. As such, for solving a planning problem we need to implement the related two component files as follows, including the respective main ingredients:

- Planning **Domain Definition**:
  - *Predicates*: Relevant properties of objects, that can be true or false
  - *Actions* (operators as preconditions and effects): Means to change the state of the world
- Planning **Problem Definition**:
  - *Objects*: Things of interest
  - *Initial state*: The initial state of the world
  - *Goal specification*: Desired goal state

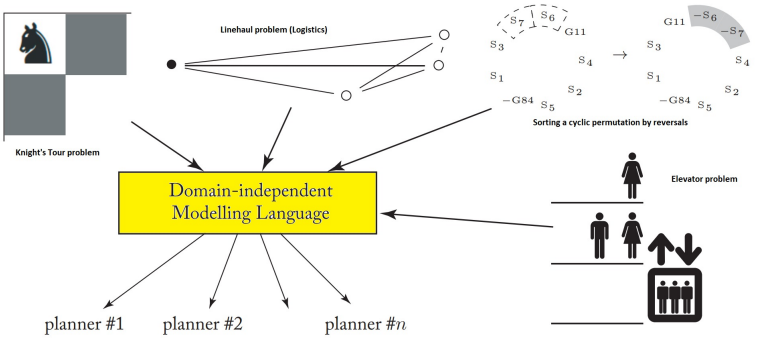


Figure 2.13 Conceptual overview of solving common planning problems: Knights Tour, Linehaul, Sorting a Cyclic Permutation by Reversals and Elevator problem in [Haslum et al., 2019]. The idea of domain-independence in AI planning is to place a modelling language between the planning problem to be solved and the planning system ("planner", for short) that solves it. A problem, once modelled, can be solved (or at least we can try to solve it) with many different planners, and a planner can be applied to any problem that can be expressed in the modelling language .

Other Planning Languages

Automated planning, as exposed in this work, make use of PDDL as a high-level programming language without considering the use of other relevant languages and techniques involved in various interesting research lines. But here we can give a short description about some of these concepts. Again we recall that these planning languages specialize in the generation of plans for achieving a given goal.

- *Extensions of PDDL* have been defined to model problems with uncertainty, in the form of a partially known initial state and actions with nondeterministic effects [Haslum et al., 2019]. MA-PDDL is another example that extends PDDL, which is designed to model multi-agent planning problems for planning both for and by several agents [Kovacs, 2012; Torreño et al., 2017].

- *Golog* is a new family of high-level programming languages suitable for writing control programs for dynamical systems [7]. In contrast to the Planning Domain Definition Language PDDL, Golog supports planning and scripting as well. *Planning* means that a goal state in the world model is defined, and the solver brings a logical system into this state. *Behaviour scripting* implements reactive procedures, which are running as a computer program [8].

[7] `http://www.cs.toronto.edu/cogrobo/kia/`
[8] `https://en.wikipedia.org/wiki/GOLOG`

- *Hierarchical Task Network (HTN)*. As a method of Automated Planning (which is an AI approach that plans for sequences of actions in order to transform the system from an initial state into a goal state), Hierarchical Task Network embodies a different view of what planning is. Instead of changing the state of the world model, a HTN planning problem is defined as a "set of abstract tasks" to be done, and "set of methods" for each task that represent different ways in which it can be carried out [Haslum et al., 2019]. HTN plans a *sequence of primitive actions* so that a certain "goal task" satisfying a condition is realized. HTN is based on *recursive decomposition* of compound tasks into smaller subtasks until reaching primitive tasks performable by planning operators, i.e., which are concrete, executable actions [Ahmadzadeh and Masehian, 2015].

- *Situation Calculus*. The situation calculus, a dialect of first-order logic [Reiter, 2001], is a logic formalism designed for representing and reasoning about dynamical domains [9]. The situation calculus represents *changing scenarios* as a set of *first-order logic* formulae.

- *Event Calculus*. In the previous Section, we showed how situation calculus represents actions and their effects. Situation calculus is limited in its applicability: it was designed to describe a world in which *actions are discrete, instantaneous, and happen one at a time* [Russell and Norvig, 2010].

  Consider a "continuous action", such as filling a bathtub. Situation calculus can say that the tub is empty before the action and full when the action is done, but it can't talk about what happens *during the action*. It also can't describe *two actions* happening at the same time, such as brushing one's teeth while waiting for the tub to fill. To handle such cases we introduce an alternative formalism known as event calculus, which is *based on points of time rather than on situations*. Event calculus reifies fluents and events [Russell and Norvig, 2010]

- *Event-driven Behavior Trees*. A behavior tree [10] is a mathematical model of "plan execution" used in computer science, robotics, control systems and video games. They describe switchings between a finite set of tasks in a *modular fashion*. Their strength comes from their ability to create very complex tasks composed of simple tasks, without worrying how the simple tasks are implemented.

  Recent works propose behavior trees as a *multi-mission control framework* for UAV, complex robots, robotic manipulation, and multi-robot systems.

[9] https://en.wikipedia.org/wiki/Situation_calculus

[10] https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)

3

WARA-PS Core System

A good overview and useful informations about the core system of **WASP Research Arena Public Safety** [1], also known as WARA-PS Core System, can be explored in the paper work of [Andersson et al., 2021].

Solving the tasks of this project is based on WARA-PS framework, shown in Figure 3.1, where we need to perform simulated experiments and evaluate our Rendezvous controller using the provided ROS-based simulators of a commonly used quadcopter research platform (DJI Matrice 100). In the second part of the project, we need as well to solve our high-level planning problem using the provided AI planner with the Task Specification Trees (TST).

Certain parts of the system need to be running on our computer through an easily installable *set of images* based on the use of *Docker* techniques, which allows us to run all the needed aspects of the system inside a "sandbox" *container* [WARA-PS, 2024]. In this way, this system works on different operating systems without the need to dual-boot or install another operating system. Then, by starting the software system's Docker container it automatically starts:

- **All software components**: that are required for a particular level of the system.

- **Tutorials**: A *personal Jupyter Notebook Server* needed to run the system tutorials without interference from others.

[1] `https://wasp-sweden.org/research/research-arenas/wara-ps-public-safety/`

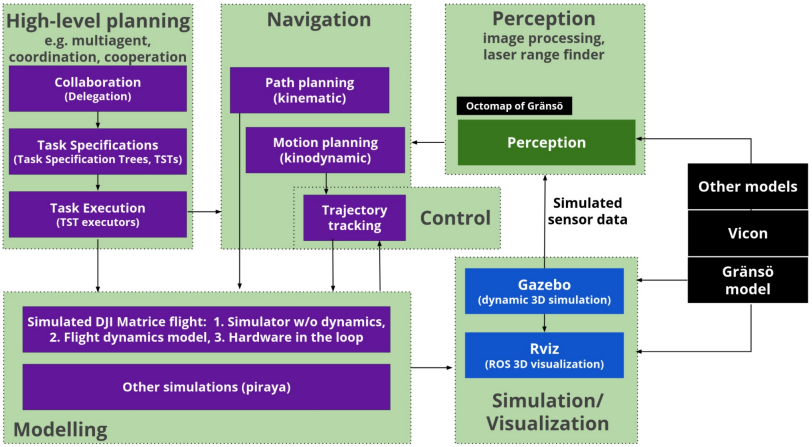


Figure 3.1 The architecture of the WARA-PS Core System: The central nodes of this ROS-based framework consist of: High-level Planning – Navigation – Perception – Control – Modelling - Simulation/Visualization. Courtesy: (LiU/AIICS) [WARA-PS, 2024]

3.1 Tutorials: WARA-PS Execution Environment

At *WARA-PS portal* (LiU/AIICS) [WARA-PS, 2024], it is provided a set of tutorials for a subset of the ROS-based WARA-PS Core System. The tutorials are based on a newly developed *Execution Environment* where it is possible to start/test the system and run DJI Matrice quadcopter missions in simulation. Through Python code we can test various aspects of the system and call its *functionalities*. However, for significant development efforts a new *Development Environment* is also available, where it will be possible to develop and test our own algorithms.

Jupyter Notebooks In general, the purpose of these tutorials is to demonstrate how to interact with the WARA-PS Core System at a *software level*. Some introductory tutorials are available as *Jupyter Notebooks*, which cover useful aspects needed for any form of software-based interaction with the system. These Jupyter notebooks are web-based connected to Jupyter Notebook Server and contain:

- **Notebook Basics**: using notebooks and interacting with other aspects of the system running outside the notebooks themselves.
- **ROS Basics**: a very brief introduction to ROS.
- **Basic DJI Simulator (djisdk)**: learning about the DJI Matrice 100 simulator, with support for simulating *system dynamics* and running alternative *trajectory following controllers*.

- **Single Agent Systems**: how each platform becomes an *agent*, and which *functionalities* are available for use even when there is only a single agent in the system.
- **Task Specifications for Single Agent Systems**: task specification and execution for a single agent, using Task Specification Trees (TST).

The tutorials are being constructed and published incrementally. Eventually they will cover a large part of the software system, including the following aspects and their relationships: High-level Planning, Navigation, Perception, Control, Modelling, Simulation/Visualization.

To go through these tutorials, certain aspects of the WARA-PS Core System needs to be running on our computer by installing a *set of images* based on the use of *Docker container*.

3.2 WARA-PS Development Environment

Using the tutorials Jupyter notebooks, it is easy to edit our own code, test and learn about various aspects of the running system. However, these notebooks are not intended to be as a "development environment" which is supposed to be provided as another extension/environment in WARA-PS system. In fact, any modifications we make to the tutorials notebooks will disappear by the next restart.

Thus, a new development environment is built on the execution environment in WARA-PS framework and provides many additional functionalities related to the core system, including those functionalities that are present in the WARA-PS tutorials. In this way, we have the ability to develop and test our own algorithms and specify our own missions with full support for standard development tools.

Software installation Unlike the tutorials, where all required software is encapsulated in Docker containers, software development instead requires other software installed directly on our *Development System*, i.e., Ubuntu LTS installation that runs either directly on computer or in a virtual machine.

The software installation process is mostly automated through *Ansible Playbooks* – a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications. However, Ansible is designed to be able to install software on remote hosts as well, so the playbooks will use SSH even for a local installation of the software on the same Development System, where these Ansible Playbooks are running.

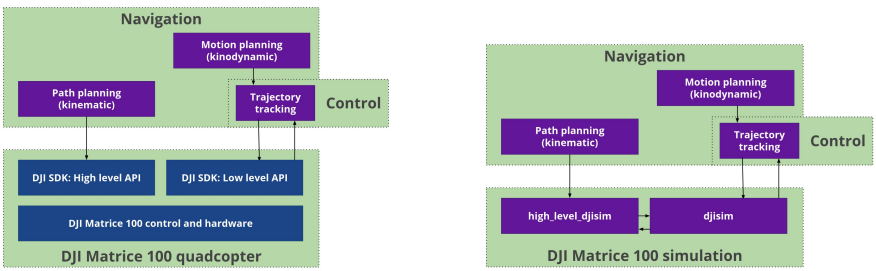
3.3 DJI SDK API

The DJI Matrice 100 natively provides an SDK (Software Development Kit), the ROS DJI SDK, through which an actual physical quadcopter can be controlled in

two ways:

- **DJI SDK low-level API**: allows us to send a continuous *stream of control signals* where we can directly control for example roll, pitch, yaw rate, and thrust.
- **DJI SDK high-level API**: allows us instead to specify a *sequence of waypoints* to fly through and the *speed* that the quadcopter should have in each of these waypoints. This API uses functionalities in the internal DJI software to determine which control signals need to be sent to the lower level API.

Both these control systems are provided by DJI and are very useful in a variety of missions, where you would choose which API to use depending on whether you simply want to reach a particular point or whether you want to control exactly how the quadcopter flies.



(a) **(b)**

Figure 3.2 (a) The architecture of the **ROS DJI SDK** provided by the DJI Matrice 100, through which an actual physical quadcopter can be controlled in two ways: A path planner generating a sequence of waypoints could use the high level API. Alternatively, a motion planner could generate a trajectory to be followed by a specific trajectory tracking controller that communicates with the low level API. (b) **WARA-PS simulators**: A pair of simulators have been defined within the WARA PS project, following the protocols defined by the ROS DJI SDK. That is, the simulation should be provided with either high-level input (to `high_level_djisim`) or low-level input (directly to `djisim`).

3.4 WARA-PS Simulators Overview

In addition to being able to actually fly a DJI M100, we need to be able to simulate *what would happen* if we flew in a particular way. Therefore, a pair of simulators have been defined within the WARA-PS project, following the protocols defined by the ROS DJI SDK.

- *Low-level simulator*: `djisim` accepts some of the same control signals as the actual low-level API (roll, pitch, yaw rate and thrust). Instead of actually

controlling *actuators* on a physical quadcopter, the simulator then simulates the *dynamics* of the quadcopter to approximate what would have happened if an actual Matrice 100 was given these signals. The output is a new "predicted" world position in a particular coordinate system, together with a "predicted" resulting attitude.

- **High-level simulator**: `high_level_djisim` accepts the same type of input as the actual high-level API and generates the appropriate control signals to send to `djisim`.

NOTE Since both simulators take their input on exactly the same form as the actual Matrice 100 and using the same ROS topics as the ROS DJI DSK, they can be used as a *plug-and-play replacement* for the Matrice 100, allowing us to

- test various control algorithms in simulation
- and then use the same control algorithms in real flight at a later date.

3.5 Running DJI Simulator in Full Dynamics with MPC

As illustrated in Figure 3.3, it is worthwhile to give a short overview of an example, provided by WARA-PS framework, illustrating the ROS communication underlying one DJI simulation configured with a trajectory tracking controller using MPC. In fact, the DJI simulator `djisim` can be configured in three different ways. In the following we summarize the important ROS topics and nodes interconnection corresponding to this simulation example:

- Activating dynamics with `dynamics:=true` does mean that the trajectory information generated by `high_level_djisim` cannot be sent directly to `djisim`. Then, starting `djisim` with `mpc:=true` we specify the use of a controller based on Model Predictive Control, based on ETH paper, which provides the desired low-level control input to djisim. We assume that `high_level_djisim` is provided with a trajectory consisting of a sequence of waypoints to follow, according to the DJI SDK API. This trajectory is not visible in the ROS graph.

- Then `high_level_djisim` generates messages on `/dji0/command/trajectory`, as in the simple simulation. It also generates pose information on `/dji0/command/pose`. These, together with information on an odometry topic, becomes the input to the `/dji0/mav_nonlinear_mpc` node, which implements the MPC-based controller and continuously generates new commands for the system on `/dji0/command/roll_pitch_yawrate_thrust`.

- This information is given to the `/dji0/converter_djisdk_mav_node` node, which also takes DJI SDK information from velocity, attitude and `world_position` topics in order to generate a single standard DJI SDK message, which is sent on `/dji0/dji_sdk/flight_control_setpoint_generic/` to djisim.

- Then djisim simulates the low-level dynamics of the Matrice 100, determines the location and attitude of the quadcopter in the next time step, and generates a number of messages using the standard DJI SDK format, such as velocity, attitude, `gps_position` and `gps_health`.

- The messages sent to `world_position` topic are converted by `sim_converter` to representations that are sometimes more convenient to use internally.

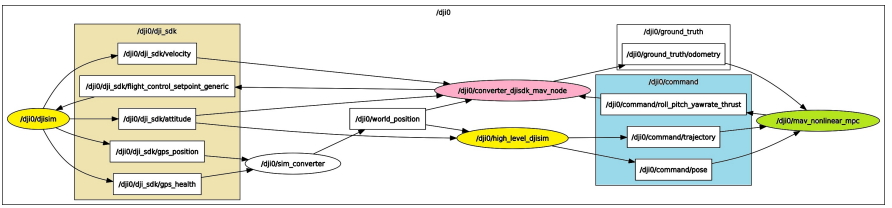


Figure 3.3 An example of simulation with a trajectory tracking controller using MPC run on WARA-PS simulator. This generated ROS connection graph shows how the simulation is connected including the ROS nodes and topics that are involved. Since this is a closed control loop, there is no perfect place to begin. Thus, in WARA-PS tutorials it is decided to follow the execution graph from `/dji0/high_level_djisim`, which is in the middle of the image. We assume that `high_level_djisim` is provided with a trajectory consisting of a sequence of waypoints to follow, according to the DJI SDK API. This trajectory is not visible in the graph.

3.6 Using WARA-PS Existing Tools

WARA-PS framework includes existing tools, outlined in the following points, of which it would be useful to take advantage for solving our two main problems:

1. `djisim_m100` **simulator**: WARA-PS framework includes `djisim_m100` simulator with a PID and nonlinear MPC controller for simulating trajectory tracking with Take-Off and Landing modes.

   Based on the ROS node `dji_sdk` which is the main wrapper node for DJI Onboard SDK, we can make use of several existing ROS topics and services useful for implementing autonomous navigation and Vertical Take-Off and Landing (VTOL) of the UAV quadrotor DJI Matrix 100.

In addition, it is possible to plug in our own controller (outer loop), by starting `djisim_100` without any controller. In this way, we can investigate the performance of the new implemented controller designed for the rendezvous landing manoeuvre of the DJI quadrotor on a ground mobile robot like Spot.

2. **Task Specification Trees:** WARA-PS framework includes *Task Specification Trees* (TSTs) that can be executed by multiple agents. These TST representations will be used in our AI planning problem. Some illustrations and description can be explored in the work of [Doherty et al., 2013] in [Doherty et al., 2013].

3. **RTK-GNSS System:** To be able to perform a safe and soft landing in the experiments on the real robots in future work, the motion of the quadruped Spot (Q-UGV) needs to be accurately tracked and positioned by the UAV DJI M100. For achieving this high-accuracy positioning in the landing manoeuvre, it is possible to use a combination of Real Time Kinematic (RTK) technology with Global Navigation Satellite Systems (GNSS) augmented by common inertial sensors, for estimating the position. This RTK system is provided by SAAB Combitech AB through the WASP Research Arena (WARA) Public Safety [Andersson et al., 2021].

4

Rendezvous Problem: Simulation and Control using ROS

In this section/chapter the focus is put on the simulation work for the Rendezvous landing task in this project. It will be to describe how ROS has been used in the robot software implementation using `djisim`, the WARA-PS quadrotor UAV simulator, and how a new simulated two-wheeled mobile robot model has been assembled using the `URDF` (Unified Robot Description Format) together with `Gazebo plugin` controller and then integrated in our simulated Rendezvous scenario to simulate the movement of Spot robot on the ground. At the end, we describe how to work with the recorded data on the robots in simulated experiments and illustrate the evaluation results using the generated bag file.

To facilitate the development of different control strategies, `djisim` simulator provides a simple ROS launch file for running the quadrotor UAV simulator without any controller which allows us to plug in our own controller. In the following section 4.6, we present the construction of our Rendezvous controller using a multi-dimensional PID control scheme. In this way, this implemented Rendezvous algorithm using PID controller acts as an ideal starting point to close the loop with desirable properties and then to implement, in possible future extensions, more advanced and suitable control strategies using Model Predictive Control (MPC) framework, as discussed in chapter 6.

4.1 Multi-rotor UAV Control Problem

For clarity purpose, before addressing the autonomous landing algorithm in the next section, it is worth to give here a short overview of the quadcopter control problem. Many researchers have been working on the control problem for quadrotor aerial

robotic vehicles, and they found that *the control problem to track smooth trajectories is challenging* for several reasons we quickly summarize in the following, reviewed from [Mahony et al., 2012]:

- *The system is underactuated*: there are four inputs $\mathbf{u} = (T_\Sigma, \tau_x, \tau_y, \tau_z)$, while $(\mathbf{R}, \xi) \in SE(3)$ is six dimensional, i.e., six outputs $\mathbf{y} = \begin{pmatrix} x & y & z & \phi & \theta & \psi \end{pmatrix}$. In general, $\mathbf{R}$ denotes the rotation matrix and ξ denotes the position vector of the quadrotor in the global (inertial) frame.

- The *aerodynamic model* described in 5.3 is only an approximate.

- *The inputs are themselves idealized*. In practice, the motor controllers must overcome the *drag moments* to generate the required speeds and realize the input thrust (T_Σ) and moments (τ). The dynamics of the motors and their interactions with the drag forces on the propellers can be difficult to model, although first-order linear models are a useful approximation.

- *Hierarchical controller structure* is suitable for multi-rotor UAVs. As illustrated in Figure 4.1, this controller is usually separated into two cascaded controllers. In the inner loop we have a firmware *attitude controller* running on the micro controller (autopilot) of the quadcopter. Then, another controller running in the outer loop, PID or Model Predictive Control (MPC), can be designed and implemented for *position tracking*. *This separation is useful to add robustness with respect to time delays in on-board estimation*, as stated by [Bloesch et al., 2010].

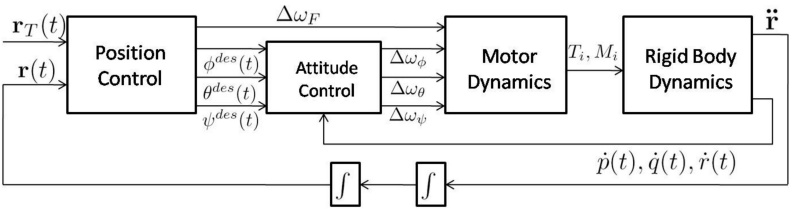


Figure 4.1 Cascade controller structure for multi-rotor system: In this nested control loops for position and attitude control, the inner attitude control loop uses onboard accelerometers and gyros to control the roll, pitch, and yaw while the outer position control loop uses estimates of position and velocity of the center of mass to control along the trajectory. Usually, position control is split into two parts: An outer trajectory tracking controller calculates attitude and thrust references, that are tracked by an inner attitude tracking controller. (Courtesy: [Mellinger et al., 2010])

This commonly used hierarchical or cascade controller structure is proposed to follow in this project for solving the rendezvous landing problem by implementing

the designed Rendezvous autopilot algorithm in the outer loop based on PID or MPC approach. Similar to [Bloesch et al., 2010], [Kamel et al., 2017b] and many other related works, for better tracking performance in the case of implementing MPC control scheme, the closed loop dynamics of the attitude controller will be included in the dynamical model of the quadrotor, as described in the multi-rotor system model in 5.3 and 5.4. This model formulation is in fact also used in our WARA-PS simulator adapted from [Kamel et al., 2017b].

(a) **(b)**

Figure 4.2 Autopilots for landing in uncertain conditions: a) Schematic overview of our Rendezvous landing problem in this project, where the autopilot's task is to force the unmanned air vehicles (UAV) DJIM100 to track a time-varying reference trajectory leading to the desired landing platform at the top of Spot robot. b) An example of Autonomous Landing at ASDS spaceX: "Of Course I still Love You". (Courtesy: Wikipedia)

4.2 Rendezvous Landing Control Strategy

As mentioned in [Marconi et al., 2002], a problem of paramount importance in autonomous flight is the design of autopilots for landing in uncertain conditions. The task of the autopilot is to smoothly regulate to zero the vertical distance from a landing platform, which in some environment structures could be subject to large vertical deviations because of different forms of disturbance, for example, due to high sea states as shown in Figure 4.2b.

In general, in robotic networks, *rendezvous* refers to the task of driving each of the agents in a robotic network, equipped with sensors detecting other agents, towards a common configuration without the need for the agents to communicate with each other [Caicedo-Nunez and Zefran, 2008].

Over the last years, various projects considered the concept of autonomous landing of UAV on a moving platform, which is also considered as a Rendezvous problem. Different approaches used for the implementation of this type of Rendezvous algorithm can be read in [Muskardin et al., 2017; Persson et al., 2017; Persson, 2021; Persson and Wahlberg, 2019; Bereza et al., 2020]. It is worth reviewing some

important features that would characterize the various Rendezvous landing algorithms discussed in the literature:

- First, many such algorithms consider *cooperative* landings in which connection/interaction between robots is necessary for exchanging information like velocity and trajectory to synchronize the rendezvous manoeuvre safely.

- Another feature is related to take in consideration the computation efficiency when investigating the performance difference between the *centralized* MPC and *distributive* MPC (DMPC). In the centralized algorithm, the control inputs for both vehicles are computed by the same process, on the drone computer, as shown in Figure 6.5. On the other hand, the distributed algorithm can be handled to control the computation load and divide it between the cooperating agents, i.e., it consists of two separate algorithms, one algorithm on each robot. This strategy is common in *Formation Control* of fleet of UAVs or in robot fleet management.

- Further, another important feature regards the separation of the quadrotor UAV motion into two sets of equations corresponding to *horizontal* and *vertical* dynamics, as illustrated in Figure 6.6.

The rendezvous landing problem in this work is in itself a *position tracking* problem before performing the landing manoeuvre at the end of the mission. This is solved in a non-cooperative manner for simplicity and better understanding the problem. In particular, the aerial robot needs to identify the position of the ground mobile robot in the environment and then to maintain its proper position relative to it, before zeroing this relative distance in a safe and soft way. Mathematically, Rendezvous problem, between the quadrotor UAV DJI M100 and Spot robot, is defined to be achieved when zeroing all the following *relative position error equations*

$$\begin{aligned} \Delta x &= x_{dji} - x_{spot} \\ \Delta y &= y_{dji} - y_{spot} \\ \Delta h &= h_{dji} - h_{spot} \end{aligned} \tag{4.1}$$

For simplicity, we consider a simulated two-wheeled *substitutive robot* (TWR) to approximate Spot robot as a moving target in the simulated experiments of our autonomous Rendezvous algorithm. And as a position tracking control, needed to compute the distance and to control our quadrotor UAV (DJI Matrice 100) in the Rendezvous algorithm, we consider the same type of PID control adapted from [Hoenig and Ayanian, 2017] used in their work with the crazyflie metapackage [1], developed as part of research at the ACT Lab at the University of Southern California.

[1] `https://github.com/whoenig/crazyflie_ros`

However, in order to successfully perform the required Rendezvous task it is crucial to have precise position tracking, especially when operating in realistic environments. In fact, external disturbances may heavily affect the flight performance and when flying in the vicinity of some structure or nearby the real target ground robot platform itself.

Therefore, a model-based controller like MPC might be an interesting future control design to improve the rendezvous performance in real world, on the real robotic platforms. Theoretical development of Rendezvous mission using MPC design, of which the implementation will be considered in future works, is investigated in chapter 6. In MPC we have the advantage of incorporating a plant model with linear or non-linear dynamics, constraints and a cost function with high interpretability.

A good overview of various *Model Predictive Control* strategies for multiple classes of unmanned aerial vehicles (UAVs) is presented in the paper from ETH Zurich in [Kamel et al., 2017b], where Various implementation hints and practical suggestions are provided with focus on MPC. In this way, we need to replace the error vector in the objective function of the optimization problem in MPC scheme by the error vector defined in 4.1, i.e., *to replace the reference x_{ref} by the position of ground robot.* As such, the quadrotor UAV need to track Spot position and perform the landing successfully.

In addition, in the real experiments the planned rendezvous trajectory can be executed using RTK GPS, where the measurements are fed back to the PID or MPC controller to compute the appropriate *error vector*.

4.3 Development System and Technical Aspects

Before integrating any DJI controller (inner loop and/or outer loop), we need the Development System installed on our computer/machine running with Windows 10. For this development system we choose the paired versions of Ubuntu 20.04 LTS and ROS Noetic that run through WSL2 (Windows Subsystem for Linux 2) [2].

All that being installed, we need to have certain parts of the WARA-PS Core System running on our machine (Windows 10) together with other additional software installed directly on our Development System, in the source directory "`src`" in WARA workspace "`wara_ws/src`", required for Software development.

Robot Operating System (ROS) is usually set up to be used in systems with multiple robots, as in our work, where each robot system is assigned a particular "namespace" used to address its named nodes, services and topics. Some separate nodes need to be realized for different components in our Rendezvous mission, as well as WARA-PS system is also based on the use of ROS and realized as a ROS workspace. In ROS development, as discussed in the following sections, many

[2] `https://jack-kawell.com/2020/06/12/ros-wsl2/`

aspects of the system require *publish* and *subscribe* communication channels, that is, *ROS topics*. Additionally, functionalities are divided into nodes that can call each others through *remote procedure calls*, that is, *ROS services*.

DJIM100 Software Development Kit (SDK)

It is important to mention that one of the main open source software for our project is the DJI Software Development Kits (SDK). The SDKs are designed to allow "registered developers" to fully access the capabilities of the quadrotor by running their own applications on the drone. The two SDKs that can be used are the **Onboard-SDK (OSDK)** for the onboard NUC computer which is mounted on the DJI platform and **ROS-SDK** which is used for the base station. The ROS-SDK works in conjunction with the OSDK and allows developers to access all of ROS capabilities through its messages and services.

Thus, these SDKs of DJIM100 platform are powerful and enable access to most functionalities and support cross-platform development environments such as Robot Operating System (ROS), Android, and iOS. However, as stated in the paper [Sa et al., 2017], it is strongly not recommended to use `ROS services` for sending control commands as designed by the manufacturer [3]. Instead, they modify the onboard SDK with a *ROS wrapper* in their paper to send direct control commands via *serial communication*.

Specifically, the variety of *sensing data* can be accessed using the OSDK through *serial communication*, such as IMU, GPS, RTK, barometer, magnetometer, and ultrasound measurements. A user can also configure an update rate for the sensor up to 100 Hz [Sa et al., 2017]. Furthermore, with the OSDK, the developers can control the UAV without a remote controller, install third party sensors, and execute precise trajectories [Sharma, 2019].

The common *transmitter inputs* are pitch, roll angles (rad), yaw rate (rad/s), and thrust (N). However, the *vertical stick input* of the platform is velocity (m/s) that permits easier and safer manual flight. The autopilot compensates total thrust variation in the translational maneuver. There are three control mode named: Function "F", Attitude "A", and Position "P". The `"F"` mode is the one of interest because it allows external control inputs such as serial commands, i.e., it allows other applications to run on the platform . `"A"` and `"P"` indicate attitude and position control modes [Sa et al., 2017; Sharma, 2019].

4.4 ROS Interface of the Simulated Robots

In this section, we give a short overview of the necessary ROS interfaces used by the two robots involved in the Rendezvous landing process in this project. For the aerial robot, quadrotor UAV DJI Matice 100, important components and the underlying architecture of the ROS-based WARA-PS simulator is briefly introduced in

[3] `http://wiki.ros.org/ROS/Patterns/Communication`

chapter 3. Here and in the following sections, we give a brief description about the ROS interface related to the ground mobile robot used in the Rendezvous simulation.

In Figure 4.3 we can see the output of `rqt_graph` tool which gives an overview of this ROS interface composed of all ROS nodes that are needed for running our Rendezvous simulation, and the topics on which the nodes are communicating. This graph of ROS nodes and topics is very helpful for debugging.

The main software documentation and packages for the WARA-PS simulator is organized on the GitLab LRS repositories maintained by *Artificial Intelligence and Integrated Computer Science* (AIICS) division at Linköping University (LiU) [4]. These repositories are not public but need to have access to the WARA-PS Portal [5] and to the djisdk tutorial notebook [6]. However, some useful information and examples regarding the WARA-PS control development environment can be read on the related GitLab LRS repository [7].

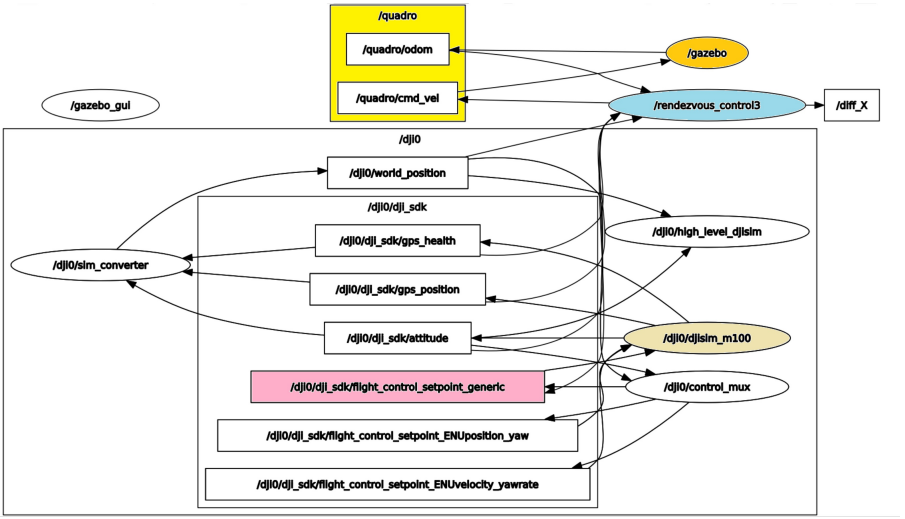


Figure 4.3 RQT graph visualizing the different ROS nodes and topics of our rendezvous simulation and their communication in ROS using `rqt_graph`. This consists of the `djisim` simulator of WARA-PS framework with the namespace `/dji0` (including all the fundamental nodes and topics), the two-wheeled simulated robot `/quadro` (approximating the movement of Spot), the Rendezvous node (running the corresponding algorithm) and the Gazebo simulator (including plugins).

[4] `https://www.ida.liu.se/divisions/aiics/`
[5] `https://portal.waraps.org/`
[6] `https://aiics.waraps.org/index.php/lrs-lrs_jupyter-md-djisdk-md/`
[7] `https://gitlab.liu.se/lrs/lrs_devenv_control/-/blob/master/README.md`

After installing the simulator repositories (cloning from LiU GitLab [8]) in the source directory in WARA workspace "wara_ws/src" on our development system, WARA-PS simulator is compiled and set up locally. This source directory contains all the simulation packages needed for the simulated and real quadrotor UAV (DJI Matrice 100). Yet, this simulator is the tool that enables the development of our Rendezvous algorithm, to be deployed on a real UAV in future work. As mentioned previously, this interface is designed to mimic most of the interfaces on the real DJI M100, to make the transition of the implemented algorithm as simple as possible.

For the ground robot platform, the quadro_sim package contains all the implementation needed by the simulated two-wheeled robot (TWR) on the ground (called quadro), as an approximate simulated model for Spot robot in the Rendezvous landing manoeuvre.

In the *RQT graph* in Figure 4.3, Gazebo is only shown as one ROS node but internally all the Gazebo plugins are running, such as a *Differential Drive controller* for sending Twist messages to quadro using various topics like velocity /cmd_vel and odometry /odom. In this way, the following messages can be published in the /quadro namespace:

- nav_msgs/Odometry message on the odom topic.
- geometry_msgs/Twist message on the cmd_vel topic.

All the states needed by the *position tracking controller* are contained in these Odometry and cmd_vel message. Namely, these states are the position, orientation, and linear and angular velocity of the TWR. Hence, the Rendezvous algorithm running alongside the simulation can directly subscribe to the /Odom topic. Additionally, we publish the difference (error) in positions between the two robots on the /diff topic using message of type geometry_msgs/Pose.

4.5 Simulating the Robots using Gazebo and WARA-PS Framework

As stated in [Furrer et al., 2016], to test algorithms on UAVs, one needs access to expensive hardware and field tests usually consume a considerable amount of time and require a trained safety-pilot. Most of the errors, occurring on real platforms, are hard to reproduce, and often result in damaging the UAV .

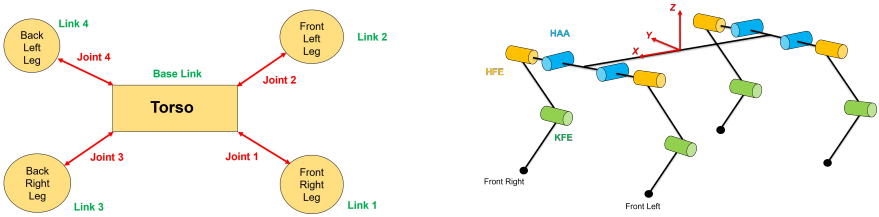
In fact, using the WARA-PS simulation framework is fundamental for our simulated experiments deploying the DJI M100 quadrotor in our Rendezvous mission and of course these two simulators will be also convenient in future real experiment scenarios to reduce field testing time and to separate various problems for testing, making debugging easier, and finally reducing crashes of the real DJI UAV. All ROS

[8] gitlab.liu.se

components were designed to be analogous to its real world counterparts and they can be used as a plug-and-play replacement for the DJI Matrice 100 [Andersson et al., 2021].

When the WARA-PS DJI simulator and development environment are already installed on the development system (my computer), we can open several Linux shells in *Windows Terminal*, or in Ubuntu terminal, and then we run the command `roscore` to start the *ROS master* and the *parameter server*, or the same by using some `roslunch` command, in order to run and test our control algorithms and specify our own missions.

The performance of our implemented Rendezvous algorithm (using PID or MPC) will be evaluated in simulation, with the ambition to illustrate, through rqt plotting in ROS, how the *relative distance* $(\Delta x, \Delta y, h)$ between the quadrotor and Spot converges to zero, as well as other useful plots. The software structure of the Rendezvous simulation is illustrated in Figure 4.3 using ROS *RQT tools*. In this ROS graph, we can see the interconnection between ROS nodes and topics necessary for our simulation work.



(a) **(b)**

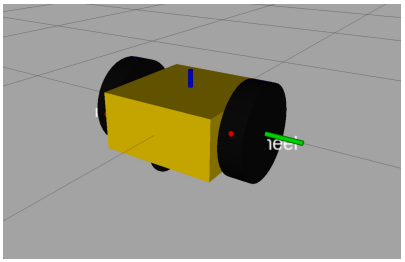
Figure 4.4 (a) Modelling Spot robot basic shapes using ROS: we layout the basic shapes of a suggested model structure for the quadruped robot Spot by using ROS simple geometry. Basically, it's a robot composed by 5 links and 4 joints. Every robot needs a base link, in this case, the torso is in charge of connecting all the parts of the robot. (b) Configuration of Spot Kinematic chain Structure: Spot robot uses 12 *proprioceptive actuators* to control 4 limbs. The numbering of joints starts from right front *hip Abduction/Adduction HAA* joint and progresses to *hip Flexion/Extension (HFE)* as well as *knee Flexion/Extension (KFE)* joints.

Modelling Two-Wheeled Differential-Drive Robot using Gazebo for Substituting Spot Robot in Simulation Experiments

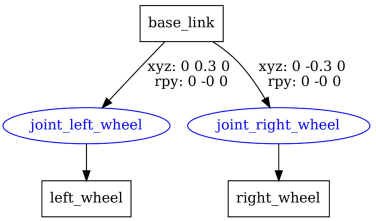
For future work, we model the quadruped Spot robot by using ROS simple geometry, of which it will be possible to design the corresponding simulation model by creating the URDF model (Unified Robot Description Format) for a four-legged robot, that will describe the main components of our robot and enable it to be visualized and controlled by ROS tools, such as RViz and Gazebo. As illustrated in Figure 4.4, this model would consist of a torso with four cylindrical legs. Each leg is attached to the torso with a revolute joint. However in this study, as mentioned

previously, this work can also be simplified by considering a two-wheeled robot (TWR), of which we build a Gazebo simulation model using URDF for simulating the movement of Spot robot in the simulated rendezvous manoeuvre.

Our 2-wheeled mobile robot named "Quadro" is basically a robot composed by 3 links and 2 joints, as shown in Figure 4.5. In URDF every robot needs a *base link*, which in our case is the gold chassis that is in charge of connecting the two wheels of the robot.



(a)



(b)

Figure 4.5 (a) Our 2-wheeled robot named "Quadro" in RViz, used to approximate the movement of Spot robot on the ground. Basically, it's a robot composed by 3 links and 2 joints. Every robot needs a base link, in this case, the gold chassis is in charge of connecting the two wheels of the robot. (b) Graphviz diagram of our URDF file created by executing the following commands after checking the URDF file using `check_urdf` tool, first by running "`urdf_to_graphiz quadro.urdf`" and next "`evince quadro.pdf`".

After having a working URDF model of our two-wheeled robot, we need to launch it into Gazebo and move it around in the simulated environment. Therefore, we first make some modifications to our URDF file to add simulation-specific tags, *Gazebo tags*, so that it properly works in Gazebo world. In particular, *Gazebo plugins* need to be added to the robot URDF model code to enable control of the robot movement in simulation. In general, Gazebo plugins are needed to simulate controllers that publish ROS messages for motor commands. *Transmission blocks* for the model joints may also be needed for having greater control of our model [Fairchild and Harman, 2017]. This Gazebo plugin is implemented for setting up `differential_drive_controller` that will publish necessary values using different *topics*. Specifically, the `/cmd_vel` is the topic in which we send `Twist` messages in ROS of type `geometry_msgs`, by reading from the keyboard or by implementing a Python node and then publishing to Twist.

This ROS Twist message contains two subsections: linear velocity and angular velocity. In ROS conventions, we use the `rostopic pub` ROS command to publish the linear and angular `Twist` data in order to move the mobile robot. The Python script is a ROS node that will accomplish essentially the same thing. This Python script send a `Twist` message on the `cmd_vel` topic to move our TWR robot for-

ward in a straight path or to make it turn in a circle in a simple example. We call our Python node `Control_quadro.py` and saved it in the `scripts` folder in our `quadro_sim` package. To make the script executable, we execute the Ubuntu command:

```
chmod +x Control_quadro.py
```

The logic behind our two-wheeled robot design and navigation is the same as in turtlesim in ROS tutorials [9]. In fact, in this tutorial series one can learn how to create python scripts to move the turtle, in order to practice the ROS basics.

For completeness, It is worth noting that Gazebo uses SDF, which is similar to URDF, but by adding specific Gazebo information, using the aforementioned Gazebo tags, we convert our quadro URDF model file into an SDF-type format [Fairchild and Harman, 2017]. Moreover, after setting up the differential drive controller using the corresponding Gazebo plugin, we get other topics available, alongside the aforementioned `/cmd_vel`, that can be checked with the ROS command `rostopic list`.

4.6 Rendezvous Algorithm using PID with ROS

A first important step for autonomous flight of a quadcopter is hovering in place [Hoenig and Ayanian, 2017]. This also requires the ability to take off from the ground and land after the flight. All these basic motions require a *position tracking controller*, which takes the UAV's current position as input in order to compute new commands for the DJI quadrotor. Hence, this position tracking controller replaces the teleoperating human.

In this section, we present the important aspects regarding the implementation of our algorithm using Python and ROS programming to control the Rendezvous landing process. Next in the upcoming section 4.7, the *RQT toolset* will be introduced and used to monitor the two robots movements and evaluate the performance of this algorithm.

Within the new created `quadro_sim` package, we implement a Rendezvous landing algorithm in Python using position tracking control of type PID similar to the PID control studied and proposed in [Hoenig and Ayanian, 2017]. In particular, in this implemented algorithm we use PID control for each of UAV's four dimensions of control: pitch, roll, thrust, and yaw, i.e., four independent PID controllers for x, y, z, and yaw, respectively. The various parameters can be tuned in a `config` file `pid_rendezvous.yaml` implemented in the folder `quadro_sim/config`.

Again as discussed previously, we remember that the DJI M100 is controlled by a cascaded design controller, where the inner attitude controller is part of the firmware. Our `rendezvous_control` node run a separated outer PID controller

[9] `http://wiki.ros.org/turtlesim/Tutorials/Moving%20in%20a%20Straight%20Line`

(or possibly using MPC in future work) which takes the current and target positions as input and produces a setpoint, attitude and thrust, for the inner controller.

In this project, the target position refers to the current position of the simulated two-wheeled robot (TWR) on the ground that can be received from the robot position feedback topic: `/quadro/odom`, of the built-in ROS message of type `nav_msgs/Odometry`. This topic is also generated after setting up the differential drive controller implemented in the Gazebo plugin. We remember that in ROS programming, one can get the definition of odometry from the following command:

```
rosmsg show nav_msgs/Odometry
```

In this way, to track the simulated quadro robot on the ground we use the data gathered from these moving sensors to estimate the change in the robot's position over time, that is, to estimate the current position of the robot relative to its starting location.

Here we have to mention that while *state estimation* is crucial on real robots, in the simulation this feature can be replaced by a generic *ideal* sensor directly provided by the Gazebo plugin. In this way, data obtained about position, orientation, linear and angular velocity of the simulated TWR mobile robot are not replicated accurately in the simulator.

WARA-PS djisim Simulator: Plug in our Flight Controller

In the following, we review the fundamental issues we need to take in consideration when plugging our own controller into WARA-PS DJI simulator:

- Flying DJI M100 using flight control topics: Usually, the user send flight control setpoints to the drone by publishing `/flight_control_setpoint_generic` topic, which are subscribed by the `djisim_m100` node. This fight control topic has message type `sensor_msgs/Joy` and the control mode should be roll, pitch, thrust, yaw rate. See "Details on flight control setpoint" [10] for how to set the control mode flag.

- Environment variables: To use the simulator in *basic mode* we have to set the following variables to a suitable value. These given origin coordinates are the ones used in the Gränsö testing area:

```
export WORLD_ORIGIN_LAT=57.7605573519
export WORLD_ORIGIN_LON=16.6827607783
export WORLD_ORIGIN_ELEVATION=29.85.2
```

- Starting the WARA-PS simulator: To plug in our own controller (outer loop), `djisim` can be started without any controller by running the following `roslaunch` command:

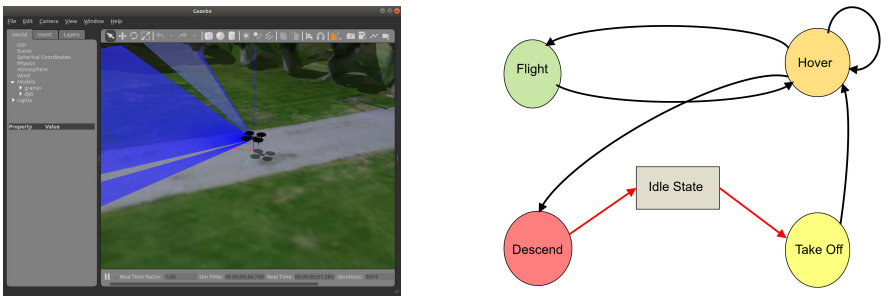
[10] `http://wiki.ros.org/dji_sdk`

```
roslaunch lrs_dji_sim sim.launch dynamics:=true mpc
    :=false pid:=false basic:=true inair:=true ns
    :=/dji0
```

Just starting the simulator, the simulated dji will fall down to -100 in z. Also, if `inair:=true` the starting yaw will be 90 degrees counter-clockwise from FLU/ENU (the vehicle points north).

- The states of the UAV are available on these ROS topics:

```
/dji0/world_position
/dji0/dji_sdk/gps_position
```



(a) **(b)**

Figure 4.6 (a) Our WARA-PS UAV DJI simulated and loaded in Gazebo. (b) In the Rendezvous control ROS node, to control the DJI M100 UAV flight in the Rendezvous landing maneuver a state machine is implemented based on its state of flight. These states include idle, takeoff, hover, flight, and descend. This is the state machine that dictates the vehicle to go to the designated target position.

Rendezvous control ROS node

For completing our mission in the Rendezvous algorithm, the position tracking controller using PID control (or possibly MPC in future work), computes the difference (the relative distance) between the UAV's current position and the target position on top of the ground robot, either hovering or landing target, to send correction commands to the quadrotor UAV to fly closer to the goal position.

Specifically, the Python script *control_rendezvous.py* creates the *rendezvous_control* node to handle the process of identifying the locations of both the DJI quadrotor and the ground robot and then publishing of their locations, as well as performing the rendezvous landing maneuver. In this node, we set up several subscribers that listen to several topics published by the *djisim_m100* node and the new created simulation model of two-wheeled robot (quadro). The input to this *rendezvous_control* node is the following topics:

`dji0/dji_sdk/gps_position`
`dji0/dji_sdk/gps_health`
`dji0/world_position`
`dji0/dji_sdk/attitude`
`quadro/odom`

which continuously generates new setpoints for the control system to attempt to achieve. Wheras, the output topic is on the flight control setpoint of this type:

`/dji0/dji_sdk/flight_control_setpoint_generic`

In the structure of the general flight control setpoint topics, axes[0] to axes[3] stores set-point data for the 2 horizontal channels X and Y, the vertical channel Z, and the yaw channel, respectively. The meaning of the set-point data will be interpreted based on the control flag which is stored in axes[4], and the control mode should be roll, pitch, thrust and yaw rate. This control flag is an *UInt8 variable* that dictates how the inputs are interpreted by the flight controller. We notice that it is the `bitwise OR` of 5 separate flags and we remember that a number in *hexadecimal notation* begins with the prefix `0x`.

In our project we choose the desired control mode as the following, such that the control flag value is to control roll, pitch, thrust and yaw rate, that is:

`roll, pitch | thrust | yaw rate | ground_ENU frame | No breaking`

After investigating the details about the flight control set-point on ROS documentation of dji_sdk [11] we can set the corresponding value of our "control mode flag" as the following:

`| 0x00 | 0x20 | 0x08 | 0x00 | 0x00 | = 0x28`

Using state machine to perform Rendezvous control

Finite-state machines are powerful mechanisms for controlling the behavior of a system, especially robotic systems [Fairchild and Harman, 2017]. ROS has implemented a state machine structure and behaviors in a Python-based library called SMACH [12]. However, for prototyping our Rendezvous control ROS node we implement our simple state machine in Python without using SMACH library.

Similar to the work in [Hoenig and Ayanian, 2017], we assume a state-based logic implemented in the iteration function `iteration_stateMach` in `control_rendezvous.py` to model the UAV basic flight motions where this controller node implemented a *state machine* to control the DJI M100 based on its state of flight, such that this process handles all of the flight command messages.

The `rendezvous_control` node has five states of flight control: idle, takeoff, hover, flight, and descend, as shown in Figure 4.6b. The variable `_dji_state` is used to indicate the current control state. For our mission, DJI M100 will be controlled to take-off, hover, fly and land.

[11] `http://wiki.ros.org/dji_sdk`
[12] `http://wiki.ros.org/smach/Tutorials`

In addition, in this Python function `iteration_stateMach`, an iterative process is implemented behaving as a *closed-loop system* that computes the difference between the positions of our two robots, and then commands the UAV quadcopter to fly in the direction of the goal position on the ground moving robot. This *iteration* continues every 20 milliseconds with a "new position" for DJI detected and a "new correction" computed and sent. In ROS programming, the class `rospy.Duration` for Python, and `ros::Duration` for C++, can be used to monitor a time difference, create timers, rates, and so on.

Using ROS Services to control take-off and landing

In this work, we choose to activate the control states of take-off and landing using ROS service calls. Within the `rendezvous_controller` node, two ROS services are created with *callback functions* to be invoked by a client when a request for the service is sent. Take-off command can be performed by sending a service command to the controller and an appropriate command will be published to UAV to take off, where this is done using `Empty` type for both services and publishing. Similarly, the same can be done for the "descend" flight state.

This type `Empty` is one of the built-in service types that it has no data fields and provided by the ROS `std_srvs` package. The services for `/dji0/takeoff` and `/dji0/descend` can be created by the following statements respectively in `control_rendezvous.py`:

```
rospy.Service("/dji0/takeoff", EmptyServiceMsg, self._Takeoff)
rospy.Service("/dji0/descend", EmptyServiceMsg, self._Descend)
```

We can also command taking-off and landing through a service call to the `drone_task_control` service with specific parameters, provided by a standard method in DJI SDK. These services for takeoff and descend can be called by the following `rosservice call` respectively

```
rosservice call /dji0/dji_sdk/drone_task_control 4
rosservice call /dji0/dji_sdk/drone_task_control 6
```

When implementing our own algorithm, it is worthwhile to know which of the *services* and *topics* that are present in the actual API from the DJI SDK, running on-board a DJI Matrice 100 platform, are also present in the DJI Simulator. This WARA-PS DJI simulator implements the following services defined by the ROS DJI API, where the `dji_sdk` package provides a ROS interface for the DJI onboard SDK and enables the users to take full control of supported platforms (DJI M100) using ROS messages and services.

```
dji_sdk/sdk_control_authority
dji_sdk/drone_task_control
dji_sdk/query_drone_version
dji_sdk/set_local_pos_ref
```

It also implements the following service specific to `djisim`, the WARA-PS simulator

```
djisim/mission_waypoint_action
djisim/reset_battery
```

4.7 Rendezvous Experiments Results: Tracking Flights and Touchdown landing

In this section, we investigate various examples of mission scenarios for simulating our Rendezvous landing algorithm in this part of the project. These simulated missions are important to evaluate the performance of the control scheme employed in our implemented autopilot and illustrate its convergence results at the touchdown landing operation at the end. Therefore, our holonomic simulated two-wheeled robot (TWR) is employed in these simulated experiments and tracked by the WARA-PS simulated UAV quadcopter, DJI Matrice 100. However, it is important to remember that just at starting the WARA-PS DJI simulator the simulated DJI quadcopter will fall down to -100 m in z-axis, as mentioned in the instructions of WARA-PS Development Environment. In fact, this issue will affect the flight performance of the DJI quadcopter in the simulated experiments.

In the following subsections, we present the simulation analysis of three different mission scenarios consisting of

- Rectilinear scenario: Moving in a straight line trajectory
- Circle pattern scenario: Moving in a circular trajectory
- Curvilinear scenario: Moving in a " $\supset$ " pattern path

In this way, the autopilot is subjected to these simulation analysis tests for debugging and evaluating the simulated behaviour of our Rendezvous algorithm. As such, we can examine the time series of ground truth of the two robots, i.e. their positions data, as well as the position errors or some other characteristics.

Yet, as mentioned previously, the goal in the Rendezvous mission is to control the DJI to zero or minimize, as much as possible, the relative distance to the ground robot, that is, to get DJI drone to hover five meters above the ground mobile robot and land on it at the end.

It is also nice to recall that in the implementation of the autopilot, we use a State Machine (State-based logic) to model the UAV basic motions (Flight command messages), i.e., it is this state machine that dictate the DJI vehicle to go to each of the designated target positions. Moreover, through an iterative process, the implemented Rendezvous controller uses the difference between the UAV's current position and the goal position (either hover or target) to send correction commands to fly closer to the goal position. This iteration continues every 20 milliseconds in which a new position for the flying DJI UAV is detected and a new correction is computed and sent.

Plotting Simulation Results using Rosbag

ROS has become a standard for the development of advanced robot systems and the community is also growing exponentially, according to [Roberto Guzman and Ariño, 2017]. `RViz` tool, the 3D Robot Visualizer in ROS, is usually used for obtaining a high-level view of sensor state in a ROS system. However, for examining individual values, it is possible to use `rqt_plot` for plotting any numeric data published in the ROS system. But `rqt_plot` does not know anything about trajectories, it just plots numeric data over time, i.e. "live plot" as in an oscilloscope instrument. Another drawback of `rqt_plot` is that with `rqt_plot` the x-axis is always the timestamp.

Therefore, to solve this problem and obtain a "static plot" based on vector of values, we need to make use of another tool called `rosbag`, using ROS *bag files*, where the `rosbag` command can be used both to record and to replay bag files. In addition, it is also useful to use `rqt_multiplot`, by running the following command in terminal `rosrun rqt_multiplot rqt_multiplot`, which is far superior to `rqt_plot` with many great features for visualizing numeric values in multiple 2D plots, as shown in our simulation results below (in Figure 4.7 for example). In addition, it is very important to use `rqt_console` by running the following command in a new terminal window `rosrun rqt_console rqt_console`, when we need to debug our ROS implementations, to show the prints in console instead of using `rospy.loginfo` on the screen, because these prints slow down the frequency. Therefore, we need to change the output from "screen" to "log" in the *launch file*. Moreover, when packages print to screen or have debug messages, they also get logged in that directory, to remove those files we need to run the command: `rosclean purge` in some terminal window, in Linux of course.

In fact, in our simulated experiments, we work with data that was recorded on our simulated robots using the topics provided by Gazebo plugins and WARA-PS simulator. The recorded bag file contains sensor measurements from among others odometry and inertial measurement unit (IMU). All these simulation data are logged using ROS bags [13], by having direct access to simulation data through listening to the corresponding ROS topics. It is worth noting that, by default, the bag files will be stored in the `.ros` folder in the user's home directory.

Finally, in this Rendezvous landing problem we need to evaluate the performance of the implemented algorithm by visualizing the data gathered from ground truth topics using `rqt` tools in ROS. Therefore, various useful `rqt_plot` nodes from a new created package `rqt_plot` are launched, as explained in the following:

- One node plots the ground two-wheeled robot TWR position data gathered by subscribing to the Gazebo plugin generated topic `quadro/odom`. As mentioned previously, this simulated TWR can be moved and controlled by running the implemented Python node `Control_quadro.py`. By the way, using

[13] `https://wiki.ros.org/rosbag`

`rqt_mulitplot` it is also possible to plot the path of the simulated robot in the x/y-plane, what we could not do it with `rqt_plot`.

- Another node plots the quadrotor UAV position data by subscribing to the provided DJI M100 position topic `/dji0/world_position`, on which the WARA-PS simulator will continuously output the local position of the simulated quadcopter, relative to the local origin. In addition, as mentioned previously, the given origin coordinates are the ones used in the Gränsö testing area, in Västervik, Kalmar county in Sweden. Here, it is useful to recall that, for upcoming commands, when we will need to have the "current position" of the quadcopter available, we can also set up another subscriber that listen to messages sent on another topic: the global GPS coordinates `/dji0/dji_sdk/gps_position` .

- Then, it is essential to visualize the relative distance (i.e., difference in positions) by plotting the *error convergence graph* of it, as illustrated in Figure 4.7. Thus, we create a publisher on the new created topic named `/diff` of type `Pose` from the `geometry_msgs` package, in which we publish the error variable that we calculate in the callback function for `/dji0/world_position`, named `world_coordinate_callback`.

In the following subsection, we explore the simulated flight experiments of the Rendezvous landing manoeuvre performed in different mission scenarios using the implemented autopilot, in which the PID is chosen as the control approach for the automatic position controller, that enables our UAV quadcopter to track a moving UGV platform and touch down on it.

For completeness, it is worthwhile to highlight that the implemented Rendezvous algorithm has proven to perform correctly in simulations under ideal or controlled conditions. For example, the aerodynamic effects have not been closely examined when controlling the landing operation of the UAV on both stationary and moving ground vehicle (UGV), especially in scenarios when this vehicle is travelling at higher speeds. In fact, the aerodynamic effects such as turbulence from the moving vehicle are indeed more complex at higher speeds, making them harder to manage. Therefore, in future work, compared to testing this algorithm in simulated experiments, using UGV as a moving landing platform for quadcopter in real-world deployment will add more complexity to the simulated system implemented in this thesis work. Of course, because dealing with this manoeuvre in a real environment would incorporate or require more challenging connectivity and interactions between the drone and the moving platform. On the other hand, the real-world Rendezvous system on a moving platform will be more flexible and helpful in applications like search-and-rescue (SAR) missions, to find and aid people in danger scenarios.

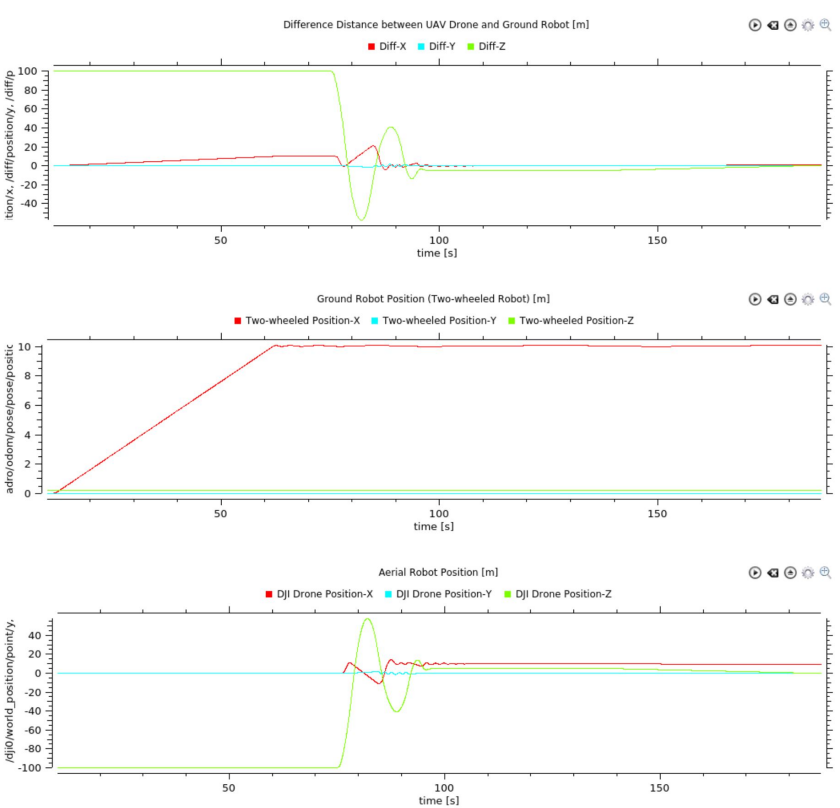


Figure 4.7 Simulation results of Rendezvous algorithm in Scenario 1: Moving in a rectilinear trajectory. Using ROS bag files and RQT tools, we visualize the position data of both UAV (lower plots) and UGV (center plots) and their relative distance (upper plots). This simulated flight experiment is completed successfully in performing the Rendezvous landing process of the quadcopter over the simulated two-wheeled robot (TWR) moving a distance of 10 m along a straight way. Here in this figure, the data plots present the landing manoeuvre performed when the ground robot is at rest, after reaching the destination position at the distance 10 m. Nevertheless, in other simulated experiments the algorithm has proven to be successful as well on a moving UGV. In fact, as illustrated in the upper plots, it is clear that the relative distance trajectories in altitude (Diff-Z), in X and Y directions as well, converge towards zero, as it is supposed to be at touchdown landing operation. Moreover, we can notice the overshoot and oscillations in the flight performance of the simulated UAV caused just after bringing it "overground" from its reference position located at coordinates $x = 0$, $y = 0$ and $z = -100$ m, as also exhibited more clearly in the lower trajectory plots. Recall, according to the WARA-PS design, when starting the DJI simulator the quadcopter should fall down to this negative starting position located at altitude $z = -100$ m. Then, this offset in z-direction, shown in the upper sub-figure in green color, becomes positive starting from $z = 100$ m according to the difference distance equation in 4.2. Anyway, such that initial -100 m offset behaviour will not exist when conducting real-world flight experiments using the real DJI M100 platform.

Rectilinear scenario: Travelling in a straight line trajectory

In this flight mission, the simulated quadrotor DJI M100 could complete successfully the Rendezvous manoeuvre, either hovering or landing, on the simulated two-wheeled robot (TWR) moving a predefined distance in a straight path. Yet, the control strategy is in general designed to enable the UAV tracking the target mobile robot on the ground and then to guarantee the safety of the touchdown manoeuvre. Therefore, the implemented Rendezvous algorithm controls the drone vehicle to take off, draw a straight line by following the ground vehicle and landing on it at the end.

More specifically, our Python code monitors the take-off operation of the UAV starting from its initial position at $(x,y,z) = (0,0,-100)$, and controls it to fly to $(x,y,z) = (x,0,5)$ for hovering first above the landing location on UGV and then for aligning with it in the xy-plane. Next, it is commanded to descend at the desired touchdown position after achieving a certain vertical velocity. Initially, this derived algorithm is basically experimented and validated in simulation on a stationary landing platform, as discussed in the following two examples. First in Figure 4.7, the simulation data plots present the landing manoeuvre performed only after reaching the destination position at the distance 10 m from the origin, i.e., on a still ground vehicle located at coordinates $(x,y,z) = (10,0,0)$ in the global frame. Whereas in Figure 4.8 we choose instead the hovering state at $(x,y,z) = (10,0,5)$ to align with UGV in the xy-plane, before a landing is possible.

Ultimately, the autopilot algorithm has been successfully tested on a travelling UGV as well through other simulated landing experiments. As such, regardless of landing on still or moving platforms, we generally consider that the implemented autonomous landing manoeuvre should work in simulation while the target ground vehicle also itself follows other trajectories travelling at a certain speed, and as well as while the simulated quadcopter satisfies the limit of certain specified threshold values on pitch, roll and vertical velocity at the touchdown operation. These threshold specifications are advantageous to keep the landing safe and/or make it more gentle, and will be necessarily useful in future real-world outdoors experiments especially when facing the challenging disturbances and ground effect at touchdown.

Moving in Pitch and Roll Directions For completeness, it is important to recall that in order to correct the relative position errors $(\Delta x, \Delta y)$ between UAV and UGV in x- and y-direction, as shown in the resulting responses in simulation plots, the controller calculates the appropriate pitch and roll angles to direct the overall lift force and obtain the required "forward thrust" to be applied to the rotorcraft in the horizontal xy-plane. In this way, the required translational velocity is generated in order to accelerate the airframe "rotationally" and enable it to approach its new goal position. In fact, as described previously in the quadcopter aerodynamic model in 6.12, the translations depend on the orientation or tilt angles (pitch and roll), such that the x-position is linked to the pitch angle, and the y-position is linked to the roll

angle. This means that to move the quadcopter in the x-y plane it must tilt (moving in the pitch- or roll-direction), which can induce oscillations in both the x and y directions, as shown in our simulation results in Figure 4.8. Yet, this indirection in controlling the quadrotor motion in the x- or y-direction is a consequence of the vehicle being an underactuated system, subject to only four inputs (one force and three moments) when a rigid body evolving in 3D Cartesian coordinates has six degrees of freedom (6 DoF).

Flight Simulations with Large Initial Distance As such, it was worthwhile to plunge closely into the dynamics of the quadrotor vehicle such that readers can get better understanding when analysing the simulations results. In conclusion, it is also nice to highlight that in all our flight scenarios we start the manoeuvre with a large relative distance in altitude Δz (initial distance) because, as discussed previously, the virtual DJI quadcopter should fall down towards the initial position $(x, y, z) = (0, 0, -100)$, every time we start the WARA-PS simulator . Therefore, because the initial relative z-position between the two virtual robots is large, the initial tracking error will be large at the start of each simulation trial. Then, as the vehicles get closer, the tracking errors for the relative x, y and z positions will be relatively small.

As such, when minimizing the errors defined as the differences between the desired and actual positions, it is important to distinguish between large-scale errors (tens of meters, like the one of the initial Δz) and small-scale errors (tens of centimetres), both of which are handled by the closed-loop system under PID control that computes the differences and drives them close to zero by commanding the UAV to fly in the direction of the target position. In such cases, it is very common that well-designed control systems can struggle to keep up with such *rapid changes* causing some oscillatory response.

Consequently, the implemented autopilot uses four independent PID controllers for x, y, z and yaw (for each of DJI four dimensions of flight control: pitch, roll, thrust and yaw) to continuously adjust the tilt force and the pitch and roll angles to keep the drone on its intended path, leading to over-correction or under-correction, which causes oscillations around the desired path. In fact, as will be discussed later, in the following initial tests results, the PID controllers result in a fast initial response to counteract the initial large error (Δz) and the rapid change in error scale, generating a strong output and putting the system into oscillation around the target trajectories, x and y coordinates over time.

Initial simulation tests Using RQT ROS visualization tool, the resulting error convergence graph along with position trajectories (1D plots showing x and y coordinates over time), for both robots DJI and TWR, are shown separately in Figure 4.7. These trajectory plots are also illustrated together in the same sub-figure with more clarity in Figure 4.8. Furthermore, plotting the path in x-y plots (2D plots),

instead of the trajectory, would give as well additional useful illustration of how the UAV and the UGV move in relation to each other during the Rendezvous simulation analysis. As such, in each simulated experiment, the corresponding path x-y plots are also obtained using RQT ROS tool and illustrated subsequently in other figures below. However, x-y-z path 3D-plots cannot be visualized with this RQT tool.

In particular, in the error convergence graph shown in the upper plots in Figure 4.8, the red and cyan plots show the relative distance between the UAV and the mobile ground robot in the x and y directions respectively. These difference trajectory plots, Diff-X and Diff-Y, show an oscillatory behaviour before converging towards zero at the end, as it is supposed to be in order to fly and align above the desired landing location (in xy-plane), before triggering the touchdown landing operation.

Whereas the green plot (Diff-Z) illustrates the relative distance in z-direction that converges towards $z = -5$ m instead and not to zero. In fact, this difference trajectory plot illustrates the UAV flight in hovering state at $z = 5$ m over the UGV. Moreover, this hovering state is expressed in negative z-direction, at $z = -5$ m in altitude, because in our implementation code the "Diff" variable, denoting the difference in positions between TWR and DJI, is formulated as in the following distance equation:

$$Diff = TWR_{position} - DJI_{position} \tag{4.2}$$

Next, the lower sub-figure shows the rectilinear route crossed by the simulated two-wheeled robot (TWR) travelling the distance of 10 m. In the same sub-figure the data displays as well the flight trajectories of the simulated DJI quadrotor, started flying immediately after setting in motion the ground robot from its initial position located at coordinates $(x, y, z) = (0, 0, 0)$ in the global frame. Practically, we notice in this experiment results that the green and purple trajectory plots of the DJI flight, in the x and y directions respectively, follow the same course as the ones related to the target ground robot in red and cyan colors, but in wavy appearance due to oscillations caused by the control effort to address the aforementioned initial large error in z-direction. At the end, such an initial state offset will not be experienced on the real UAV platform DJI M100, but only in this WARA-PS simulation environment. The plots in the z-direction are not included here in this subfigure just for better visibility of the tracking performance in x and y directions.

As such, the initial simulated experiments of UAV-UGV Rendezvous landing, as well as of hovering at $z = 5$ m over the target robot, is completed successfully as shown in the upper convergence error plot. In general, this implemented algorithm has been proven to be satisfactorily successful at controlling the UAV position under the Rendezvous manoeuvre in all the three simulation scenarios, as discussed and shown below. However, the resulting data plots still present noticeable oscillations that would be advantageous to minimize as much as possible. Hence, as explained in another subsection below, we will discuss an ulterior improvement strategy developed to decrease the oscillations of this wavy motion and achieve better flight performance.

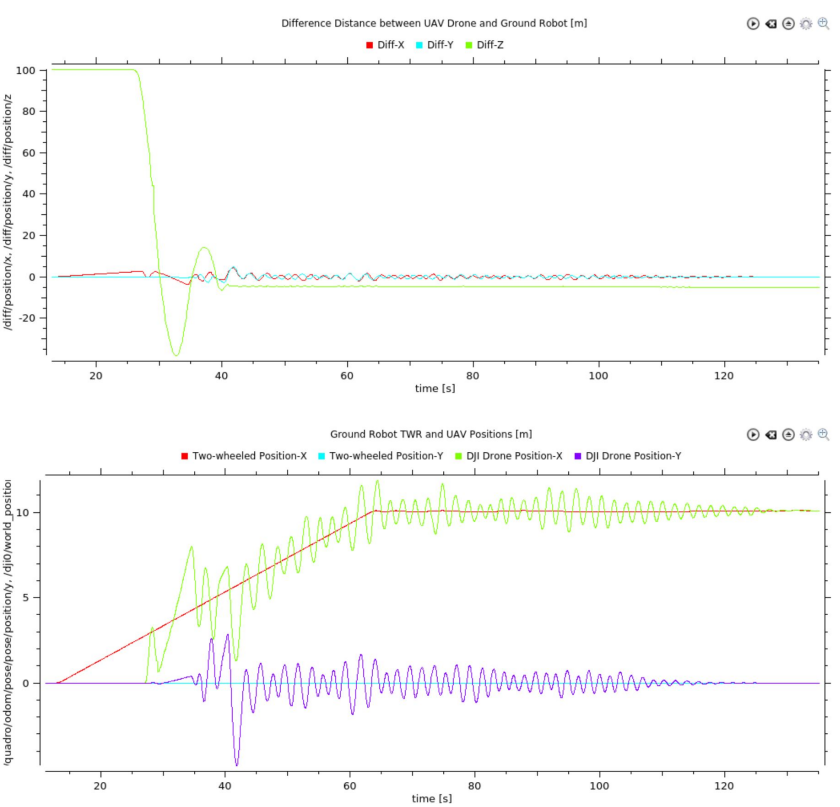


Figure 4.8 Output of RQT plots showing simulation data from one of the initial tests for hovering manoeuvre. This simulated mission is carried out successfully to perform the tracking and hovering of the quadcopter at $z = 5$ m over the simulated two-wheeled robot (TWR) moving along a straight way. The resulting plots in the lower subfigure, in fact, present the performance of the designed Rendezvous algorithm along with the PID position controllers in controlling the UAV's flight . These plots demonstrate that the quadcopter was able to track the moving ground robot, by following a very similar path to the UGV, as shown by the overlapping position trajectories (in x and y directions) of the UAV and UGV in the same graph. However, an obvious drawback is the observation of oscillatory tracking behaviour around the desired path. Here, the position trajectory z-plots for both robots are not included for clarity purposes. Furthermore, the upper sub-figure displays the relative distance (Diff-plots) in x, y and z directions between the two vehicles. This data illustrates the UAV autonomous hovering manoeuvre, where it maintains a constant altitude of five meters above the ground robot. Consequently, this distance is represented as a negative offset of $z = -5$ m in the green Diff-Z plot, consisting with the distance equation in 4.2.

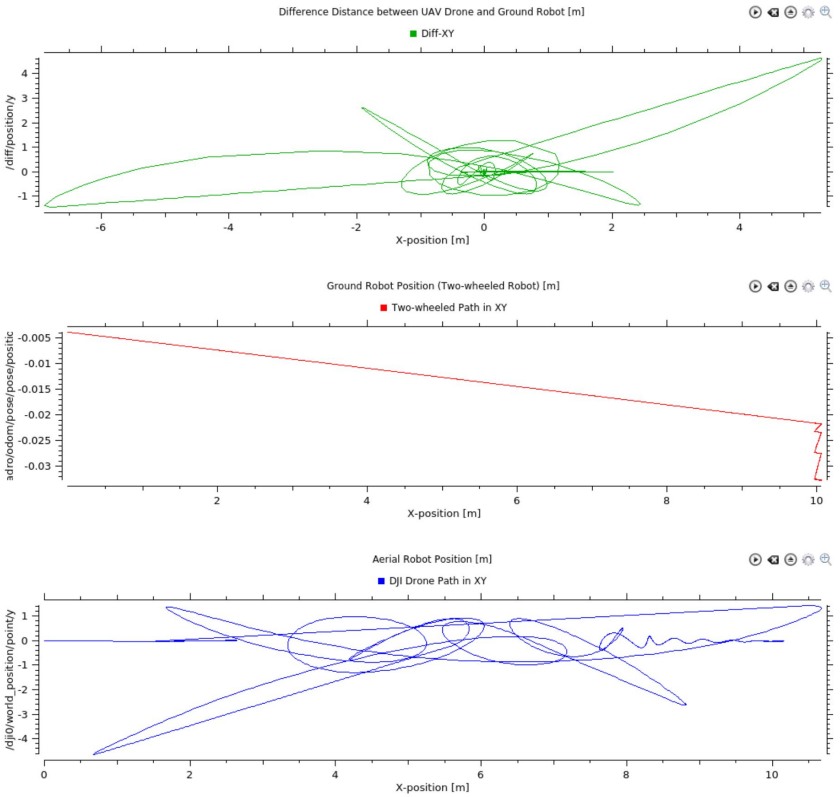


Figure 4.9 This figure displays the 2D *xy*-path of the simulation position data corresponding to the same oscillatory 1D trajectories observed during the UAV tracking and hovering manoeuvres in the rectilinear movement scenario shown above in Figure 4.8. In particular, the lower sub-figure illustrates how these 1D oscillations result in the wavy path seen in the horizontal plane.

Circle pattern scenario: Moving in circular trajectory

Here in this second mission scenario, another simulated flight test is experimented to investigate the performance of the implemented autopilot (Rendezvous algorithm and position controller replacing the teleoperating human) in controlling the UAV-UGV Rendezvous mission. This test flight was performed by following the ground robot along a circular shape trajectory. The resulting plots of the succeeded tracking and hovering manoeuvres are illustrated in Figure 4.10. Similarly to the results in the first scenario, the lower subplot displays the overlapping position trajectories of the two unmanned vehicles demonstrating that the simulated DJI quadrotor was able to follow the travelling ground robot, but with a noticeable oscillatory be-

haviour around the target path. Furthermore, UAV's autonomous hovering flight is demonstrated by keeping an altitude of five meters above the UGV, which is however illustrated by $z = -5$ m in the upper green Diff-Z plot consisting with the difference equation in 4.2.

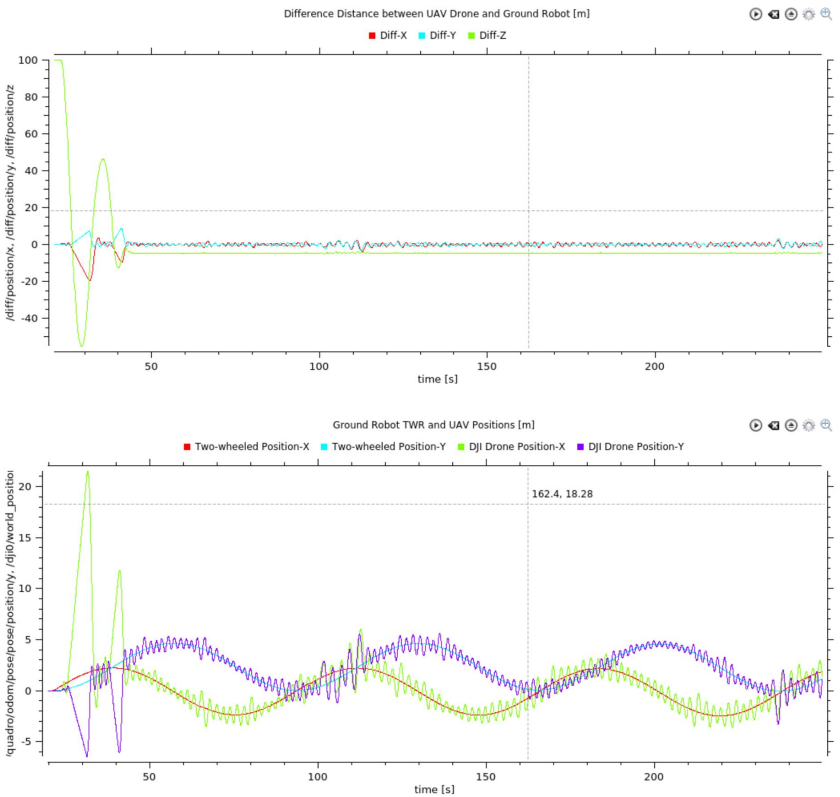


Figure 4.10 This figure presents the simulation data for the UAV-UGV tracking and hovering manoeuvres experimented in the circular pattern trajectory scenario. Also here, moving in a circular trajectory, the resulting position 1D-plots in x and y directions illustrated in the lower sub-figure, together with the upper Diff-plots, suggest a satisfactory tracking and hovering performance but with noticeable oscillations.

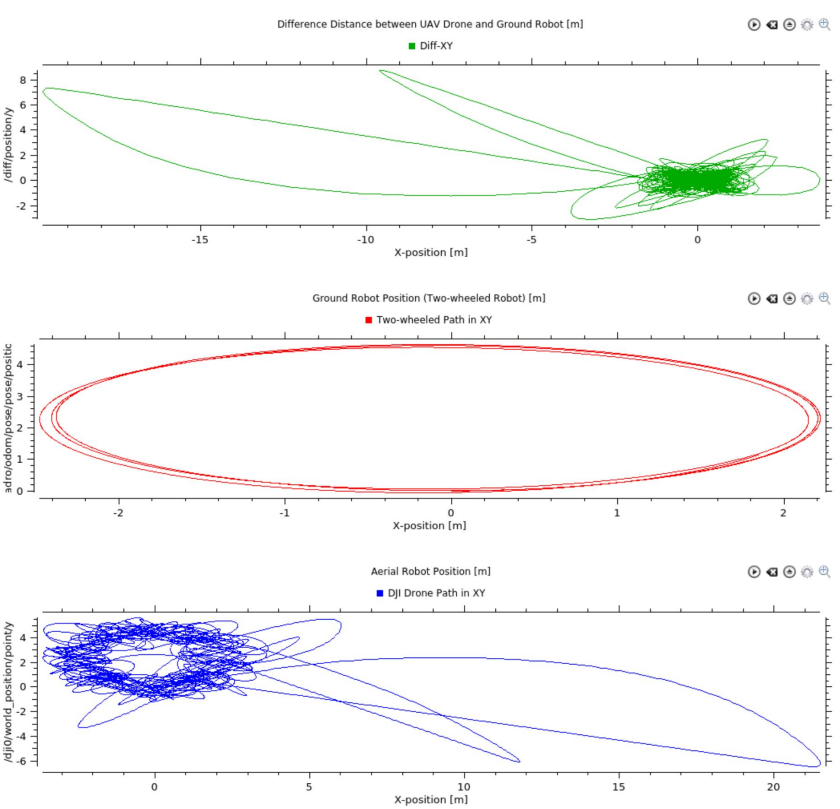


Figure 4.11 The resulting 2D plots of the simulation position data presenting the UAV and UGV *xy*-paths during the tracking and hovering experiments in the circular trajectory scenario. The lower sub-figure illustrates how the UAV's oscillatory 1D trajectories observed in Figure 4.10 are translated into a wavy circular path in the horizontal plane, which is generated by tracking the circular UGV's movement shown in the center sub-figure.

Curvilinear scenario: Moving in a " $\supset$ " pattern path

Another shape of trajectory is also designed here to verify the UAV-UGV Rendezvous implementation. This third simulated flight scenario is performed by moving the simulated TWR along a curvilinear trajectory drawing a " $\supset$ " shape, such that the ground vehicle travels ten meters ahead and then performs a turning to go back towards the origin point on the negative x-direction until reaching the position at $x = -10$ m.

As illustrated in Figure 4.12, the resulting simulation plots suggest that the quadrotor tracking and hovering tasks are also here completed successfully but with oscillatory fashion. In fact, as shown in the lower sub-figure, the two overlapping

paths in x and y directions clearly assure the tracking capability of our UAV in following the course of the UGV moving in curvilinear trajectory. Additionally, we can view the relative z-distance in the upper sub-figure converging towards $z = -5$ m, which corresponds to the UAV hovering state at the altitude of five meters above the UGV's position.

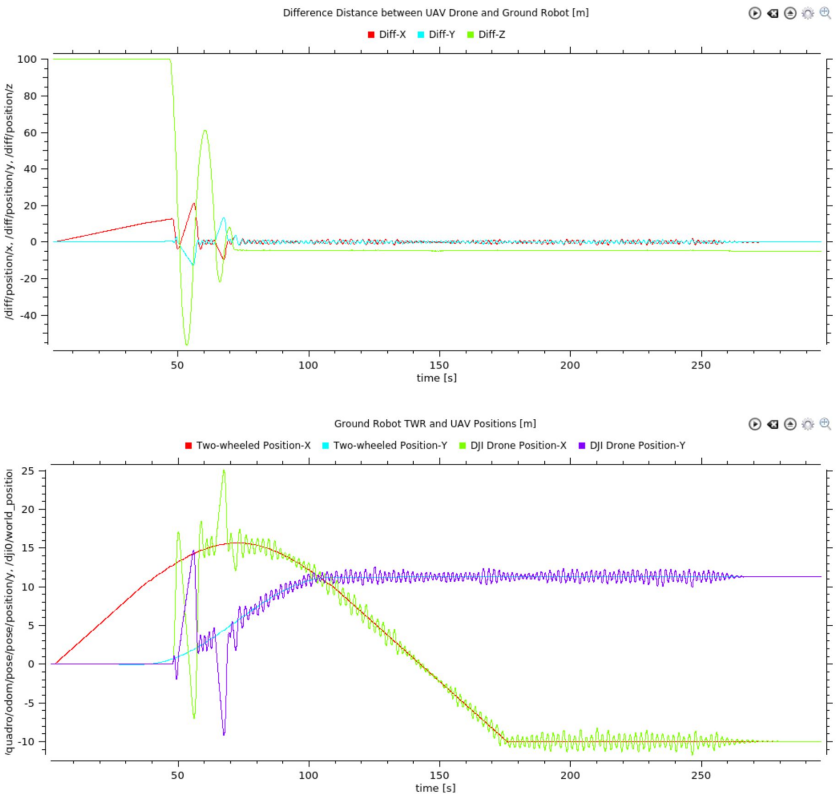


Figure 4.12 Simulation position data for the UAV-UGV tracking and hovering flight experiment in scenario 3 when travelling in a curvilinear trajectory.

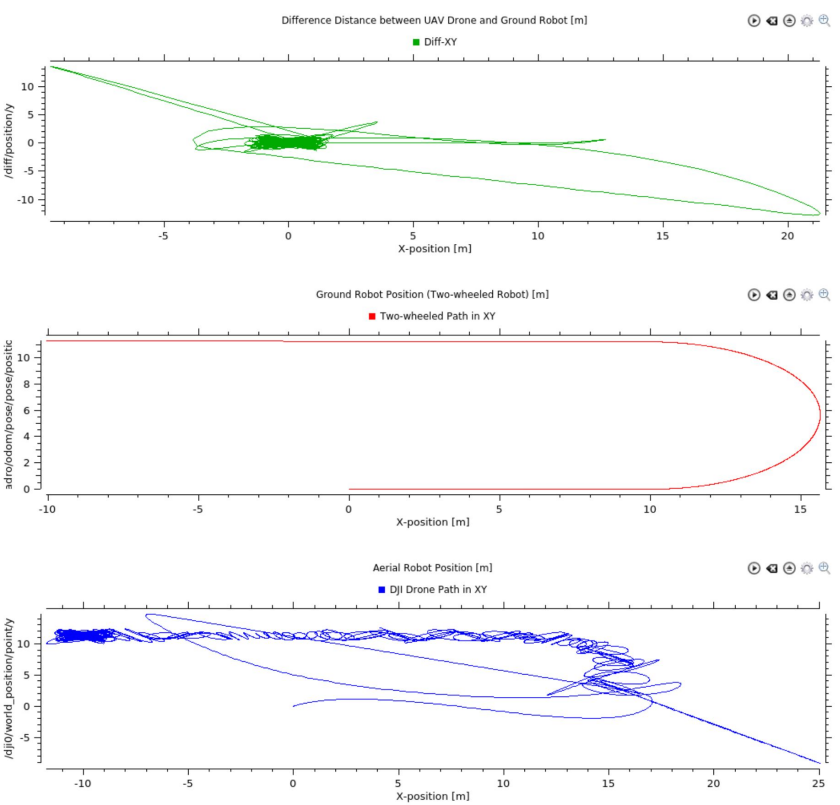


Figure 4.13 Output of RQT ROS tool displaying the *xy*-path 2D plots of the same simulation position data for the UAV and UGV in the curvilinear trajectory scenario presented in Figure 4.12. Again, also here the UAV movement in the lower graph exhibits a clear wavy oscillatory behaviour when compared to the UGV path in the center subfigure.

Final Flight Performance: Functional PID Tuning and Activating Rendezvous Controller Overground

The initial trial tests of our implemented PID based autopilot algorithm, discussed above in the experimented three flight simulation scenarios, demonstrate a satisfactory performance in controlling the UAV-UGV tracking and Rendezvous mission but with oscillatory behaviour, that we are going to address in this section.

In general, as discussed in various literature, such oscillatory motion can be caused by factors like wind disturbances, imperfections in the control design, or the quadcopter's own dynamics. Thus, to counteract these oscillations in this project, we need to investigate the PID position controller (a new PID tuning) and the UAV WARA-PS simulator itself. Wind disturbances are excluded here because at this

stage we are dealing solely with a simulated environment.

As such, in the following, we firstly reconfigure the implemented position controller to improve the performance of our Rendezvous algorithm through tweaking the appropriate PID gains. Next, we explore how the flight performance is further improved as well through implementing a suitable strategy to manage the large starting relative distance that affects the UAV flight behaviour during the tracking mission.

Damping Oscillations with Retuning the PID Loop First, for completeness, it is worthwhile to review briefly the working principles and characteristics of the *PID control loop* by showing the big picture without diving in the details that can be available from various sources online. As summarized in the block diagram [14] presented in Figure 4.14, a Proportional-Integral-Derivative (PID) control loop is a mathematical formula of which the output is a result of proportional, integral, and derivative calculations done at a *set/scan rate*, for example, once per 20 milliseconds in our implementation. This PID formula is commonly expressed by the overall control output having the proportional, integral and derivative terms on the error variable $e(t)$ as the following

$$u(t) = K_P\, e(t) + K_I \int_0^t e(t)\, dt + K_D\, \frac{d}{dt} e(t) \tag{4.3}$$

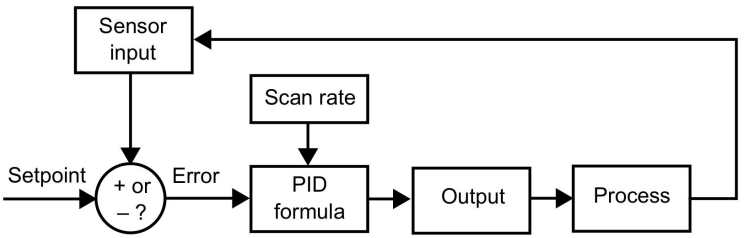


Figure 4.14 Block diagram showing broadly how the signal flows through a PID loop. At the scan time, the *sensor input* is compared to the *setpoint*, and an error (the difference between them) is produced. This error is plugged into a mathematical formula with the P, I, and D components, and the result becomes the output that is then applied to the process. The sensor input is again compared to the setpoint, and the loop repeats. (courtesy: Opto22)

The PID equation can also be expressed in the following *standard form* using the *time constants* T_I and T_D, in which K_I and K_D are respectively replaced by K_P/T_I and $T_D\, K_P$.

$$u(t) = K_P \left(e(t) + \frac{1}{T_I} \int_0^t e(t)\, dt + T_D\, \frac{d}{dt} e(t) \right) \tag{4.4}$$

[14] `http://www.opto22.com/site/pid.aspx`

where

$$T_I = \frac{K_P}{K_I} \quad \text{and} \quad T_D = \frac{K_D}{K_P} \tag{4.5}$$

in which, T_D K_P represents the time constant with which the controller will attempt to approach the setpoint. Whereas K_P/T_I determines how long the controller will tolerate the output being consistently above or below the setpoint [15].

After reviewing the PID loop background, it would be easier for readers to compare the elements of the PID theory with the aforementioned PID based autopilot of our implementations. In particular, the flight control system must continuously monitor the current lift force (trust linked to z-direction) and the current attitude angles (pitch and roll linked to x and y directions) and correct any error to keep the drone on its intended path.

Additionally, tuning is another important practice to know and understand for improving performance of the PID controller. Overall, in addition to the trial-and-error tuning practice, there are several popular methods for tuning PID loops including the "Reaction Curve Closed-Loop" (also known as the infinite oscillation and often referring to the Continuous Cycling method) and the "Reaction Curve Open-Loop" (also called step response method). These two important approaches are both used by the well-known Ziegler-Nichols classical tuning method to calculate PID controller parameters based on the system's characteristics.

In the case of our flight control in this work, in order to achieve more effective performance in our simulation results, we need to retune our PID settings of the implemented position controllers in x, y and z directions. Practically, while we were able to reduce the smaller oscillations observed previously in the initial flight performance tests, the initial starting overshoots persisted, remained at a safe level. We did try to retune the PID loop in the z-direction to soften these overshoots, but this resulted in a worse outcome and was hard to tune. Again, this starting strong oscillation is the resulting PID controller response to correct the "large initial error" (large offset) calculated to bring the drone's altitude from $z = -100$ m to ground level.

Based on my experience, I believe that tuning PID control loop is not so easy and "understanding and insight" are strongly important tools in this tuning practice. Therefore, it is technically practical to follow an overall strategy for tuning the PID loop using its proportional (P), integral (I), and derivative (D) terms to manage both large and small errors, as follows:

1. *K_P is primarily used as the first step for handling "large errors" to achieve a quick response, by providing immediate control output/response proportional to the magnitude of the current error*. However, as in our case, while higher K_P gives a stronger/faster correcting action for our large initial deviation from the setpoint, it (on the other hand) leads to oscillations and overshoot in the system, but still in a stable level.

[15] `https://en.wikipedia.org/wiki/Proportional-integral-derivative_controller`

2. *Then the integral term K_I is usually used for correcting/eliminating any residual steady-state error (cumulative error over time).* This corrective action is not needed in our problem, because our system already reaches the setpoint. Especially, when we consider no external forces (disturbances) can intervene in our simulation environment, in which no need for some I-gain to hold the drone's attitude against wind, for example.

3. *Consequently, after the proportional response is established, the derivative term K_D is used to reduce overshoots especially for "small-scale errors", by predicting and counteracting "future error", because it responds to the rate of change of the error. However, it is impractical with noisy systems, in which noise can be interpreted as a fast-changing error.* In fact, retuning the derivative term in our problem could effectively damp the oscillations for small-scale errors but it was less effective for reducing enough the strong initial overshoots caused by the large initial offset.

Improved Manoeuvre by Routing the UAV Overground As a result, after retuning the derivative term (D-gain), the oscillatory response of the quadrotor flight in our problem could be then softened, as observed in the simulation results (for the first test scenario) in Figure 4.15. At the end, D-term action works in fact as a damper to counteract the oscillations caused by an excessive P-gain, especially in small rapid changes preventing them becoming large errors. However, it had little effect on the initial overshoots, at the beginning of the plots, caused by counteracting the starting large setpoint deviation (the initial tracking error).

Therefore, we solve this issue by preventing such a large initial relative distance (considered as large-scale error) from being handled by the PID controllers. Instead, we minimize the initial tracking error and make it of small-scale errors by pulling back the drone from "the underground structure" at altitude $z = -100$, in order to be then addressed appropriately by the PID loop. In this improvement strategy we direct the Rendezvous landing controller to be activated only overground, that is, after routing the UAV in the WARA-PS simulator to a new starting point over the ground surface, namely somewhere at positive altitude near $z = 0$.

Final Simulation Results In conclusion, some significant improvement was obtained compared to the first oscillatory behaviour. One resulting path plot (in *xy*-plane) obtained by this updated rendezvous maneuver is illustrated in Figure 4.17 for the rectilinear scenario, in which we can notice the improved behaviour compared to the corresponding oscillatory one in Figure 4.9. As such, the retuned PID controllers could provide better performance with less oscillations and ensure that the basic control logic is sound.

Overall, we can conclude, through experimental simulations data and graphical analysis, that our UAV quadcopter is flying well, such that the implemented autopilot algorithm runs successfully and is able to safely complete the UAV-UGV

Rendezvous landing manoeuvre in simulated environment. In fact, the following responses in simulation plots show that the Rendezvous controller works as it should and makes it possible to start the manoeuvre from large distance initial state. A clear damping of the wavy behavior (with oscillations) can be observed in the UAV movement shown in the lower graphs shown in Figure 4.15 and Figure 4.16, contrasting with the plots of the initial tests shown in Figure 4.7 and Figure 4.8.

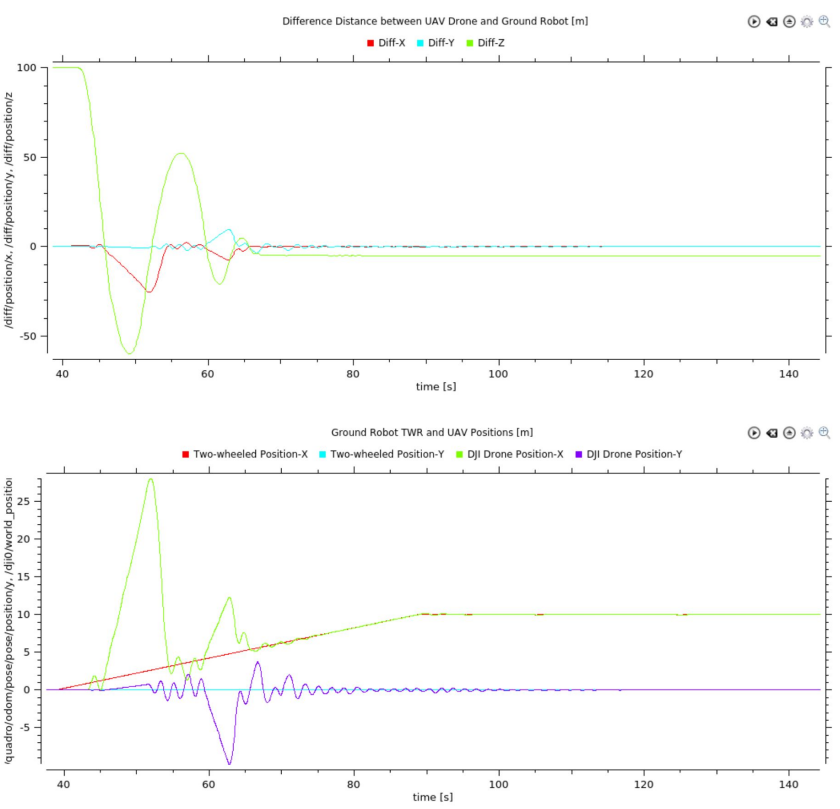


Figure 4.15 Simulation results after retuning PID Control loop with the derivative action in scenario 1, moving in a rectilinear trajectory. Comparing to Figure 4.8, we can observe how we could noticeably minimize and soften the oscillations of the trajectories after retuning the D-term of PID controllers in x, y and z directions. But it had little effect on the initial overshoots or the non-damped oscillations at the beginning of the plots, caused by the starting large setpoint deviation (the initial relative distance from negative altitude $z = -100$ m).

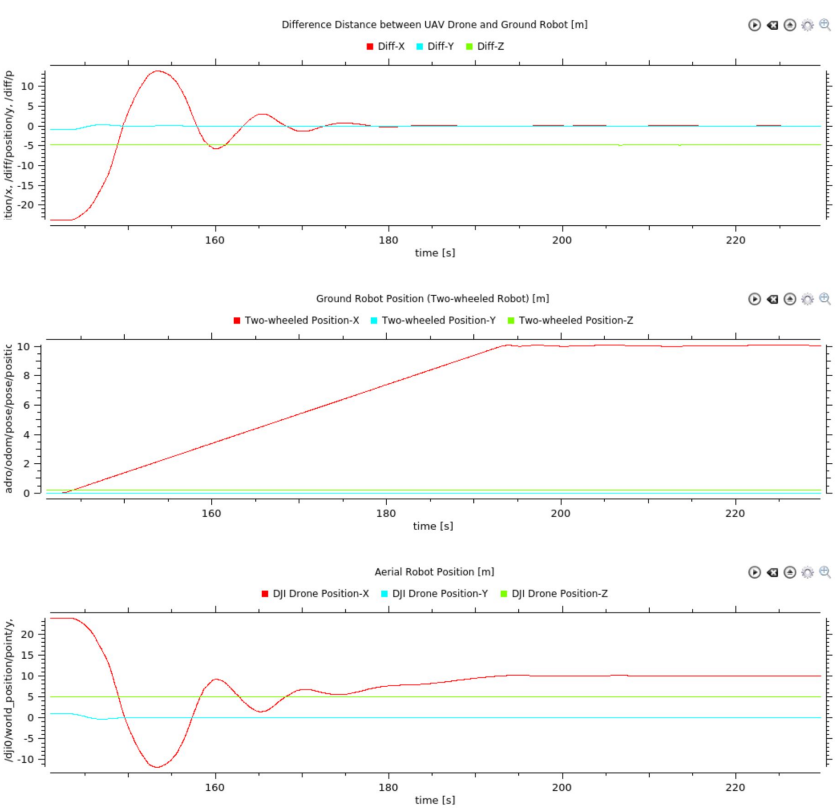


Figure 4.16 Improved 1D trajectories of the simulated UAV-UGV Rendezvous manoeuvre in scenario 1 after conducting the improving strategy, in which the Rendezvous tracking controller is activated overground after pulling back the drone from "the underground structure" at altitude $z = -100$. A clear decreasing of the oscillations can be observed in the UAV movement shown in the lower graph, contrasting with the behaviour in the initial simulation tests in Figure 4.7 and Figure 4.8.

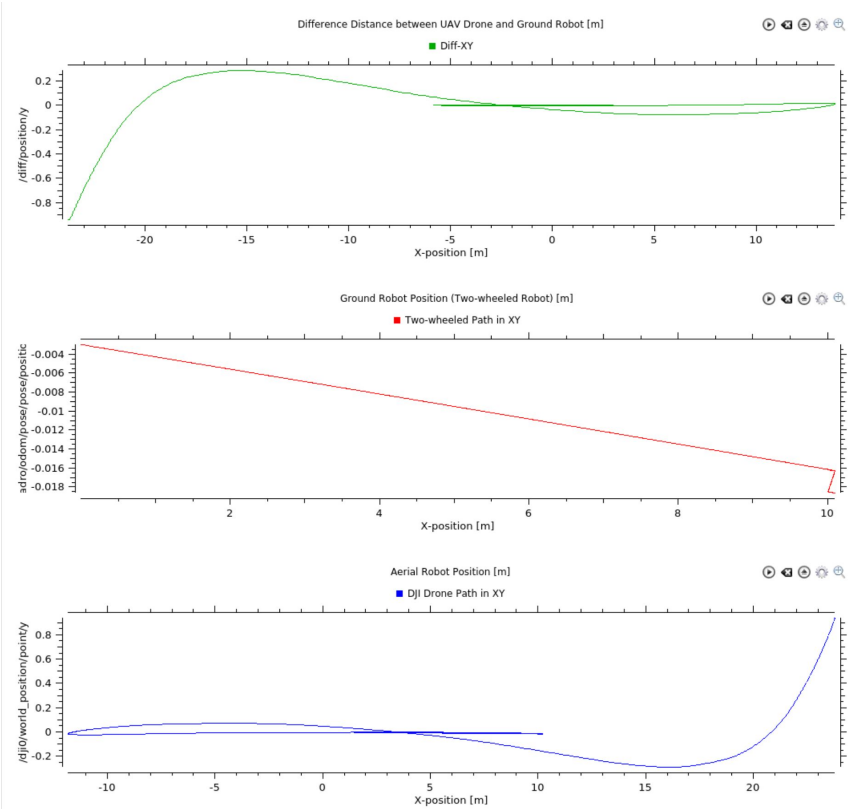


Figure 4.17 Improved *xy*-path (2D-plots) for the UAV-UGV Rendezvous manoeuvre in scenario 1, moving in a rectilinear trajectory. Comparing to Figure 4.9, we can see how we could noticeably minimize the oscillations and improve the flight performance, after bringing the UAV over the ground surface, to a position close to zero altitude $z = 0$, in which the Rendezvous tracking controller is then activated.

Concluding Remarks

From the results illustrated in figures above, we can conclude that the developed Rendezvous algorithm, using PID as a position tracking controller, performs a satisfactory behaviour and has desirable closed-loop properties in simulation.

The other simulated motivating examples are performed to investigate the versatility of this designed algorithm in simulation, including scenarios in which the quadrotor UAV tracks the ground robot *moving in a circle*, as well as by *performing a turning curve to return back*. Another scenario could also be, for example, by varying the initial direction.

The simulation in this thesis is used to assess the quality of the implemented

non-cooperative Rendezvous landing algorithm in a relatively idealized environment. These simulated experiments was fundamental to draw general statements and generate a comprehensive overview of the performance of the closed-loop system by proving properties like stability and convergence. In fact, Simulation was really ideal for gathering the data used in analysing this developed system.

Of course some degree of deviation from the true system we will see when applying this successful results and solution itself, in future work, on physical experiments in real-world settings. Therefore, these simulations will need to be complemented with more aggressive experiments and testing to increase the external validity of this solution and illustrate the applicability of our results in the real world. Furthermore, bringing this simulated solutions from theory to practice will highly likely necessitate tuning or even calculating certain suitable control parameter values.

Another context where the rendezvous problem appears frequently is for various vehicle manoeuvres, such as spacecraft docking, aerial refueling or traffic manoeuvre coordination. In these settings, focus is on the *interaction* itself, and not the *distributed properties* of the system. One way of solving these problems is to use *optimization based control* [Persson et al., 2017], like Model Predictive Control (MPC).

Thus, in future work, in addition to demonstrate the real-world performance of this simulated Rendezvous algorithm on real quadcopter hardware, it would be fruitful as well to increase robustness by implementing the promising optimization based MPC approach with different flavours as described in chapter 6.

5

Physical Modelling and Robotic Platforms: Spot and DJI Matrice 100

For the completion of this project work, RobotLab at Lund university (LTH) provides the quadruped mobile robot from Boston Dynamics known as Spot, as well as the common quadrotor UAV drones DJI Matrix 100 borrowed from Linköping University (LiU). In addition, we have access to the ROS-based software WARA-PS framework portal from AIICS at Linköping University (LiU). In section 9.1 and section 9.2, we are going to give an overview describing briefly these two platforms deployed in our project.

At first, a model of a system is a fundamental tool in any project. Modelling is necessary to help with the development of the Rendezvous landing algorithm when using model-based approaches like MPC. If the real process is described accurately by the resulting model, then it is likely that the control strategy works well on the real process.

Our Rendezvous landing problem belongs to the field of *multi-agent networked control* of which the dynamics model are usually considered relatively simple and the focus is put on how the system network properties affect the convergence performance. Before discussing the model of our multi-robot system in this project, it is worthwhile to present the main characteristics of this Rendezvous problem by the following :

- UAV quadrotor will track Spot mobile robot and land on the desired position on the landing platform on top of this Spot, using a suitable control approach (PID or MPC).

- Design information/parameters of the robots is protected, and cannot be shared with customers.

- No prior model is available, and we don't have access to the technical details that lay at the low level in Spot and DJI M100 platforms. The actuators and underlying control systems are bespoke and proprietary.

- DJI Drone and Spot mobility is externally controlled exclusively by trajectory and velocity commands.

Thus, it is not possible to obtain the system's transfer function analytically, but we need to use some usual approximation and/or simplification methods to obtain it.

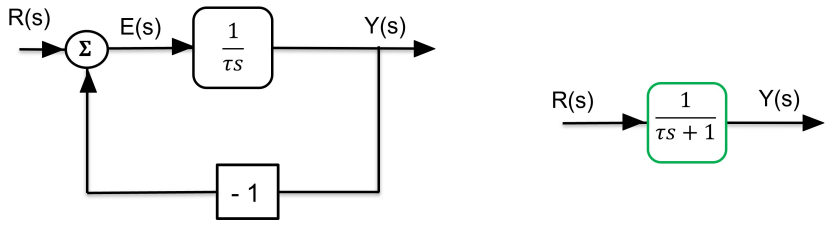
5.1 Process approximation: First-Order System Model

Since design information/parameters cannot be available for us, it is not possible to use the classic Newton-Euler methods to find the necessary kinematic/dynamic model of the robots involved in the project. In order to tackle this issue, a practical approximation technique, based on low-pass filter (LPF), is usually utilized. In summary, the control closed-loop dynamics of the robotic system is simply modelled as a **first-order system**.

NOTE Approximation of the controlled process based on **pure integration** might introduce a drift or saturation problem of the estimated values. Therefore, applying a first-order LPF to replace the ideal integrator is a common "smoothing" technique. In fact, first-order LPF filter will behave almost like an integrator for fundamental frequency signals. Therefore, it is worth to analyse/investigate the amplitude and phase, perhaps there can be significant amplitude attenuation and small phase error.

Method: First-order system model By definition, first-order systems are systems whose input-output relationship is a first-order **differential equation**, and has only one pole. This system is commonly represented by the following differential equation in 5.1 and represented by the block diagram shown in Figure 5.1a

$$\tau \frac{dy}{dt} + y = Kr \tag{5.1}$$



(a) **(b)**

Figure 5.1 (a) Block Diagram of a first-order system. (b) Equivalent simplified block diagram of a first-order system.

By computing the Laplace transform of the equation above, we get the transfer function of the first-order system, of which the block diagram is shown in Figure 5.1b

$$\tau s Y(s) + Y(s) = KR(s) \implies G_{LPF}(s) = \frac{Y(s)}{R(s)} = K\frac{1}{\tau s + 1} \tag{5.2}$$

where K is the "DC Gain" and τ is the "time constant" of the system. It is worth recalling that

- *Time constant* is a measure of how quickly a first-order system response to a unit step input.
- *DC gain* of the system is the ratio between the steady state value of output and the input signal.
- The obtained transfer function in 5.2 represents a first-order *low-pass filter*.
- *Smoothing*: In this way, any possible output "fluctuation" can be smoothed/decreased as the time constant of low-pass filter increases.
- *Parameters identification*: With a "step input" we can measure the time constant τ and the steady-state value K, from which the transfer function can be obtained/estimated.

This class of systems are extremely important because they can represent many practical applications like, for example, the mass-damper system and the mass heating system. Sometimes, to a reasonable degree of accuracy, higher order systems can also be approximated as first-order systems.

5.2 Simplified Model: Quadruped UGV Spot Robot

Since it is not possible to extract a kinodynamic model for the robot Spot by using Newton-Euler methods, we are going to follow the first-order system approach

mentioned above. Thus, the dynamic behaviour of Spot robot, i.e. the control closed-loop dynamics, can be approximated as a first order system, as illustrated in Figure 5.2a, where the input R denotes the Cartesian velocity reference.

In this way, a first order system would be sufficient to reproduce the dynamics between the Cartesian velocity reference/setpoint v_r and its true velocity v. As such, *fast closed-loop dynamics*, from v_r to v, would yield the desired approximation $v \approx v_r$.

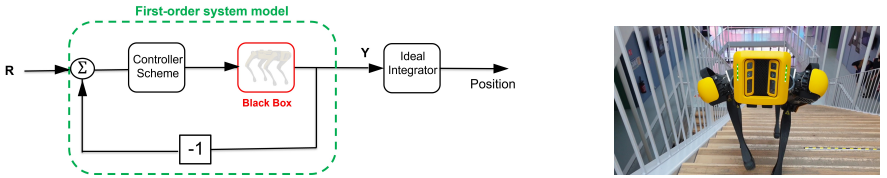


Figure 5.2 (a) Block diagram for the control closed-loop dynamics, of the quadruped robot Spot, approximated by a first-order system model. In this way, Spot system behaves as a "quasi double integrator" from velocity reference to position. That is, we need to add an ideal integrator to LPF in order to get the position. (b) The agile four-legged mobile robot Spot going the stairs.

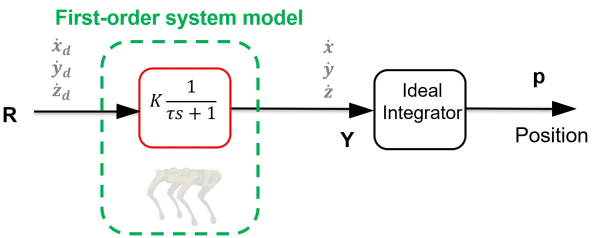


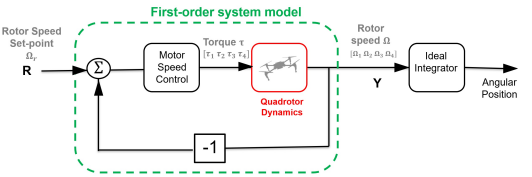
Figure 5.3 Schematic view of a first-order low-pass filter approximation, where R denotes the Cartesian velocity reference. The resulting transfer function describes the relationship between output and input.

It is nice to recall that the low-pass filter let pass the lower components of the frequency of velocity. Consequently, it is easy to investigate that the velocity fluctuation decreases as the time constant of low-pass filter increases.

5.3 Modelling of the Quadrotor Platform: DJI Matrice 100

In this project, the UAV used is a vertical taking-off and landing (VTOL) platform of the type "DJI Matrice 100", a common commercial quadcopter research platform.

Again, Since the underlying design parameters of the low-level flight controller, already provided by the quadcopter manufacturer, are not disclosed to the public, a simplified model could be estimated. In principal, the control closed-loop dynamics of the quadrotor can be also modelled as a first-order system, in the same fashion we followed above in order to approximate a model for Spot robot. A block diagram for the quadrotor is illustrated in Figure 5.4a.



(a)

(b)

Figure 5.4 (a) Block diagram for the quadrotor DJIM100: A possible approximation of the control closed-loop dynamics that will be assumed to behave as a first-order system to sufficiently reproduce the dynamics between the setpoint values of the speeds of the 4 motors $\Omega = (\Omega_1, \Omega_2, \Omega_3, \Omega_4)$ and its true ones. (b) Similar DJI Matrice 100 usually used in WARA-PS to collect data and perform experiments in the field at KTH campus. (Courtesy: [Bereza et al., 2020])

However, when using MPC controller to drive the system to Rendezvous state, we will consider the common *quadrotor system model* adapted from various research works at ETH Zurich and well reviewed in [Bouabdallah, 2007], [Blösch et al., 2010] [Bloesch et al., 2010] and [Kamel et al., 2017b]. Similar dynamical model structure is also used in the WARA-PS simulator to control the quadrotor UAV (DJI Matice 100) in trajectory tracking using ROS, based on the ETH work in [Kamel et al., 2017b].

In particular, as shown in [Kamel et al., 2017b], we assume that the dynamics of the firmware *internal low-level attitude* controller, already provided by the quadcopter manufacturer, to behave as a first-order system and are included in the multi-rotor system model, as illustrated in Figure 5.5. This low-level controller will track the reference/desired roll and pitch angles (ϕ_d, θ_d) with first order behaviour.

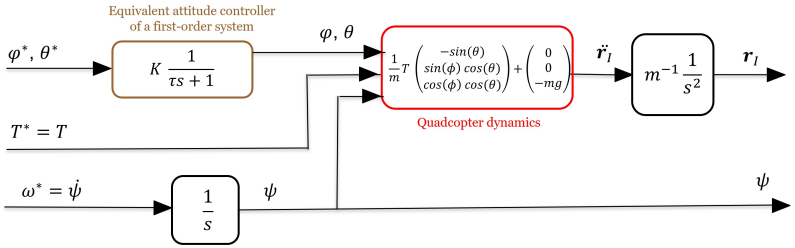


Figure 5.5 Model of the entire quadrotor platform with the roll angle ϕ^*, the pitch angle θ^*, the total thrust T^* and the yaw rate ω^* as inputs. The outputs are the position of the quadcopter r_I and the yaw angle ψ. The dynamics of the internal **low-level attitude controller** are assumed to behave as a first-order system to sufficiently reproduce the dynamics between the setpoint values and its true ones. In principal, attitude and position of the quadrotor can be controlled to desired values by changing the speeds of the 4 motors $\Omega = (\Omega_1, \Omega_2, \Omega_3, \Omega_4)$. Note that time delay, external disturbances and noise are not included in the model sketch.

In summary, the ordinary differential equations (ODE) in 5.3 and 5.4 representing the nonlinear dynamics of the quadrotor can be partitioned into two subsystems. That is, the model can naturally be partitioned into one dynamics for translational coordinates $\xi = (x, y, z)$ denoting the position of the center of mass of the quadrotor relative the inertial (fixed) frame, and another dynamics for rotational coordinates $\eta = (\phi, \theta, \psi)$ representing the orientation of the quadrotor. In fact, the second subsystem in 5.4 describes the inner-loop attitude dynamics. The overall system model is expressed by the following equations:

$$\dot{p} = v(t) \tag{5.3a}$$

$$m\dot{v}(t) = T \begin{pmatrix} -\sin(\theta) \\ \sin(\phi)\cos(\theta) \\ \cos(\phi)\cos(\theta) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} - Av(t) + d(t) \tag{5.3b}$$

$$\dot{\phi}(t) = \frac{1}{\tau_\phi}(K_\phi \phi_d(t) - \phi(t)) \tag{5.4a}$$

$$\dot{\theta}(t) = \frac{1}{\tau_\theta}(K_\theta \theta_d(t) - \theta(t)) \tag{5.4b}$$

$$\dot{\psi}(t) = \omega_d \tag{5.4c}$$

where the vehicle configuration is described by the position $p = (x, y, z)$ and velocity v of the UAV center of mass (CoM) in the global (inertial) frame and by the attitude vector (ϕ, θ, ψ), i.e., the vehicle orientation $R_{IB} \in SO(3)$. Additionally, it is common to extend the model with the aerodynamic drag and external disturbance (wind) by adding the drag coefficients matrix A and d, respectively, as shown in Equation 5.3.

The first-order approximation that describes the inner-loop attitude dynamics in 5.4 would provide sufficient information to the Rendezvous landing controller when using high-level MPC approach to take into account the low-level controller behavior. This is crucial in order to achieve an accurate trajectory or position tracking.

Note that the vehicle heading/yaw angular rate ψ of the vehicle is controlled such that it is approximately equal to a desired angular velocity $\psi = \omega_d$, i.e., it is assumed to track the commanded angular velocity instantaneously. This assumption is reasonable because the UAV heading angle has no effect on the UAV position [Kamel et al., 2017a].

It is important to know that ETH Zurich software packages for the quadrotor DJI Matrix 100 including modified SDK, linear MPC and system identification tools and their documentation are available to the community on ETHZ ASL on Github [1].

[1] `https://github.com/ethz-asl/mav_dji_ros_interface`

6

Rendezvous using MPC: Theoretical Design for Future Work

In this section, we present the necessary background to understand the basic ideas of Model Predictive Control (MPC) framework, that we will apply as a more advanced control approach in our Rendezvous landing problem. We then propose a general theoretical development of a non-linear MPC controller used for position tracking of the ground mobile robot (UGV) by the quadrotor UAV (DJI Matrice 100) deployed in the implementation of this Rendezvous landing algorithm. As a simpler start in the topic, this algorithm has been previously implemented using PID approach and described in section 4.6. The implementations and practical experiments of this second approach will be considered in future work.

After reading several related works, we believe that MPC framework would be more favourable control strategy to the Rendezvous landing problem, compared to PID already developed in a previous section. Thus, it raises the relevant question *why MPC is more suitable to Rendezvous landings*. To answer this question, we will briefly discuss some important issues in the rest of this chapter.

Model Predictive Control (MPC) is an optimization-based control technique, where at every sampling instant a finite-horizon *optimal control problem* (OCP) is solved. As stated in [Persson and Wahlberg, 2019], there are several reasons for why MPC is a suitable controller for the landing manoeuvre:

- Because of the way that *future trajectories* are predicted, potential *unwanted situations* can often be avoided before they occur if the event is within the planning horizon.

- MPC also has a natural way of dealing with *constraints* (such as actuator saturations) directly in the computation of the control inputs. As such, no "ad hoc fix" is needed and the controller will always return *feasible control*

inputs. Additional constraints, for increasing the safety for example, can also be imposed, making the controller safe by construction.

On the other hand, one of the potential issues that is often pointed out when using MPC is that the required *computational load* of solving finite-horizon *optimal control problems* (OCP) at a high rate can be very large. However, the combination of modern hardware and recent algorithmic developments have made it possible to implement MPC also on systems that require very fast updates.

Furthermore, in a *model-based control* approach such as MPC, having an accurate prediction model of the system dynamics is essential for how well the controller will perform. In MPC, an inaccurate model due to either *modeling errors* or *external disturbances* can make it difficult to achieve the desired control performance for the closed-loop system.

Specifically, if there are substantial errors in the model, then at each time step, the predicted trajectories and control inputs will be wrong, which might result in *steady-state errors* or even instabilities. To robustly handle a larger set of scenarios, the prediction needs to take these effects into account. In the following we outline several different methods developed for handling robustness issues in MPC as explained in [Persson and Wahlberg, 2019; Kamel et al., 2017b; Kamel et al., 2017a]:

- *Offset free MPC* is one flavor of model predictive control where external disturbances and modeling errors can be taken into account in the optimization, by including a *disturbance term* in the system dynamics. In [Kamel et al., 2017b], the results are also extended to nonlinear MPC.
- In [Kamel et al., 2017a], a linear and a nonlinear MPC for quadcopter trajectory tracking under external disturbances using offset-free methods are compared. The results show that the controllers have comparable behavior but that the disturbance rejection capabilities are slightly better for the nonlinear MPC.
- *Explicit MPC* is one such method where the optimization problem is computed *off-line* for the operating conditions of interest, turning the online optimization into a *function evaluation*. This method is mainly applicable to small problems due to the exponential growth of the dimension of the constraints.

6.1 Background: MPC scheme, Receding Horizon Control

Receding horizon control (RHC) is another term often used alternatively to model predictive control (MPC). This model and optimization based control offers a straightforward method for designing *feedback controllers* that deliver good performance while respecting complex *constraints*. A designer specifies the RHC controller by specifying the objective, constraints, prediction method, and horizon, each

of which has a natural choice suggested directly by the application [Mattingley et al., 2011].

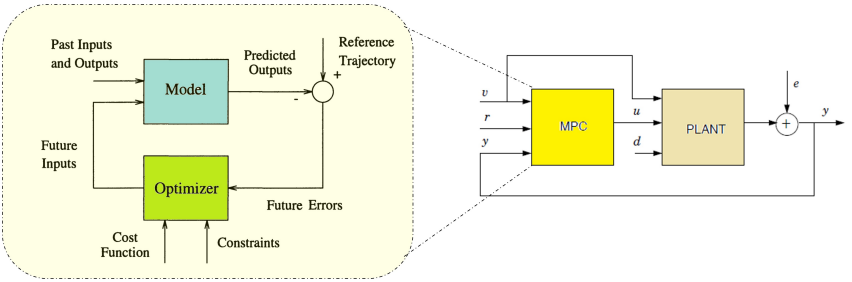


Figure 6.1 General schematic of a process controlled by MPC approach, where v,d and e denote the disturbances and r is the reference value. In the MPC basic structure in yellow, the idea of model predictive control, linear or nonlinear, is to utilize a model of the process in order to predict and optimize the future system behavior, as a function of possible optimized control actions.

As illustrated in Figure 6.1, the first key feature of MPC framework is to utilize a *model of the process* in order to predict and optimize the *future system behavior* as a function of possible optimized control actions. Moreover, by explicitly taking the *dynamics* into account in this model-based control scheme, we can choose the inputs to avoid *overshoots* and *constraints violations*. The controller should also take into account potential *disturbances* to some degree [Furrer et al., 2016].

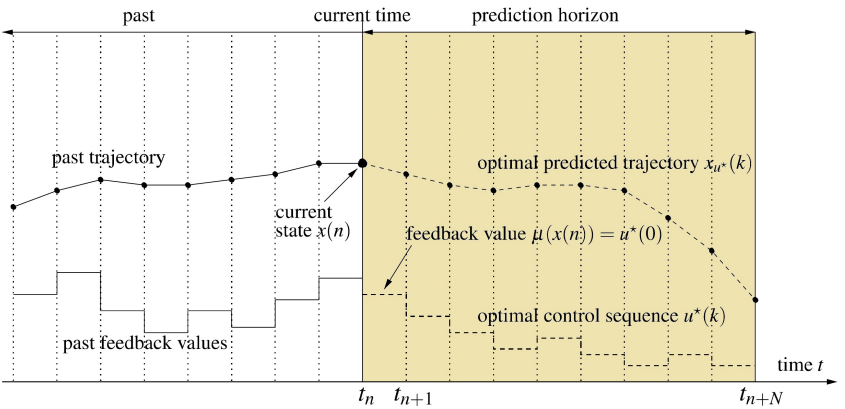


Figure 6.2 Illustration of the model predictive control (MPC) with one prediction step at time t_n: at each sampling instant we optimize the predicted future behavior of the system over a finite time horizon for $k = 0, \cdots, N-1$ of length $N > 2$ and use the "first element" of the resulting optimal control sequence as a feedback control value for the next sampling interval [Grune and Pannek, 2011].

Furthermore, *moving horizon* is the second key feature of model predictive control (MPC). An illustrative picture of one prediction step at time t_n in the model predictive control (MPC) is shown in Figure 6.2 and can be summarized by the following, as discussed in [Grune and Pannek, 2011]:

- At each sampling instant we optimize the *predicted future behavior* of the system over a finite time horizon for $k = 0, \cdots, N-1$ of length $N > 2$ and use the "first element" of the resulting *optimal control sequence* as a *feedback control value* for the next sampling interval.

- From the *prediction horizon* point of view, proceeding the iterative way, the *trajectories* $x_u(k)$ for $k = 0, \cdots, N$ defined iteratively via

$$x_u(k+1) = f(x_u(k), u(k)) \quad \text{where} \quad x_u(0) = x_0 \tag{6.1}$$

  provide a prediction on the discrete interval $t_n, \cdots, t_{n+N}$ at time t_n, and the same on the interval $t_{n+1}, \cdots, t_{n+1+N}$ at time t_{n+1}, and so on *by iterating* (moving) one step ahead. Hence, *the prediction horizon is moving* and this second feature (moving horizon) explains the reason of calling MPC alternatively with the other term, i.e., Receding Horizon Control (RHC).

As mentioned in various literature, like the works in [Kamel et al., 2017b; Sa et al., 2017; Kamel et al., 2017a], aerial vehicles behavior is better described by a set of *nonlinear differential equations* to capture the aerodynamic and coupling effects. Therefore, in the next section, a continuous time Nonlinear Model Predictive Control (NMPC) controller will be presented in similar fashion as in [Kamel et al., 2017b; Kamel et al., 2017a], in which the quadrotor system dynamics will be reviewed for completeness and clarity purpose. Next, we formulate the corresponding *Optimal Control Poblem* (OCP) and then we outline the computation techniques employed to generate the nonlinear solver, using the open source solver *ACADO Toolkit* [1] as discussed in [Kamel et al., 2017b], that is often used to solve the OCP and achieve the real-time implementation.

6.2 Nonlinear MPC Controller Formulation

In terms of autonomous landing problem, numerical papers and thesis ([Hartley, 2015], [Wenzel et al., 2011], ["Optimal UAV rendezvous on a UGV" 2016], [Persson et al., 2017], [Persson and Wahlberg, 2019], [Persson, 2021]) treated this type of problems and the majority used the promising Model Predictive Control (MPC), which is considered as an advanced *optimal control* strategy. In this way, UAV-UGV rendezvous landing problem (of unmanned aerial and ground vehicles) will be formulated as an on-line *constrained optimization problem*, as shown in Figure 6.1.

[1] `http://acado.github.io`

In the typical form of receding horizon control (RHC), for the continuous specification, the control objective might be to optimize according to a *cost function* J that is written as a finite horizon cost of the form [Åström and Murray, 2020; Murray, 2022]

$$J = \int_{t_i}^{t_i+T} L(x,u)\, dt \; + \; V(x(t_i+T)) \tag{6.2}$$

where the problem that we solve at each time step t_i is a constrained, optimal trajectory generation problem of the form $u_{[t_i,t_i+\Delta T]} = \arg\min J(x,u)$

$$u_{[t_i,t_i+\Delta T]} = \arg\min \int_{t_i}^{t_i+T} L(x,u)\, dt \; + \; V(x(t_i+T)) \tag{6.3a}$$

$$\text{subject to} \quad \dot{x} = f(x,u) \tag{6.3b}$$

$$x(t_i) = \text{current state} \tag{6.3c}$$

$$g_j(x,u) \leq 0, \quad j = 1,\ldots,r \tag{6.3d}$$

$$\psi_k(x(t_i+T)) = 0, \quad k = 1,\ldots,q. \tag{6.3e}$$

where $L(x,u)$ represents the *integrated cost* along the trajectory over a *fixed horizon* T along with a *terminal cost* $V(x(t+T))$ (end-of-horizon), where V is an appropriate positive function and it should be small near the final operating point that we seek to reach. In addition, we allow for the possibility of trajectory constraints on the states x and inputs u given by a set of functions $g_j(x,u)$ and a set of terminal constraints given by $\psi_k(x(t+T))$.

In summary, *three key ingredients are used in Model Predictive Control (MPC) design: the prediction model $\dot{x} = f(x,u)$, the constraints $g_j(x,u)$ and the cost function $J(x,u)$*. Specifically, the MPC problem setup is usually expressed in terms of optimal control problem (OCP), where the most frequently used *cost function* (stage and terminal costs) in MPC is a "weighted quadratic function" of the error between the *predicted* state and input and their respective *steady-state targets* at current and future sampling instants, weighted by the appropriately sized matrices Q_x, R_u, and P. As such, the optimal control problem (OCP) can be formulated as follows

$$\min_{U} = \int_{t=0}^{T} \left(\Delta x^T \; Q_x \, \Delta x \; + \; \Delta u^T \; R_u \, \Delta u\right) \; dt \; + \; \Delta x^T(T) \; P \, \Delta x(T) \tag{6.4a}$$

$$\text{subject to} \quad x(0) = x(t_0) \tag{6.4b}$$

$$\dot{x} = f(x,u) \tag{6.4c}$$

$$u(t) \in \mathbb{U}_C \tag{6.4d}$$

By letting the notation of the *quadratic form* $\|y\|_Z^2 \triangleq y^T Z y$, we rewrite the *stage* and *terminal* costs in the classical *quadratic cost function* in 6.4a using the error terms $\Delta x = x(t) - x_{ref}(t)$, and $\Delta u = u(t) - u_{ref}(t)$, where x_{ref} and u_{ref} are respectively the reference state vector and steady state control input at time t. As a result,

the *objective function* can be expressed as the sum of *weighted squared distance* to the state x and control input u. Every time step t, the following **optimal control problem (OCP)** is solved repeatedly in real-time, which it is not a trivial task:

$$\min_{U} = \int_{t=0}^{T} \left(\left|\left| x(t) - x_{ref}(t) \right|\right|^2_{Q_x} + \left|\left| u(t) - u_{ref}(t) \right|\right|^2_{R_u} \right) dt + \left|\left| x(T) - x_{ref}(T) \right|\right|^2_{P} \tag{6.5a}$$

$$\text{subject to} \quad x(0) = x(t_0) \tag{6.5b}$$

$$\dot{x} = f(x,u) \tag{6.5c}$$

$$u(t) \in \mathbb{U}_C \tag{6.5d}$$

In general, within this nonlinear MPC formulation we use the nonlinear system of the form

$$\dot{x} = f(x,u) \tag{6.6}$$

to model the process or the vehicle involved in the control work and it is usually composed by the *ordinary differential equations (ODE)* of the related "full dynamics" model (meaning not linearised). Next, the weighting matrix $Q_x \geq 0$ is the penalty on the state error, $R_u \geq 0$ is the penalty on control input error and $P \geq 0$ is the terminal state error penalty. These weights are tuned to suit the high-level *objectives* and levels of *uncertainty* of a given application [Hartley et al., 2013].

6.3 Rendezvous Total System Dynamics

When applying the aforementioned NMPC formulation on our Rendezvous landing problem we need to consider the *total system* consisting of both vehicles involved in the Rendezvous manoeuvre, i.e., the quadrotor UAV along with the ground mobile robot represented by the simulated two-wheeled robot (TWR). Therefore, in a similar fashion as in [Persson, 2021], we need to define the total state vector and system dynamics, by assuming the *stacked variable x* as follows:

$$x = \begin{pmatrix} x_{uav}^T & x_{twr}^T \end{pmatrix}^T, \qquad \dot{x} = f(x,u) \tag{6.7}$$

Specifically, the quadrotor dynamics described in 6.12, together with the two-wheeled robot (TWR) dynamics model described in 6.8 form the total prediction model of this nonlinear MPC. In the following paragraphs, we give a short review of the dynamic models corresponding to these vehicles.

Two-Wheeled Robot Dynamics

We remember that our URDF-based 2-wheeled robot was build in chapter 4 to approximate the movement of Spot robot on the ground in real-world experiments.

It is worth noting that when using PID control approach, as in our solution in section 4.6, no model of the plant is required but only tuning of just three gains. The *posture dynamics* of this two-wheeled robot (TWR), i.e., position and orientation, can be described by the following ordinary differential equations [Uddin, 2020]:

$$\dot{x} = u\cos\psi \tag{6.8a}$$

$$\dot{y} = u\sin\psi \tag{6.8b}$$

$$\dot{\psi} = r \tag{6.8c}$$

The state vector of this TWR consists of

$$\xi = \begin{pmatrix} x & y & \psi \end{pmatrix}^T \tag{6.9}$$

where x and y are the robot position and ψ is the orientation angle, with respect to the reference coordinate system, as illustrated in the schematic overview in Figure 6.3.

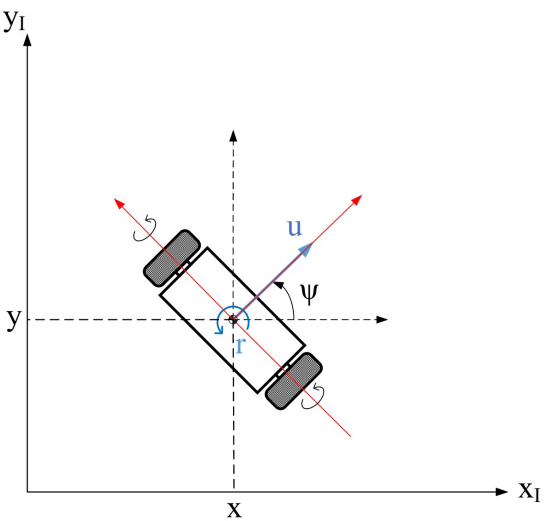


Figure 6.3 A schematic diagram of two-wheeled robot (Courtesy: [N. Uddin, 2020] [Uddin, 2020]). This corresponds to our URDF-based 2-wheeled robot built to approximate the movement of Spot robot on the ground in real-world experiments.

Quadrotor UAV System Dynamics

For clarity purpose, we review the overall simplified model representing the quadrotor nonlinear *full system dynamics*. By using Z-Y-X Euler angles, we define

$R(\psi, \theta, \phi)$ as the rotation matrix from the body frame $\mathbb{B}$ of the vehicle to the world frame $\mathbb{W}$. Then, the state vector of the UAV consists of

$$x = \begin{pmatrix} p & v & \phi & \theta & \psi \end{pmatrix}^T \tag{6.10}$$

where, p and v are the UAV position and velocity respectively, expressed in the inertial frame $\mathbb{W}$. Then, the control input vector is defined as

$$u = \begin{pmatrix} \phi & \theta & \dot{\psi} & T \end{pmatrix}^T \tag{6.11}$$

As described previously in 5.3 and 5.4, the quadrotor UAV model can be expressed by the following ordinary differential equations (ODE):

$$\dot{p} = v \tag{6.12a}$$

$$m\dot{v} = T \begin{pmatrix} -\sin(\theta) \\ \sin(\phi)\cos(\theta) \\ \cos(\phi)\cos(\theta) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} - Av(t) + d(t) \tag{6.12b}$$

$$\dot{\phi} = \frac{1}{\tau_\phi}(K_\phi \phi_d - \phi) \tag{6.12c}$$

$$\dot{\theta} = \frac{1}{\tau_\theta}(K_\theta \theta_d - \theta) \tag{6.12d}$$

$$\dot{\psi} = \dot{\psi}_d = \omega_d \tag{6.12e}$$

where the aerodynamic drag coefficients matrix A and the estimated external disturbance d are added to the system model. Next, the included first-order systems can be identified using classical system identification techniques as described in section 6.4, adapted from [Sa et al., 2017].

Specifically, the linear drag coefficients from air friction were estimated by assuming that the drag force is proportional to the velocity in that direction. The effect of including a simple model of the friction drag gives a significant improvement in the model [Persson, 2021].

Further, as described in [Borrelli et al., 2017; Kamel et al., 2017b], to achieve *offset-free tracking* under model mismatch, the system model is augmented with additional disturbances state $d(t)$ to capture this model mismatch. An *observer* is employed to estimate $d(t)$ and provide it to the MPC controller each time step, in real-time. These disturbances can include external forces (such as the wind for instance) and also a modelling error. The observer is designed using an augmented Extended Kalman Filter (EKF) based on the second order dynamics model [Sa et al., 2017].

Moreover, note that the vehicle heading (yaw) angular rate $\dot{\psi}$ is assumed to track the command instantaneously, i.e., perfect tracking. This assumption is reasonable because the UAV heading angle (yaw) has no effect on the UAV position [Kamel et al., 2017a].

Constraints

Finally, being able to directly include constraints in the optimal control problem (OCP) in the MPC formulation is an important ingredient for dealing with real systems with physical constraints. In our work, the UAV quadrotor has physical limitations in thrust, and it is unsafe to fly with too large attitude angles. These limitations are formulated as constraints on the control signals $u \in \mathbb{U}_C$, embedded in the MPC controller formulation in 6.5 and can be expressed by:

$$\begin{bmatrix} \phi_{min} \\ \theta_{min} \\ \psi_{min} \\ T_{min} \end{bmatrix} \leq u = \begin{bmatrix} \phi_{ref}(t) \\ \theta_{ref}(t) \\ \psi_{ref}(t) \\ T(t) \end{bmatrix} \leq \begin{bmatrix} \phi_{max} \\ \theta_{max} \\ \psi_{max} \\ T_{max} \end{bmatrix} \tag{6.13}$$

In the following table 6.1, we summarize the relevant numerical specifications of the UAV quadrotor DJI M100, found on the corresponding website [2], showing the limitations in the input variables. The same can be arranged for the ground mobile robot (UGV) involved in the Rendezvous. manoeuvre.

Inputs	Variables	Limits
Roll and pitch angles	ϕ, θ	35 degrees (0.611 rad)
Horizontal velocities	v_x, v_y	30 m/s
Angular rates	ω_x, ω_y	150 deg/s ($5/6\pi$ rad/s)
Vertical speed	$\dot{h}$	-5 to +5 m/s
Height	h	0 to 120 m
Thrust	T	0% to 100%

Table 6.1 DJI Matrice 100 relevant limitations.

In similar fashion as [Persson, 2021], for the touchdown of the landing to be gentle, it would be nice to impose an additional constraint on the vertical velocity at touchdown.

6.4 DJIM100 Full Dynamics System Identification

As we have mentioned, to achieve high performance with the cascade approach, followed in this project as in many different literature like among others [Kamel et al., 2017b; Kamel et al., 2017a; Sa et al., 2017; Persson, 2021], it is necessary to account for the inner-loop dynamics in the MPC position tracking controller.

DJI drone manufacturer does not provide the essential scientific resources such as *attitude dynamics* and the structure of underlying *autopilot controller*. Therefore,

[2] `https://www.dji.com/se/matrice100/info`

in the paper [Sa et al., 2017], they addressed these gaps by performing full dynamics system identification, both in simulated and experimental environment, using only the built-in onboard IMU and without applying any restrictive simplifying assumptions [Sa et al., 2017]. Then, this critical information is used to design the subsequent MPC-based horizontal position controller addressed and formulated in [Kamel et al., 2017b]. Open-source code of the related software packages including modified SDK, linear MPC, and system identification tools and their documentation are available to the community on ETHZ ASL Github [3].

Their presented full dynamics system identification results from the simulator and experiments, where they estimate the dynamics by recording the input and output data at 100 Hz using Virtual RC to send actual control commands and the attitude measurements response from the IMU, while performing short manual flights on an onboard computer. As a result, they logged two sets of dataset for model *training* and *validation*. Experimental results for the control performance are evaluated using a motion capture system while performing hover, step responses, and trajectory following tasks in the present of external wind disturbances.

Assuming the first-order low-level attitude dynamics for MPC-based position controller and the second-order for the disturbances observer, in this section, we are going to summarize the important findings, in the aforementioned paper [Sa et al., 2017], of the DJIM100 identified dynamics using the batch-based system identification techniques. The estimated values of the related parameters such as time constant τ, DC gain K, damping ξ and natural frequency ω are presented in Table 6.2.

- *Roll and pitch attitude dynamics*: They assume a low-level flight controller that can track the reference roll ϕ^*, and pitch θ^* angles with first order behavior. This first-order approximation provides sufficient information to the MPC to take into account the low-level controller behavior. The resulting first-order dynamics identified for roll and pitch attitude angles are as follows:

$$\frac{y(s)_\phi}{u(s)_\phi} = \frac{K_\phi}{\tau_\phi s+1} = \frac{3.544}{s+2.118} \qquad \frac{y(s)_\theta}{u(s)_\theta} = \frac{K_\theta}{\tau_\theta s+1} = \frac{3.827}{s+2.43} \tag{6.14}$$

  where $y(s)_\phi$ is the IMU measurement and $u(s)_\phi$ is the input command. Then, the identified second-order dynamic models exploited by disturbances observer are

$$\frac{y(s)_\phi}{u(s)_\phi} = \frac{K_\phi\ \omega_\phi^2}{s^2+2\ \xi_\phi\ \omega_\phi\ s+\omega_\phi^2} = \frac{26.37}{s^2+5.32s+27.04} \tag{6.15}$$

$$\frac{y(s)_\theta}{u(s)_\theta} = \frac{K_\theta\ \omega_\theta^2}{s^2+2\ \xi_\theta\ \omega_\theta\ s+\omega_\theta^2} = \frac{28.86}{s^2+6.00s+27.45} \tag{6.16}$$

[3] `https://github.com/ethz-asl/mav_dji_ros_interface`

- *Yaw and height dynamics*: The input commands of yaw and height are rates, u_ψ, $u_{\dot{z}}$, and the desired references are orientation, ψ and position, z. This implies there are controllers that tracks the desired yaw rate, $\dot{\psi}^*$, and height velocity, $\dot{z}^*$.

$$\frac{y(s)_\psi}{u(s)_\psi} = \frac{K_\psi}{\tau_\psi s+1} = \frac{5.642}{s+5.268} \qquad \frac{y(s)_{\dot{z}}}{u(s)_{\dot{z}}} = \frac{K_{\dot{z}}}{\tau_{\dot{z}} s+1} = \frac{3.342}{s+2.99} \tag{6.17}$$

Similarly, the corresponding identified second-order dynamic models are

$$\frac{y(s)_\psi}{u(s)_\psi} = \frac{K_\psi\ \omega_\psi^2}{s^2+2\ \xi_\psi\ \omega_\psi\ s+\omega_\psi^2} = \frac{593.3}{s^2+89.0s+549} \tag{6.18}$$

$$\frac{y(s)_{\dot{z}}}{u(s)_{\dot{z}}} = \frac{K_{\dot{z}}\ \omega_{\dot{z}}^2}{s^2+2\ \xi_{\dot{z}}\ \omega_{\dot{z}}\ s+\omega_{\dot{z}}^2} = \frac{25.43}{s^2+7.16s+24.8} \tag{6.19}$$

where $y(s)_\psi$ is obtained from the built-in IMU, gyro measurement along z-axis, and $y(s)_{\dot{z}}$ is provided by a motion capture device.

	ϕ	θ	$\dot{\psi}$	$\dot{z}$
1st Order	$\tau_\phi = 0.472$ $K_\phi = 1.673$	$\tau_\theta = 0.472$ $K_\theta = 1.575$	$\tau_{\dot{\psi}} = 0.161$ $K_{\dot{\psi}} = 1.057$	$\tau_{\dot{z}} = 0.334$ $K_{\dot{z}} = 1.118$
2nd Order	$K_\phi = 0.975$ $\xi_\phi = 0.512$ $\omega_\phi = 5.200$	$K_\theta = 1.052$ $\xi_\theta = 0.573$ $\omega_\theta = 5.239$	$K_{\dot{\psi}} = 1.079$ $\xi_{\dot{\psi}} = 1.898$ $\omega_{\dot{\psi}} = 23.448$	$K_{\dot{z}} = 1.024$ $\xi_{\dot{z}} = 0.718$ $\omega_{\dot{z}} = 4.985$

Table 6.2 DJI Matrice 100 identified dynamics summary, found by [Sa et al., 2017]. The resulting estimated parameters are for the low-level attitude dynamics modelled as a first-order behaviour and the second-order model for the disturbance observer, that is, time constant τ, DC gain K, damping ξ and natural frequency ω

.

Using DJIM100 platform has many advantages such its low-cost, high payload, ease of use and the ready availability of replacement parts, user friendly interface, powerful SDK, and a large user community [Sa et al., 2017].

6.5 MPC Controllers Optimization-Based Implementation

Historically, development and application of model predictive control (MPC) originated in *process control industries* where the processes being controlled are often

"sufficiently slow" to permit its implementation. Thus, the need for "fast computation" was one of the main challenges in implementing MPC. However, the recent rapid advances in computation have enabled MPC to be used prevalently in *autonomous vehicles applications*, where *trajectory generation* is particularly important for achieving safe operation in complex environments [Murray, 2022].

Indeed, nowadays it is possible to perform mapping, 3D reconstruction, localization, planning and control "completely on-board" thanks to the great advances in electronics and semiconductor technology. To fully exploit the robot capabilities and to take advantage of the available computation power, *optimization-based control (OBC) techniques* are becoming suitable for real-time UAV control [Kamel et al., 2017a].

We remember that in the assumed cascade control approach, the high-level MPC-based controller generates attitude commands for the low-level controller running on the autopilot, the firmware already provided on the commercial UAV platform. A classical Linear Model Predictive Controller (LMPC), based on a linearised model of the UAV, is presented in [Kamel et al., 2017a] and compared against a more advanced Nonlinear Model Predictive Controller (NMPC) that considers the full system model, as the one proposed in this project. Both LMPC and NMPC controllers enable dynamic trajectory tracking for the UAV quadrotor together with an open source C++ implementation of both controllers on [Kamel et al., 2017b].

As explained in [Kamel et al., 2017b; Kamel et al., 2017a; Sa et al., 2017], to implement the *linear MPC*, an efficient QP solver using `CVXGEN` code generator framework by [Mattingley et al., 2011] is employed to solve the optimization problem every time step. Whereas the *nonlinear MPC* is implemented by solving the optimization problem in 6.5 every time step using an efficient solver generated by `ACADO toolkit` using automatic code generation techniques for nonlinear model predictive control algorithms, based on the work in [Houska et al., 2011] and ACADO Toolkit Homepage [4]. Two useful examples on how to implement an MPC problem using ACADO tools and CVXGEN are reviewed in Appendix in Listing 9.3 and Listing 9.4.

For completeness, it is worthwhile to note that *Differential flatness* and *motion primitives* can be very important techniques to use for enabling rapid computation when implementing receding horizon control (MPC) [Murray, 2022] and [Åström and Murray, 2020].

6.6 Solving Rendezvous Landing Problem with various MPC Algorithms

For solving our Rendezvous problem, tracking and landing the quadrotor UAV on top of a moving ground robot, we are going to review three alternative landing

[4] `http://acado.github.io`

control strategies using MPC, that have been considered in several project works over the last years. All of these MPC controllers will guide the drone to track the target ground robot at the landing task.

These methods are different such that two of them are more advanced and consider *cooperative* landings, using centralized MPC, by exchanging the kinematic informations (trajectory and velocity) of each other and sharing the control action to synchronize the Rendezvous manoeuvre and achieve a safer *touchdown*. Whereas in the other simple strategy only the *tracking and landing* are considered using linear or non-linear MPC without having any interactions between the UAV and the ground robot, namely non-cooperative Rendezvous landing.

Because of the *distributed nature* of the cooperative autonomous landing problem, we can distinguish between *centralized* MPC and *distributed* MPC. A comparison between a centralized MPC implementation and a distributed one is described in the work of [Bereza et al., 2020; Persson, 2021], to investigate the computation efficiency. In the distributed MPC, the computation load is divided/distributed between the cooperating vehicles. Each vehicle computes its own control inputs, based on a predicted state trajectory of the other vehicle. The vehicles exchange their predicted trajectory and adjust their own trajectory to better adapt themselves to each other [Persson, 2021].

Again, in order to control the position of the quadrotor UAV in the tracking and landing tasks this problem requires stabilization and control of the attitude dynamics. For this reason, we need to make use of the *cascade control* architecture, with inner and outer loops.

Method 1: Non-Cooperative Rendezvous using MPC

In this method, the Rendezvous landing problem is solved with the existing classic Model Predictive Control (MPC) theory which plans a safe trajectory and follows it in a non-cooperative manner and the computations are done on the UAV computer. This model-based trajectory tracking controller for multi-rotor UAV has been considered by several similar projects, in particular, in the ETH Zurich paper work in [Kamel et al., 2017b]. This paper work presents a good detailed description of MPC strategies, linear and non-linear MPC, to achieve precise *trajectory tracking* for multi-rotor and fixed-wing UAVs, which is also similarly used by our WARA-PS simulator.

This controller design for aerial robots trajectory tracking consists of a cascade structure with high-level MPC position control, as illustrated in Figure 6.4. However, in our Rendezvous landing algorithm the error vector in the objective function of the optimization problem in MPC formulation is defined by the error vector described in 4.1. *Actually, we replace the reference x_{ref} in Figure 6.4 by the position of the target robot.* As such, the quadrotor UAV can track robot position and perform the landing manoeuvre successfully in non-cooperative manner, that is, without any interactions with the ground moving robot.

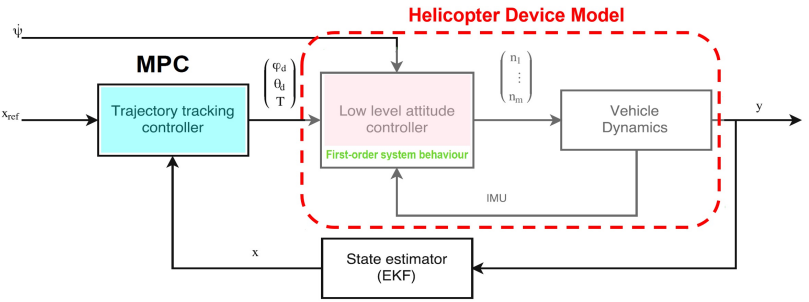


Figure 6.4 Cascade controller structure for multi-rotor system. $n_1 \cdots n_m$ are the i-th rotor speed and y is the measured vehicle state. This cascade control approach is used for precise trajectory tracking based on MPC in [Kamel et al., 2017b], where a high-level MPC generates attitude commands for a low-level controller running on the autopilot. The dynamic behavior of the attitude controller is considered as a first-order system by the high-level controller. As such, position control is split into two parts: An outer trajectory tracking controller that calculates attitude and thrust references which are then tracked by the inner attitude tracking controller.

Method 2: Cooperative Rendezvous using Centralized MPC

The methods in this section, and the next one, develops the existing MPC theory as a basis by adding cooperative control flavours. Specifically, the *optimal solutions* from the MPC $u^* = (u^*_{spot}, u^*_{uav})$ are sent to *low-level autopilots* that compute the actuator inputs. Here, the control inputs for both vehicles are computed by the same process, *centralized* controller. Because the dynamics of the UAV are faster than the dynamics of the ground vehicle (UGV), the computations are done on the drone computer [Bereza et al., 2020], and the ground robot receive its control inputs from the drone, as illustrated in Figure 6.5.

Again, we remember that the Cascade control approach for multi-rotor system is followed here as well, in which the position control is split into two parts: An outer trajectory tracking controller calculates attitude and thrust references, that are tracked by an inner attitude tracking controller.

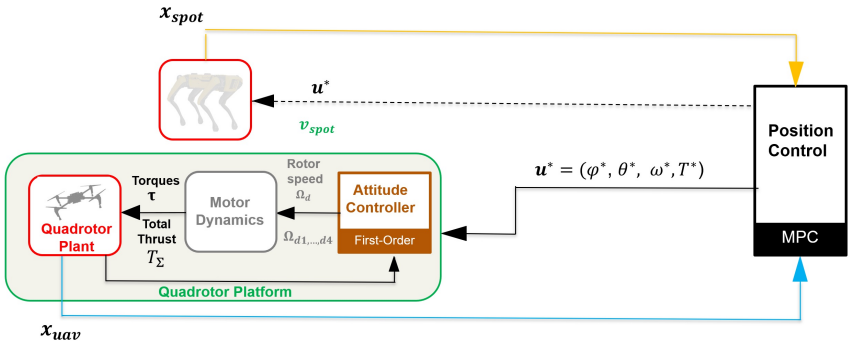


Figure 6.5 Schematic of the centralized cooperative MPC architecture: To generate feasible rendezvous trajectories for the vehicles, we use centralized Model Predictive Control (MPC). The optimal solutions from the MPC, $u^*_{uav} = (\phi^*, \theta^*, \omega^*, T^*)$ and $u^*_{spot} = v^*_{spot}$, are sent to low-level autopilots of both ground and aerial vehicles. Here, the control inputs for both vehicles are computed by the same process. We notice that for this aerial vehicle (UAV) a Cascade control approach is used.

Method 3: Cooperative Rendezous using Centralized MPC with Decoupling

Similarly, as described in the aforementioned cooperative approach in the previous method 2, cooperative centralized MPC is also used here to generate feasible rendezvous trajectories for the cooperating vehicles . However, this MPC is now partitioned into two parts, one acting in the horizontal plane (lateral MPC controller) and another one in the vertical plane (vertical MPC controller), as shown in Figure 6.6.

Mathematically speaking, we separate the UAV motion into two sets of equations and at the end we formulate two distinct optimization problems in MPC, namely one horizontal OCP problem and another vertical OCP of the same form as the aforementioned optimal control problem (OCP) described in 6.5.

according to the work in [Bereza et al., 2020; Persson, 2021], by considering both the horizontal and vertical dynamics, the trajectory is planned in a safe and efficient manner where the *feasibility* is taken into account during the descent. This partition significantly simplifies the solution of the optimization problem.

Many of MPC solutions for autonomous landings on moving platforms do not consider the *vertical part* of the landing trajectory in the MPC. On the other hand, several solutions over the last years decouple the planned MPC trajectory in both the horizontal and vertical plane as in [Persson et al., 2017; Persson and Wahlberg, 2019; Bereza et al., 2020; Persson, 2021]. In particular, the method presented in [Persson, 2021] integrates the vertical part of the landing trajectory in the MPC trajectory planning such that the vertical trajectory can start before the *lateral alignment* in $\Delta x, \Delta y$ is completed, but still can guarantee that Δh will not go to zero before

it is safe to do so.

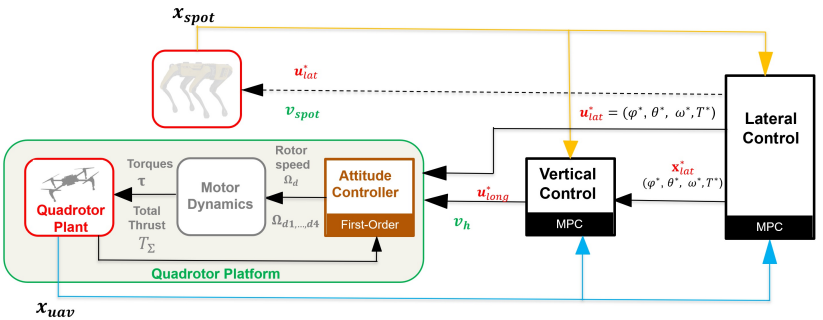


Figure 6.6 Schematic of the decoupled dynamics MPC architecture: To generate feasible rendezvous trajectories for the vehicles, the MPC is now separated into two parts, one acting in the horizontal plane (lateral MPC controller) and one in the vertical (vertical MPC controller). Then, the optimal solutions from the MPC (u^*_{lat}, u^*_{long}) are sent to low-level autopilots of the both ground and aerial vehicles.

General Statements about Rendezvous Landing Algorithm

For better understanding and safely performing the Rendezvous manoeuvre, it is wise to review the following key points to take into account when implementing such autonomous Rendezvous landing algorithms, based on the various related work mentioned previously like [Muskardin et al., 2017; Persson et al., 2017; Persson and Wahlberg, 2019; Bereza et al., 2020; Persson, 2021]. In the context of Rendezvous settings, focus is on the *interaction* itself, with emphasis on how the system properties affect convergence:

- Yet, Rendezvous problem between the quadrotor UAV and the ground vehicle (UGV) is defined to be achieved when zeroing all the *relative position error* terms. Specifically, the relative states in the horizontal plane $(\Delta x, \Delta y)$ need to converge to and remain close to zero before the relative altitude (Δh) can go to zero. This requirement that $(\Delta x, \Delta y) \to 0$ before $\Delta h \to 0$ is crucial for the safety of the manoeuvre. Otherwise there is a large risk of the vehicles crashing into each other.
- The vehicles would rather be able to perform the rendezvous while moving at a certain speed, and/or satisfying other constraints.
- The touchdown velocity cannot be greater than some bound h_{td}^{max}.
- *Replanning of trajectory*: Neither the rendezvous coordinate, nor the time of arrival are fixed, but are both solved for online. With MPC, the trajectory is replanned in each iteration, meaning that the last produced trajectory will be feasible and efficient based on the latest state measurements.

- The *planned trajectory* can be executed using RTK-GPS system for better precision, where the measurements are fed back to the MPC control to compute an appropriate *distance error vector*, defined between the current pose of the UGV and the pose of the quadrotor UAV.

- *Low-level Control*: The attitude dynamics of the internal low-level controller in the commercial quadrotor UAV are included in the model and can be approximated as first or second-order systems. For example, in [Persson, 2021] the transfer functions from command roll ϕ^* and pitch θ^* to attitude ϕ and θ are in the form of Laplace transfer functions expressed as

$$G_\phi(s) = \frac{K_\phi \ \omega_\phi^2}{s^2 + 2 \ \xi_\phi \ \omega_\phi \ s + \omega_\phi^2} \qquad G_\theta(s) = \frac{K_\theta \ \omega_\theta^2}{s^2 + 2 \ \xi_\theta \ \omega_\theta \ s + \omega_\theta^2} \tag{6.20}$$

  For the *yaw rate* $\dot{\psi}$, the control is modelled as

$$\Psi(s) = \frac{1}{s} \ G_\psi(s) = \frac{1}{s} \ \frac{K_\psi}{\tau_\psi \ s + 1} \dot{\Psi}^*(s) \tag{6.21}$$

  In the *vertical direction*, thrust is controlled such that the vertical velocity $\dot{h}$ approximately follows

$$\dot{H}(s) = \frac{K_v}{\tau_v \ s + 1} \dot{H}^*(s) \tag{6.22}$$

6.7 Control Strategy for Robust Rendezvous Landing

The Rendezvous landing controller can be designed to *robustly* land on a small surface on the ground vehicle, as described in the related works in [Mellinger et al., 2010] and [Lee et al., 2012].

We assume the position and velocity of the quadrotor are available for the feedback controller (MPC or PID) and the x and y position of the landing location can be sensed with zero mean error. We do not require the exact z height of the landing location to be sensed as it is detected from a change in *quadrotor UAV performance*.

The *sensing of an event* or the passing of a specified amount of time triggers a change in the controller mode, as illustrated in Figure 6.7. First, the quadrotor is controlled to *fly and align* (to hover) above the desired landing location on the ground vehicle. Next it is commanded to descend at a specified velocity. If a z velocity close to zero is sensed then the quadrotor has likely made contact with the surface and the quadrotor enters the *Idle Props mode*.

Note that event-triggered transitions are labelled with the event and time-triggered transitions are denoted by clocks. During the "Descend" state the quadrotor waits to sense an event before transitioning to the next state. If an error is sensed the quadrotor is controlled to hover (fly and align) at its original location.

The concept of an error is an important part of this control strategy. Here we sense an error by checking if the roll angle, pitch angle, or velocity are above the threshold values ϕ^{max}, θ^{max}, and v^{max}. These conditions are designed to catch situations when the quadrotor is falling off the desired landing location or when the quadrotor is in free fall because it switched into the Idle Props state when it was not actually on a surface.

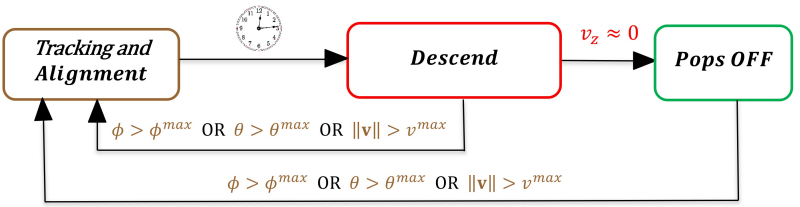


Figure 6.7 Control strategy for robust rendezvous landing: Note that event-triggered transitions are labeled with the event and time-triggered transitions are denoted by clocks. While in the Descend state the quadrotor waits to sense an event before transitioning to the next state. If an error is sensed the quadrotor is controlled to fly and align at its original location.

6.8 Position Estimate: Accurate Relative Positioning using RTK

The key state estimates required for the control of a quadrotor are its position (which is typically considered separately as position in the plane and height), attitude, angular velocity, and linear velocity. Of these states, the attitude and angular velocity are the most important as they are the primary variables used in attitude control of the vehicle [Mahony et al., 2012].

The most basic instrumentation carried by any quadrotor is an inertial measurement unit (IMU), which could be used to measure *angular rates* which are used to estimate the Euler angles (ϕ, θ, ψ). It is often augmented by some form of "height measurement", either acoustic, infrared, barometric, or laser based. Many robotics applications require more sophisticated sensor suites such as VICON systems, global positioning system (GPS), camera, Kinect, or scanning laser range finder.

While GPS-based control is sufficient for general positioning, assuming the signal is not blocked, the accuracy of GPS receivers fit for use on miniature UAVs is measured in meters, making them unsuitable for precision tasks such as landing [Lee et al., 2012].

Instead, for achieving high-accuracy positioning in the landing manoeuvre, it is possible to use a combination of Real Time Kinematic (RTK) technology [5] with

[5] https://en.wikipedia.org/wiki/Real-time_kinematic_positioning

Global Navigation Satellite Systems (GNSS), such as the american GPS, the russian GLONASS, the european Galileo, BeiDou, and other constellation systems. As mentioned in WARA-PS portal [WARA-PS, 2024], the system can easily be integrated with different types of vehicles and platforms, making them compatible with different types of UAVs in the project.

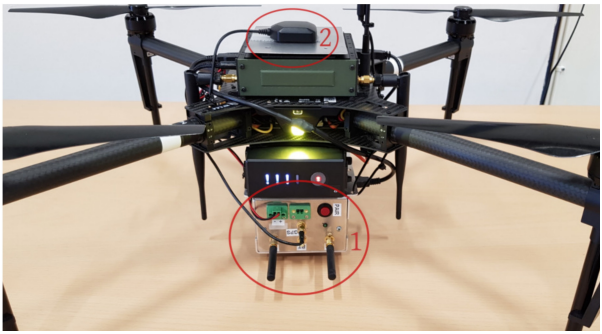


Figure 6.8 The RTK box attached under the DJI Matrice 100 drone in red circle (1) and the GNSS antenna in red circle (2). This RTK system is provided by SAAB Combitech AB through the WASP Research Arena (WARA) Public Safety [Persson, 2021]. (Courtesy: [L. Persson, 2021])

As illustrated in Figure 6.9, RTK technique uses the receiver's measurements of the phase of the satellite signal's carrier wave. Then it takes a correction service from an already surveyed GPS base station and applies that correction to a moving device called the rover, the moving GPS system. This combination of measurements and corrections will allow the receiver to solve "carrier ambiguities", i.e. it is then able to correct for atmospheric distortions and any errors in the existing GPS system. This changes our data information from meter-level accuracy positioning to centimeter-level accuracy positioning [6].

Some useful information about the RTK-GNSS system employed in WARA-PS can be found on GitLab LRS at LiU university [7]. In fact, the GPS system onboard of the DJI Matrice aerial vehicle platforms, used in WARA-PS for research and experimentation, makes use of the Real-Time Kinematic (RTK) positioning technique to deliver centimeter accuracy measurements [Andersson et al., 2021]. This RTK system comes from another project that SAAB Combitech AB is a part of, called DRIWS (Digital Runway Incursion Warning System). More details about this RTK setup through WARA-PS can be read in [Persson, 2021] and illustrated in Figure 6.8.

The demand for *scalable high precision technology* is growing rapidly, as evident in the automotive world with next generation ADAS (Advanced driver-

[6] https://inertialsense.com/how-rtk-provides-gps-centimeter-level-accuracy/
[7] https://gitlab.liu.se/lrs/lrs_wara_rtk

assistance systems) and V2X (Vehicle-to-everything) applications, and in robotics with applications such as UAVs and robotic lawnmowers [8].

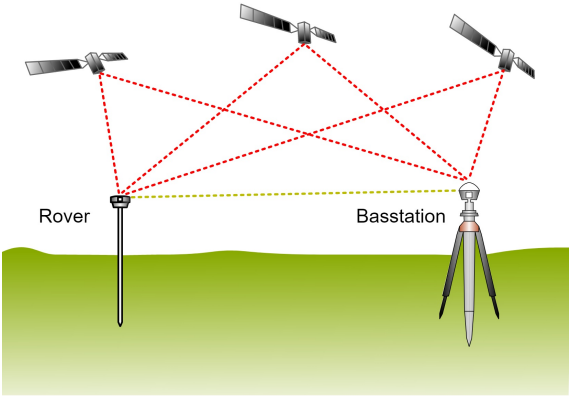


Figure 6.9 The Real-Time Kinematic (RTK) concept: RTK is the application of surveying (examination) to correct for common errors in current satellite navigation (GNSS) systems. **Carrier-phase tracking**: RTK follows the same general concept in GNSS but uses measurements of the phase of the satellite signal's carrier wave as its signal, ignoring the information contained within. RTK uses a fixed base station and a rover to reduce the rover's position error. The base station transmits correction data to the rover. (Courtesy: Wikipedia)

[8] `https://www.u-blox.com/en/technologies/high-precision-positioning`

7

AI Planning Problem: Automated Robotic Inspections (ARI)

Deploying aerial, maritime or ground robots in real-life applications is growing rapidly thanks to their ability to perform tasks that humans are unable to do easily. However, this requires "higher autonomy" when operating in real environment and particularly in the field of inspection operations, where mobile robots are already employed to perform critical tasks and need to be facilitated by *autonomous planning capabilities*, while respecting their own sensor limitations and motion constraints.

From the topic of "automating industrial inspections using agile mobile robots" we consider a small inspection scenario to use in our planning problem, in the second part of the thesis. We suppose that we have a "world" consisting of *a collaborative robots system* that must perform a set of inspection tasks in different locations inside an industrial/construction site.

More specifically, this *scenario* of "automated robotic inspection (ARI)" requires that the *quadruped UGV Spot robo*t, with the collaboration of at least one *quadrotor UAV DJI M100 drone*, will navigate the construction/industrial site and perform "visual inspections" on points of newly constructed work or, generally speaking, on predefined inspection points situated in different locations. As stated previously, we will use *Planning Domain Definition Language* (PDDL) as the language for modelling our AI planning problem, in which we must provide two files: the PDDL domain and problem.

Here it is worth to note that *hierarchy* lends itself to *efficient plan construction* because the planner can solve a problem at an abstract level before delving into details. Domain-independent and domain-specific planning complement each other. In a hierarchically organized actor, planning takes place at multiple levels of the hierarchy. At high levels, abstract descriptions of a problem can be tackled us-

ing domain-independent planning techniques [Ghallab et al., 2016a]. A sample of abstraction hierarchy for our robotic inspection mission is illustrated in Figure 7.1.

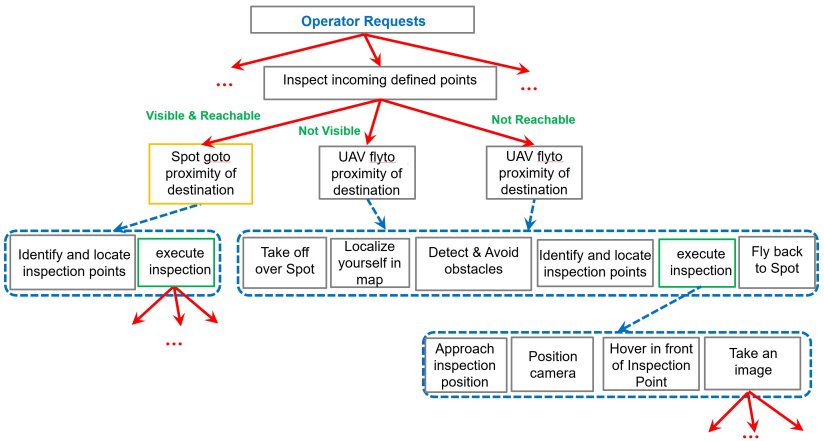


Figure 7.1 Multiple levels of abstraction for the inspection domain: Each solid red arrow indicates a "refinement" of an abstract action into more concrete ones. Each dashed blue arrow maps a task into a **plan of actions**. For unmanned aircraft, actions may for example include taking off and landing, flying between waypoints or to a distant destination, hovering, positioning cameras and taking pictures.

7.1 Mission Specification: Generating and Executing

As reviewed in previous sections, planning implements a mapping of its "input", a predictive model and a problem, into an "output plan" expressed in some representation. Given a specification of the "current state" of the *world* together with a declarative specification of the "prerequisites and consequences" of the *available actions*, a task planner can automatically search for a way of combining actions into a *solution plan* $\pi = a_0, \ldots, a_n$ that achieves a given goal [Ghallab et al., 2004].

For Spot robot actions can be moving between waypoints or to some destination, positioning cameras and taking pictures, whereas for unmanned aircraft expanded actions may for example include taking off and landing, flying between waypoints or to a distant destination, hovering, positioning cameras, taking pictures, and loading or unloading cargo, as illustrated in Figure 7.1. In particular, multi-rotor UAV is appropriate for inspection tasks that requires flying close to structures to obtain detailed footage.

A variety of *off-the-shelf planners*, from WARA-PS and other academic planning research groups, are available and can be used to generate "mission specifications" by specifying a mission goal to be planned for. Indeed, for our planning

problem, we have used the planner available on the website "planning.domains" [1], with an integrated PDDL Editor [2] where a solution plan could be generated, composed of the sequence of actions for the goal, as shown in Listing 7.8.

as illustrated in Figure 7.3a, after generating a plan for the goal using an off-the-shelf automated planner this output mission specifications are translatable into "executable" and fully specified *Task Specification Trees (TSTs)* created through a *TST Factory* in WARA-PS framework. In this way, goal nodes in a task specification tree will be expanded into "detailed" task specifications that should then be executed.

TST Factory In WARA-PS framework "tasks" can be concretely represented as Task Specification Trees. Nodes in such trees, which are general and declarative specifications of tasks to perform, are both created by and stored in a *TST Factory*, which is a ROS node providing services related to Task Specification Trees and their creation and execution. There are two fundamental ways of creating a TST node or a full TST tree in a TST factory:

1. **ROS service calls**: You can call a set of services in the TST Factory to create root nodes and child nodes and to set individual parameters of each node.

2. **JSON specifications**: You can create a tree specification in a JSON-based language. This removes the need to make potentially hundreds of service calls and replaces this with generating an explicit tree structure in the language of your choice.

[1] `http://planning.domains/`
[2] `http://editor.planning.domains`

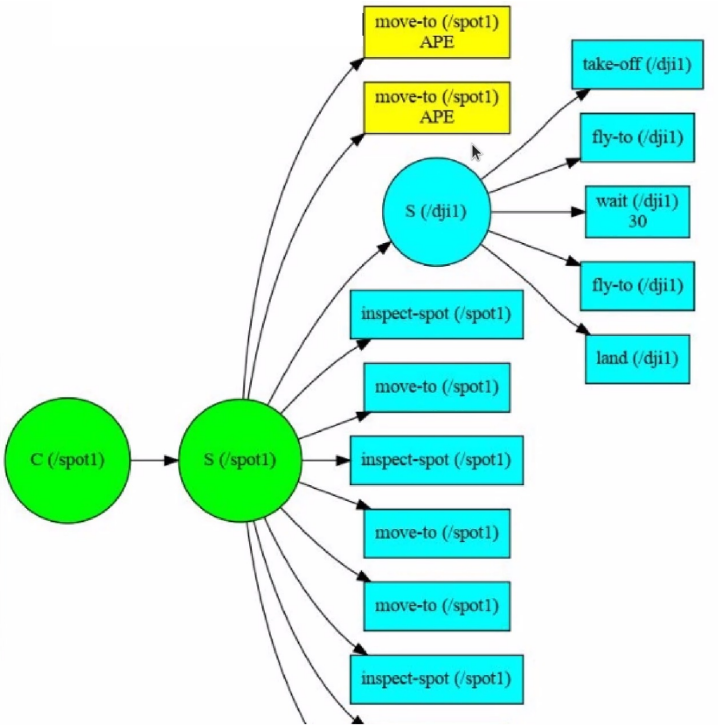


Figure 7.2 A fragment of the Task Specification Tree (TST) converted from the plan solution using TST factory in the WARA-PS framework: The corresponding TST uses elementary action nodes, corresponding to elementary actions of type `inspect-spot`, `move-to`, `fly-to`, etc. Furthermore, it requires a concurrent node (marked C) specifying that the actions can be performed concurrently, as well as a sequential node (marked S).

To recapitulate, for solving our automated planning problem we need to solve the following main two sub-tasks:

1. Model the inspection "domain" and the "problem" instance in the high-level mission specification language PDDL and select a suited PDDL-based planner to solve this problem.

2. Translate the resulting plan solution, output of the PDDL-based task planner, to TST using WARA-PS framework to eventually generate "executable actions" for Spot and AUVs to carry out.

(a) **(b)**

Figure 7.3 (a) Automated Planning for Mission Specifications: Task planners can be used to generate mission specifications by specifying a mission goal to be planned for, and then WARA-PS framework will be used to convert the plan into a fully specified and **executable TST**. For unmanned aircraft, expanded actions may for example include taking off and landing, flying between waypoints or to a distant destination, hovering, positioning cameras and taking pictures. (b) Inspecting in construction environment: As construction progresses, inspecting data can be used to compare the newly constructed work against the as-designed model or drawings for quality assurance. In construction sites, the flexible platform of Spot can be used to automate sensing, inspecting and data capture. In addition, multi-rotor UAV is also appropriate for inspection tasks that requires flying close to structures to obtain detailed footage. (Courtesy: Boston Dynamics)

7.2 Modelling the Robotic Inspections Scenario

In this section we describe how the robotic inspection problem is modelled in PDDL. First, it is important to note, as recommended in [Ulam et al., 2007], that in order "to fully specify a mission", namely step-by-step instructions of the generated solution plan, to guide one or more robots to accomplish a set of tasks , one must detail: (1) *The tasks to undertake*; (2) *The way to perform the tasks*; and (3) *any temporal constraints that may exist between the tasks or behaviors* (for example, the requirement of finding a target before tracking it). As such, let's define the scenario and describe the planning problem that we need to solve in this thesis work in the following key features:

- **Inspection scenario**: We consider a small scenario of construction validation that consists of some inspecting points of the construction site to be analysed by taking picture or video live-streaming. In construction, these inspection tasks are used to check construction progress against the design, i.e., to validate that the work newly put in place is as designed. In general, I would remark that our PDDL model presented in Listing 9.1 would represent not only the scenario in construction domain but in any other industrial site as well.

- **Problem specification**: The mission is a small inspection problem in building construction domain with at least two robots and five inspection points, where the quadruped Spot robot would start the inspection tasks at point ip0

and come back to it at the end of the mission. In case of higher and difficult-to-access inspection points, not reachable locations or not visible objects of interest, then these tasks will be allocated to a UAV DJIM100 drone, instead of the expensive scaffolding. Thus, this quadrotor should intervene to inspect instead of the mobile ground robot and then fly back to land atop Spot, before discharging the respective battery. In Figure 7.4 it is illustrated a conceptual overview that would formalize the *robotic inspections world* in which it accommodates Spot robot and inspection points and the conceptual representation of the *two inference tasks*: "visibility" of objects and "reachability" of locations.

- **High-level abstract actions**: Domain file of the PDDL model presented in Listing 9.1 describes robot operations that would lead to the desired goal state, that is, to move between the predefined inspection points *(Move-spot)* and to perform visual inspection tasks (`inspect-spot, inspect-uav`). However, this is a baseline PDDl-model, where we consider only high-level abstract actions to accomplish the mission goals. Therefore, when needed it is possible to "refine" these actions into more concrete ones, like for example to expand the action (`inspect-spot`) and formulate more detailed or specified actions *(take_picture, position_camera, etc.)*, as shown in Figure 7.1. Again as stated in [Ghallab et al., 2016a], At high abstract levels, the planner can solve a planning problem before delving into details using *domain-independent planning* techniques.

- **Temporal constraints**: There are some temporal constraints that may be considered when operating the two different robot systems, Spot and DJIM100. However, such these constraints we don't consider in our PDDL domain in Listing 9.1 because we would perform *offline tasks* without struggling with the problematic of integrating simultaneous execution, when it is not necessary. We recall that AI planning can be done online or offline. Online planning would be essential in more challenging domains with a "changing environment". As mentioned in [Torreño et al., 2017], online planning poses a series of challenges derived from the integration of *planning and simultaneous execution* and the need to respond in complex and real-time environments. Real-time planning, planning and simultaneous execution in a changing environment, is an interesting and exciting research line that is very relevant in applications like soccer robots, space exploration and military operations.

  For completeness and future study/work it is worth to discuss briefly some examples of the temporal constraints related to our inspection domain in the following:

  – In order to move between the inspection points, the robots which are available to be tasked during mission execution, Spot and DJI, must

power on the *navigation system* to localize a target point before inspecting it.

- In order to perform the required inspection (take a picture or point the camera), Spot/DJI must be *stopped/hovered* at the desired location.
- In order to avoid energy overconsumption, some robot operations are necessary to *manage the power on/off* in the two different robot systems, Spot and DJIM100. Each subsystem can be on or off and must be on before some operation.

Construction sites and industrial facilities often require periodic inspections and revisions of the newly constructed work (hold points) or key components/installations. Examining these points of interest can be time consuming, potentially hazardous or require special equipment to reach. Teaming mobile robots with Unmanned Air Vehicles (UAVs) are ideal platforms to automate this expensive and tedious inspection task.

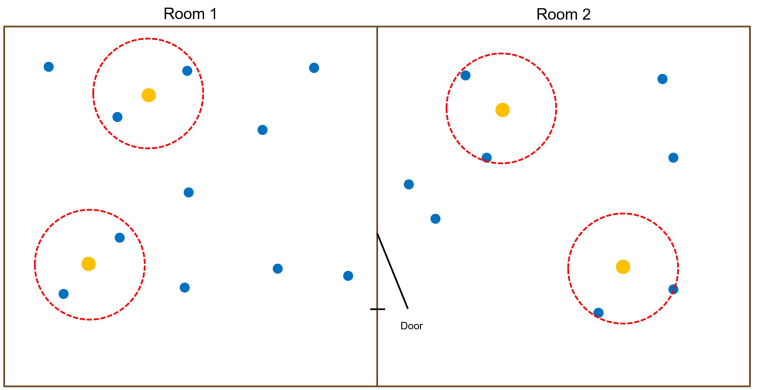


Figure 7.4 Conceptual overview depicts the **automated robotic inspections (ARI) world**: "Room1" and "Room2" formalize the space to accommodate robots and inspection points and allow the actions needed to accomplish the goal of inspection missions. Yellow circles represent Spot robot, whereas the blue ones represent the predefined inspection points in some construction/industrial environment. The red dashed circles indicate the conceptual representation of the two inference tasks: the **visibility** of objects (inspection points) or the **reachability** of locations by Spot robot to these inspection points.

PDDL-Model of Planning Problems: Domain and Problem Descriptions

'*Physics, not advice*' was the slogan of Drew McDermott, the originator of PDDL. This indicates a *neutral specification* of the planning problems, that is, the language should focus on expressing the *physical properties* of the world rather than advising the planner on how to search for solutions [Steenstra, 2019].

Before presenting the PDDL code corresponding to our robotic inspection domain, we review how to model a "world" in PDDL, in which there are certain things we need to keep track of. As discussed in previous sections, a world is described by a *set of states*, each containing a list of *facts* and/or *objects*. It begins with an *initial state* and is governed by a set of rules and constraints that limit which actions can be taken in each state, and each *action* generally represents a transition to a different state. In general, a PDDL planning model compactly describes the mechanics of the problem environment [Komenda et al., 2016].

Another important feature of these PDDL planning models, according to various literature like in [Geffner and Bonet, 2013; Haslum et al., 2019; Torreño et al., 2017], PDDL separates the model of the planning problem in two major parts as follows, defined by the respective main ingredients shown in Figure 7.5:

1. **domain** description: The domain definition contains the domain *predicates* (relevant properties of objects, that can be true or false) and *operators*, called "actions" in PDDL (means to change the state of the world). It may also contain *types, constants, static facts* and many other things, but these are not supported by all the planners. The format of a simple PDDL domain definition is depicted in Figure 7.5a.

2. the related **problem** description: The problem file represents an "instance" of the world we established in the domain. It determines what is true at the start of the plan (initial state), and what we want to be true at the end of the plan (goal state). It may also contain *objects* (things of interest). The basic syntax of a PDDL problem file is depicted in Figure 7.5b.

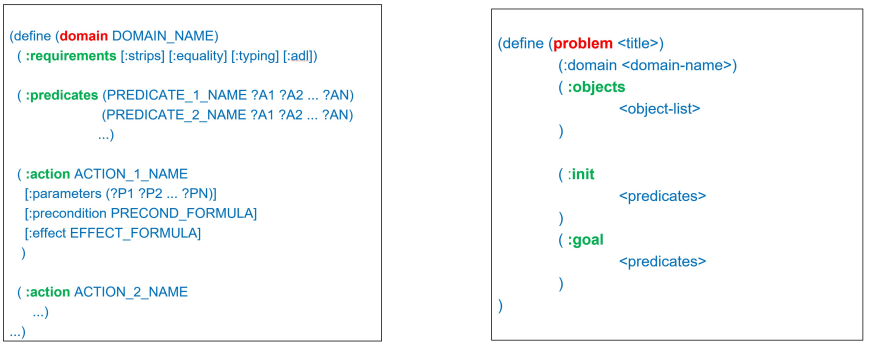


Figure 7.5 (a) The format of a (simple) PDDL **domain** definition: As we know in formal grammars elements in [ ]'s are optional. **Names** (domain, predicate, action, etc.) are usually made up of alphanumeric characters, hyphens ("−") and underscores ("_"), but there may be some planners that allow less. **Parameters** of predicates and actions are distinguished by their beginning with a question mark ("?"). The parameters used in predicate declarations have no other function than to specify the number of arguments that the predicate should have, i.e. the parameter names do not matter. Predicates can have zero parameters (but in this case, the predicate name still has to be written within parentheses). (b) The basic syntax of a PDDL **problem** file, where <title> is the title of the problem file and <domain-name> refers to the name of the corresponding domain file.

Therefore, it is important to note that *several problem descriptions* may be connected to the *same domain description*, just as several instances may exist of a "class" in OOP (*Object Oriented Programming*) for example. Thus, a domain and a connecting problem description forms the PDDL-model of a planning problem, and eventually this is the input of a planner software, usually a domain-independent AI planner, which aims to solve the given planning problem via some appropriate planning algorithm.

The corresponding domain and problem definitions of our inspection domain, presented in Listings 9.1 and 9.2, are placed in two files, using the extension "`.pddl`" respectively. This is not mandated by the PDDL language, but many planners require it.

PDDL Domain Definition for the ARI Inspection Task

In this section and the next one we give a brief explanation about the encoding process of our planning problem in PDDL, described by domain and problem definitions respectively. When modelling a problem in PDDL, one must always be prepared to reformulate the model to work around planners' limitations and quirks [Haslum et al., 2019].

In the following, we formulate a small-scale practical *robotic inspection*

problem in a basic PDDL encoding. As shown in Listing 7.1, a representation of our domain file starts with defining the *domain name* `(define (domain inspections))` and some *requirements* for the PDDL-solver are listed in the `(:requirements ...)` field.

The keyword `:strips` indicates that the domain is of the simplest form and comprises the basic PDDL functionality. Adding `:typing` to the requirements list allows the planner to define variables for *objects of different types*. Indeed in the `:types` field we declare the names of two different object types used in this model that correspond to the robots (`spot, uav`), and the locations of these robots and the inspection points (`position`). It is worth noting that these locations are conceptual types, i.e., the actual longitude and latitude of these locations are not stored in PDDL. Next, `:fluents` allows the use of *numeric values* for planning. Furthermore, with `:action-costs` we can assign to each action a *cost*.

If a domain stipulates no requirements, it is assumed to declare a requirement for `:strips` implicitly [Ghallab et al., 1998]. In general, the requirements specification is somewhat "redundant", most planners will examine the domain definition itself to determine if it uses any PDDL features that they cannot deal with rather than rely on the requirements keywords. It is, however, good practice to always write a correct requirements specification [Haslum et al., 2019]. In fact, a domain set of requirements allow a planner to quickly tell if it is likely to be able to handle the domain [Ghallab et al., 1998].

Listing 7.1 Planning variables declaration in the domain file - Robotic inspection

```
1 (define (domain inspections2)
2
3     (:requirements :strips :typing :fluents :action-
          costs)
4
5     (:types
6         spot uav position - object
7     )
```

As we have mentioned, PDDL is intended to express the "physics" of a domain, that is, what *predicates* there are, what *actions* are possible, and what the *effects* of actions are [Ghallab et al., 1998].

The `(:predicates ...)` block of the domain definition contains the list of the model's *state variables*. These are binary variables (Boolean-valued), meaning they represent facts that are either true or false, for describing certain properties of the objects used in the PDDL-model.

A predicate can be *static*, meaning a predicate that is not affected by any action. There is no special syntax to mark the predicate as static, it is only the fact that it does not appear in the `:effect` of any action that makes it so [Haslum et al., 2019].

Our predicates in Listing 7.2 indicate if Spot and UAV are at a particu-

lar position (spot-at) and (uav-at), some location is reachable for Spot (spot-reachable), the inspection point is visible for robots (spot-inspectable) and (uav-inspectable), and finally one predicate to represent the attaching state of the UAV quadrotor at the top of Spot (attached). A question mark indicates a parameter of the predicate. Then, we add *type specification* to every parameter in the predicate declarations, and the same for action declarations.

It is worthwhile recalling that sometimes, in our planning problem, some inspection points are partially or completely non-inspectable with Spot. That can be caused for example by high vertical or occult position which is difficult-to-access for Spot robot. Therefore, the UAV quadrotor should intervene in this particular inspecting task instead of Spot. In this case, uav-inspectable(?spot-pos, ?inspect-pos) is evaluated with the "intended semantics" that it can take-off, fly, inspect, fly-back and land as one uav-do-inspection action, i.e., as a high-level action.

Listing 7.2 Predicates in the domain file - Robotic inspection task

```
1
2     (:predicates
3         ;; Reachability of locations for Spot robot
4         (spot-reachable ?from-spot-pos ?to-spot-pos -
              position)
5
6         ;; Visibility: when one objective is visible
              for Spot
7         (spot-inspectable ?spot-pos ?inspect-pos -
              position)
8
9         (uav-inspectable ?spot-pos ?inspect-pos -
              position)
10
11        (spot-at ?spot - spot ?spot-pos - position)
12
13        ;; predicate to express the goal:
14        (inspected ?inspect-pos - position)
15
16        (uav-at ?uav - uav ?inspect-pos - position)
17
18        (attached ?uav - uav ?spot - spot)
19
20    )
```

The *planning states* are represented not only by *predicates* but by *functions* as well. These functions are declared in the field named `(:functions ...)` and can be numeric expressions. Like predicates, functions can have parameters, which are instantiated with objects (of suitable types) defined in the domain or problem. In the *numeric fragment* of PDDL, all functions are *number-valued*, so the value type of a function does not need to be specified. But it is also possible to declare the value type of a function explicitly. In Listing 7.3, we declare the numeric function `(total-cost)` to store the action costs in it. This special numeric function is without any argument. Next, the reserved word `number` is used to denote the numeric value type.

Listing 7.3 The functions in the domain file - Robotic inspection task

```
1
2       (:functions
3           ;; Actions cost
4           (total-cost) - number
5
6           ;; Operation time (optimization metric) [s]
7           ;(total-time) - number
8       )
```

In Listing 7.4 we write PDDL sentences for Spot and DJI respective allowed "high-level actions". We often refer to a parametrised action definition as an *action schema*. As we have mentioned, the values that parameters can assume correspond to objects declared in the PDDL problem definition. In our planning formulation of the robotic inspection domain, three actions make up the domain: `move-spot`, `inspect-spot` and `inspect-uav`. While the first action moves the mobile robot Spot in the specific environment from one inspection point to another, the other actions are used for performing inspections by Spot and UAV respectively.

With PDDL planning metrics, we introduce action costs in the PDDL problem, where actions may have *explicit costs*. For planning this means that there is obvious action to choose first. The specification of action costs in a domain requires three steps [Haslum et al., 2019] to include in our PDDL-encoding as follows:

1. declaring a special numeric function called `total-cost`, with no arguments; (Number-valued functions that are not *static* are often called *fluents*).

2. adding to the `:effect` of each action an expression that specifies how the action *increases* the total cost; and

3. adding a `:metric` specification to the problem definition.

Listing 7.4 The allowed actions in the domain file - Robotic inspection task

```
1
2       (:action move-spot
3           :parameters (?spot - spot
4                        ?uav - uav
5                        ?from-spot-pos - position
6                        ?to-spot-pos - position)
7
8           :precondition (and (attached ?uav ?spot)
9                              (spot-at ?spot ?from-spot-
                                   pos)
10                             (spot-reachable ?from-spot-
                                   pos ?to-spot-pos))
11
12          :effect (and (spot-at ?spot ?to-spot-pos)
13                       (uav-at ?uav ?to-spot-pos)
14                       (not (spot-at ?spot ?from-spot-pos
                              ))
15                       (increase (total-cost) 1))
16
17      )
18      (:action inspect-spot
19          :parameters (?spot - spot
20                       ?spot-pos - position
21                       ?inspect-pos - position)
22
23          :precondition (and (spot-inspectable ?spot-pos
                ?inspect-pos)
24                             (spot-at ?spot ?spot-pos)
25                             (not (inspected ?inspect-pos
                                   )))
26
27          ; This effect makes predicate (inspected ?
                inspect-pos) true
28          ; so that the robot won't return to this point
                later in the plan.
29          :effect (and (inspected ?inspect-pos)
30                       (increase (total-cost) 2))
31
32      )
33      (:action inspect-uav
34          :parameters (?dji - uav
35                       ?dji-pos - position
36                       ?inspect-pos - position)
```

```
37
38          :precondition (and (uav-inspectable ?dji-pos ?
                inspect-pos)
39                             (uav-at ?dji ?dji-pos)
40                             (not (inspected ?inspect-pos
                                   )))
41
42          :effect (and (inspected ?inspect-pos)
43                       (increase (total-cost) 1))
44      )
45  )
```

PDDL problem description for the ARI Inspection Task

The problem file describes the actual problem that needs to be solved by the PDDL-solver. Then, as a resulting output from the planner, we obtain the mission solution plan for Spot and DJI to perform the inspection tasks on the predefined inspection points. The complete problem instance for our planning task is found in Listing 9.2 in Appendix.

The following PDDL sentences in Listing 7.5 shows the first step towards a small example instance of the robotic inspection problem, with five inspection points `(ip0, ip1, ip2, ip3, ip4, ip5, ip6)`, as well as the two available robots: one ground mobile robot `spot1` and the aerial robot `dji1`. For Spot we also declare its possible locations `(sp0, sp1, sp2, sp3, sp4, sp5, sp6)` at the defined five inspecting tasks.

First, the problem refers to the *PDDL domain* of Listing 9.1, and then declares the *objects* we have mentioned. These objects are used when *grounding* the predicates and action schemas, as shown in Listing 7.6 . Usually, the task planner uses the *grounded predicates* to describe an initial state and a goal condition. Whereas, the *grounded actions* determine how a state can be transformed into a new state. In fact, a planner tries to find a *sequence of actions* that transforms the initial state into a state which satisfies the goal condition (Ferber and Seipp, 2022).

Listing 7.5 Objects defined in a small problem instance file - Robotic Inspection

```
1
2  ( define (problem inspect02)
3      (:domain inspections2)
4
5      (:objects
6          spot1 - spot
7          dji1 - uav
8          ip0 ip1 ip2 ip3 ip4 ip5 ip6 - position
9          sp0 sp1 sp2 sp3 sp4 sp5 sp6 - position
```

```
10      )
```

Afterwards, all predicates and functions are initialized, being the initial state of the problem. In the following Listing 7.6, We write PDDL sentences for the initial state, where Spot `spot1` start at point sp0 and come back to it at the end of the mission. At this starting point `sp0`, the quadrotor `dji1` should be attached to Spot, atop `spot1`. In addition, the cost function is initialized to zero. In this way, The `(:init ...)` section lists the facts that are "true" in the initial state of the problem instance. Every fact that is not explicitly mentioned is assumed to be false initially [Haslum et al., 2019].

Then, after initializing the states, positions and predicates, we need to define the topology of the problem, that is, the way in which constituent inspection points are interrelated or arranged. This means that we have to define all inspection points and their connections by grounding the reachability and visibility predicates of the inspection points in the problem. In terms of graph, we specify how inspection nodes are adjacent with each other, and it is also possible to generate additional *information of the route* between tasks nodes when it is necessary. Said that, the task-planner would be responsible of ordering the multiple tasks/goals placed in a inspection environment to minimize the distance travelled for the final plan.

Listing 7.6 Initialized states in a small problem instance file - Robotic Inspection

```
1
2       ;; The initialized states
3       (:init
4
5           ; The starting position of Spot1 and dji1
6           (spot-at spot1 sp0)
7           (uav-at dji1 sp0)
8           (attached dji1 spot1)
9
10          ;; initialise total cost to zero:
11          (= (total-cost) 0)
12
13
14      ;; Defining the topology of the problem:
15      ;; The way in which constituent parts are
            interrelated or arranged.
16          (spot-reachable sp0 sp3)
17          (spot-reachable sp0 sp5)
18
19          (spot-reachable sp1 sp2)
20          (spot-reachable sp1 sp3)
21          (spot-reachable sp1 sp4)
```

```
22
23          (spot-reachable sp2 sp1)
24          (spot-reachable sp2 sp5)
25
26          (spot-reachable sp3 sp0)
27          (spot-reachable sp3 sp1)
28          (spot-reachable sp3 sp4)
29
30          (spot-reachable sp4 sp1)
31          (spot-reachable sp4 sp3)
32          (spot-reachable sp4 sp5)
33
34          (spot-reachable sp5 sp0)
35          (spot-reachable sp5 sp2)
36          (spot-reachable sp5 sp4)
37
38
39          (spot-inspectable sp2 ip2)
40          (spot-inspectable sp3 ip3)
41          (spot-inspectable sp4 ip4)
42          (spot-inspectable sp5 ip5)
43
44          ; consider the case where ip6 is not reachable
                by Spot
45          (uav-inspectable sp5 ip6)
46
47      )
```

The `(:goal ...)` block contains the *conjunction of facts* that must be "true" for the mission goal to be achieved, i.e., a condition that must be satisfied at the end of a valid plan. As specified in Listing 7.7, the mission goal in our problem consists first of five predefined inspection points `(ip2, ip3, ip4, ip5, ip6)` that need to be visited for performing visual inspection, i.e., pictures acquisition or video live-streaming. Then, an *optimization metric* is defined as well, which tries to minimize the total action cost `(total-cost)`. The same could be done for minimizing the operation time stored in the function `(total-time)`. The related PDDL encoding regarding the operation time is found in the provided PDDL files in Appendix, however, these PDDL sentences are commented out to avoid eventual limitations and quirks when testing some "incompatible" PDDL-solver.

Listing 7.7 The goal defined in a small problem instance - Robotic Inspection

```
1
2       (:goal
```

```
3           (and (inspected ip2) (inspected ip3) (
                inspected ip4)
4               (inspected ip5) (inspected ip6))
5       )
6
7       (:metric
8           minimize (total-cost)
9       )
10  )
```

Recall that the `:metric ...` specification must be placed in the problem definition, not the domain, so that the expression can refer to objects declared in the problem. *The metric value of a plan is the value of the expression in the final state reached by the plan's execution*. In the case of sum of action costs, the value of the `(total-cost)` function is initially set to zero, and each action increases it by the cost of the action in the `:effect` fields; thus, its value at the end of plan execution is the sum of the costs of the actions in the plan [Haslum et al., 2019].

7.3 Candidate Solution Plan

For this thesis work, it would be sufficient to show the capabilities of the deliberative AI planning system for autonomous robots in industrial inspection domain and show the benefits of using PDDL as modelling language to solve the planning problem and find a possible solution plan, i.e., a sequence of actions like the one shown in Listing 7.8. In future work, it might be valuable to explore more PDDL-solvers and test multiple planners in order to evaluate the best plan quality by selecting a suited planner that will further improve the quality of the task planning in inspection domain. In this subsection, we are going to discuss how to evaluate the quality of the generated PDDL-plans and give a brief explanation of how the inspection task is modelled in PDDL, consisting of domain description in Listing 9.1 and problem description in Listing 9.2.

After implementing our "*baseline inspection model*", domain and problem files, using the on-line PDDL Editor [3] on "*planning.domains*" repository website and then running the provided off-the-shelf PDDL-solver an initial possible solution is computed. The textual representation of this plan is shown in Listing 7.8. This PDDL-model would represent a small inspection problem with two robots and five inspection points, where Spot start at point ip0 and come back to it at the end of the mission. This solution plan is then specified by a graph of Task Specification Tree (TST) using a Python script provided by WARA-PS framework. A fragment of this TST graph is illustrated in Figure 7.2.

[3] `http://editor.planning.domains`

In terms of modelling, because it is quite unlikely that the environment will satisfy all of the restrictive assumptions in Classical Planning, *a planning domain will almost never be a fully accurate model of the actor's environment*. Hence if a planning algorithm predicts that a plan $\pi = (a_1, a_2, a_3, a_4, a_5)$ will achieve a goal g, this does not ensure that π will achieve g when the actor performs the actions in π [Ghallab et al., 2016a]. In real world, when the robot tries to perform the plan π, several kinds of problem may occur, like execution failures, unexpected events, incorrect information, partial information or others.

Yet, having a PDDL-model of the robotic inspection domain, enables the usage of a wide variety of PDDL-planners of which we can then select a suitable one, based on the quality of the diverse generated plans. We should recall that this is the important *domain-independence* property of planners in PDDL modelling language, where multiple planners should be able to solve the same problem. *However, each solver has its own approach in solving the problem, where the quality of the plans can differ significantly* [Steenstra, 2019].

The following output in Listing 7.8 is a possible sequence of actions (mission plan) for the small robotic inspection domain generated by the off-the-shelf planner and editor on "planning.domains" website. The input to the planner is the PDDL-model presented in Listing 9.1 and 9.2. The actions that appear in the plan are names of "ground instances" of the action schemas in the domain.

Listing 7.8 The retrieved mission specification (planner output)

```
1
2       (move-spot spot1 dji1 sp0 sp3)
3       (inspect-spot spot1 sp3 ip3)
4       (move-spot spot1 dji1 sp3 sp4)
5       (inspect-spot spot1 sp4 ip4)
6       (move-spot spot1 dji1 sp4 sp5)
7       (inspect-spot spot1 sp5 ip5)
8       (inspect-uav dji1 sp5 ip6)
9       (move-spot spot1 dji1 sp5 sp2)
10      (inspect-spot spot1 sp2 ip2)
```

Quality Evaluation of Candidate Plans

In [Ghallab et al., 2016a] and [Haslum et al., 2019] planning is also defined as *the problem of finding a path that maps the initial state into a goal state that achieves a set of goals*. Moreover, as mentioned in the exploration domain in [Muñoz et al., 2016], the task planner is considered responsible of ordering the tasks/goals to *minimize the total distance travelled* for the final plan. Thus, the PDDL planner is in charge of "ordering" multiple tasks $\pi = (a_0, a_1, \ldots, a_n)$ placed in the environment and obtaining a plan.

When testing multiple PDDDL-planners and in different scenarios, a common practice in the field is to compare the quality of the diverse plans by looking at the *operation time* (the execution time employed by the robot to complete the plan) and/or the *total distance travelled* to achieve the goals (measured by the odometry sensors of the robot). The *search time* (computation time) to generate a plan by the PDDL-planners can be a third parameter to measure as well. A trade-off between the quality of a solution and its computing time is often desirable [Ghallab et al., 2016a]. In addition, although the PDDL-solver tries to find a valid plan based on the provided problem and domain file, the quality of this plan, or whether a plan is found at all, is highly dependent on the PDDL-solver that is being used, as explained in the survey domain work in [Steenstra, 2019].

In general, the objective is to evaluate the highest quality of the candidate plans in terms of runtime and/or distance travelled, i.e., plans with the lowest cost. Sometimes, it can be that some suited PDDL-solver would provide the highest quality, i.e., plans with the lowest cost, but on the other hand, it also takes more time to solve the problems. When the planning system is intended to plan "offline", the *computation time* is not that important. In fact, as stated in [Haslum et al., 2019], there is a clear difference in emphasis: in planning, the emphasis is on *efficiently finding a plan*, and in some cases on finding a plan of *good quality*.

Plan Quality Metrics in Terms of Operation Time To Specify the quality of a candidate plan in terms of operation time, the function term `total-time` need to be defined in the PDDL domain file, included in the section of `(:functions ...)`. Then, the quality of the plan can be evaluated by an *optimization metric*, defined in the PDDL problem file, which will try to minimize the operation time of the robot, expressed by the value of the variable `(total-time)` denoting "makespan". This is done by adding the following metric to the problem description: `(:metric minimize (total-time))`. As such, the lower the operation time when executing the generated plan, the higher the quality of this plan is. The quality is therefore the value of `(total-time)` after the last action of the plan provided by the PDDL-solver [Steenstra, 2019].

Further, plan quality in terms of operation time can be also evaluated using *durative actions* in temporal planning domains. In *temporal planning*, a fragment of PDDL where planning is more expressive than classical planning, actions are "durative" in nature [Haslum et al., 2019], compared to the classical "instantaneous" actions assumed in classical planning to transition the world from one state to another. Normally, the objective in temporal planning is to achieve the goal as early as possible, often referred to as the minimization of the plan makespan [Geffner and Bonet, 2016].

From the modelling perspective, we distinguish between these two action styles by using `(:durative-action ...)` to define the durative action in the domain description, where the use of durative actions is indicated in a planning domain by

adding `(:durative-actions ...)` to the list of `(:requirements ...)`. Note that the function term `(total-time)` is implicitly defined in temporal planning domains; it does not need to be declared in the `(:functions)` section. Moreover, a useful feature in the temporal setting is the specification of the *function terms* used to define the duration of actions, by writing expressions that evaluate this duration. These functions are usually "static", meaning functions that are not affected by any action.

In the same way, numeric functions are used in our PDDL-model to express *handling of action costs*, where the objective is to avoid certain actions a by assigning high cost $c(a)$ and adding the corresponding optimization metric `(:metric minimize (total-cost))` to the problem description.

It is important to remember that the PDDL-solver obviously needs to be able to meet the "requirements" defined in the PDDL domain file. The use of *functions* and *metric values* already excludes a significant amount of PDDL-solvers. For example, The PDDL-solver used in WARA-PS framework is a cost-optimal planner based on Fast Downward (FD) AI algorithm . Then, using the function term `(total-time)` may not be compatible with WARA-PS PDDL-solver to obtain a solution plan, because FD method can only handle one specific function `(total-cost)` [Steenstra, 2019].

7.4 Planner Selection

As we have mentioned, when modelling a given planning task in PDDL one needs to try multiple planners to choose the suited ones which can generate plans of high quality, because no classical planner "consistently" outperforms all others. To try other planners, it is possible to explore the archive of *planning competition web page* [4], where we can only try the deterministic tracks [Haslum et al., 2019]. For this reason, it is not easy to select an appropriate solver for a certain planning problem, due to the large amount of planners being developed since the first International Planning Competition (IPC). All of them exhibit different strengths and weaknesses and therefore no single planner is preferable to all others for all planning tasks [Ferber and Seipp, 2022].

In addition to the on-line editor and solver provided by "planning.domains" [5], that we have used to find our solution plan in Listing 7.8, it is also possible to use the *VAL tool suite* [6] from King's College London (KCL), which is another tool that can be valuable in the debugging process . It includes a *PDDL syntax checker* and a plan validator. A *plan validator* is a tool that takes as input a problem definition

[4] `http://icaps-conference.org/index.php/Main/Competitions`
[5] `https://planning.domains`
[6] `https://github.com/KCL-Planning/VAL`

and a plan, and determines if the plan solves the problem [Haslum et al., 2019]. An alternative implementation of a plan validator for PDDL is *INVAL* [7].

After exploring different planning literature similar to our inspection domain, including exploration, survey/coverage task, mine removal, logistics and other domains and examples [Geffner and Bonet, 2013; Haslum et al., 2019; Muñoz et al., 2016; Steenstra, 2019; Torreño et al., 2017; Ulam et al., 2007], we notice that there is a number of relevant PDDL-planners, that have shown good performance in recent International Planning Competition (IPC), such as Fast Downward (FD), LAMA, Metric-FF, POPF, LPG and OPTIC. Some individual results for some domains are obtained after experiments performed in the literature, demonstrating good and bad cases for the techniques implemented in the planners. Yet, each solver has its own approach in solving the problem. For detailed description the reader is referred to the respective literature. In the following we review briefly the main characteristics of these well-known PDDL-planners that could be "inspirational" for future work and to help readers in choosing suited planners and test them when modelling planning problems:

- *Fast Downward (FD)*: The cost-optimal planner Fast Downward has proven remarkably successful: It won the "classical" (i. e., propositional, non-optimising) track of the 4th International Planning Competition (IPC) at ICAPS 2004, following in the footsteps of planners such as FF and LPG [Helmert, 2006]. The FD planning method can only handle one specific function `(total-cost)` [Steenstra, 2019] (a cost-optimal planning method).

- *LAMA*: The PDDL-solver LAMA builds on the Fast Downward (FD) planning system, finding high quality plans. For that reason LAMA is often used to compare different PDDL-solvers based on quality. LAMA showed best performance among all planners in the *sequential satisficing track* of the International Planning Competition 2008. Overall, they find that using landmarks improves performance, whereas the incorporation of action costs into the heuristic estimators proves not to be beneficial [Richter and Westphal, 2010].

- *Metric-FF* [8]: It is considered, according to the IPC-3 results in the *numeric track*, one of the two currently most efficient *numeric planners*, together with LPG, both in terms of runtime and solution length [Hoffmann, 2003]. However, it does not consider the optimization metric properly, according to individual results for survey domain in [Steenstra, 2019].

- *POPF*: *Partial-order planning* is an intuitively attractive strategy, but has proved difficult to achieve efficiently. With POPF [Coles et al., 2010] they

[7] `https://github.com/patrikhaslum/INVAL`

[8] `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`

have shown that it is possible to achieve some of the advantages of partial-order planning in a *forward planning* framework using an expressive planning language, PDDL2.1 temporal models. POPF was the default PDDL-solver of the ROSPlan framework and a successful planner during the third IPC [Steenstra, 2019]. Before that, VHPOP (Younes and Simmons 2003) was the only recent partial-order planner to exploit the approach for temporal planning[Coles et al., 2010].

- *LPG*: LPG is a PDDL-solver that uses *local search* and *planning graphs*, called numerical planning graphs. An extended version of LPG handling complex PDDL2.1 domains involving numerical quantities and action durations [9] participated in the 3rd International Planning Competition, and it was awarded for "distinguished performance of the first order" [Gerevini and Serina, 2002].

- *OPTIC*: In 2012 an improved "temporal" PDDL-solver, called Optimizing Preferences and Time-dependent Costs (OPTIC), was developed. It uses a modified version of the POPF heuristic, and is able to comply with the newest PDDL versions. Instead of minimizing the number of actions, this planner focusses on the optimization metric. After an initial plan is found it repeats the planning procedure in order to improve on this optimization metric [Steenstra, 2019]. For more details, the reader is referred to [**optic**].

Portfolio-based Planner Selection using Machine Learning techniques

As we have discussed, it is hard to select the suitable planner from an extensive pool of planning systems, also called planners, available on International Planning Competition (IPC), for solving a given planning task. As a result, planning researchers has focused on exploiting diverse approaches for solving planning tasks. indeed, in the last few years machine learning techniques have been used by the planning community to select the suitable planner for a given PDDL planning task. Having a large collection of planners, it is often beneficial to aggregate multiple planners in a "portfolio" [Ferber and Seipp, 2022]. For an overview of *planning portfolios*, see [Vallati (2012)] and [Cenamor, de la Rosa, and Fernandez (2016)].

This interesting research direction in automated planning, using portfolio-based techniques for planner selection, and its connections to machine learning has recently become popular. There are relevant research works in this field like the online portfolio selector of *Delfi* using Deep Learning techniques [Katz et al., 2018] and the explainable planner selection of [Ferber and Seipp, 2022] using elementary Machine Learning techniques. It is worth giving attention to this important research by abstracting these two exciting works of portfolio-based planner selection in the following.

[9] `http://prometeo.ing.unibs.it/lpg`

How to create the *online portfolio planner* called *Delfi* is described in [Katz et al., 2018]. Delfi, the winner of the IPC 2018, exploits deep learning techniques to learn a model that can predict which of the planners in the portfolio can solve a given planning task within the imposed time and memory bounds. In particular, Delfi uses graphical representations of planning tasks and then turns them into images which allows exploiting existing tools for image convolution, i.e., using convolutional neural networks (CNN) to train the model.

The performance of Delfi showed that the learned model generalized well to the new benchmarks used in the competition. Delfi portfolio planner consists of a collection of 17 planners. With the exception of `SymBA*` (Torralba et al. 2014), the winner of the IPC 2014, included as-is in the collection of planners, all planners are based on a recent version of `Fast Downward`, a collection of cost-optimal planners.

In contrast to Delfi, the interesting work of [Ferber and Seipp, 2022] shows that complex black-box models (only the model and not the code is available) such as (graph) convolutional neural networks are not needed to learn strong planner selectors. Indeed, they train the most *elementary machine learning techniques* with understandable task features and obtain portfolios selectors for *optimal planning* that solve approximately as many tasks as the complex approaches based on neural networks, comparing with the strongest portfolio selector Delfi. In addition, their models have the advantage that they are *explainable/interpretable* and fast to train [Ferber and Seipp, 2022]. "Explainable" or "interpretable" means we can ask the model *why it selects a certain planner* and *which task features are actually important for the selection* (Cybenko 1989; Rudin 2019).

Ferber and Seipp use in their portfolio selector the same 17 optimal planners as Sievers et al. (2019a) and Ma et al. (2020): `SymBA*` (Torralba et al. 2017) and 16 `Fast Downward` configurations [Helmert, 2006]. The machine learning models trained in this selector are, in decreasing order of *interpretability*, linear regression, decision trees, random forests, and multi-layer perceptrons. Each model takes as input a vector $v \in \mathbb{R}^N$ containing the values of the input features.

In conclusion, authors in [Ferber and Seipp, 2022] stated that simple and explainable machine learning techniques like linear regression produce strong portfolio selectors. Their simple linear regression model solves roughly the same number of tasks as Delfi1, the state-of-the-art for planner selection.

8

Summary, Conclusions and Future Work

The goal of this thesis is to investigate two interesting problems related to the fields of Multiagent Networked Control and AI Robotics. Specifically, the project work is centred around the following two problems addressed in this thesis deploying two robotic platforms, a quadrotor UAV together with a ground mobile robot:

1. Rendezvous landing problem
2. AI automated task planning problem

In the following, we summarize the thesis by recalling the contributions, achievements and future directions regarding each part. Then, we review the benefits of Robotic Industrial Inspections in industry digitalization (Industry 4.0).

8.1 Rendezvous Landing Problem

We find the motivation of our first problem in designing a rendezvous landing algorithm that guides the quadrotor UAV (DJI M100) to track the ground mobile robot (the quadruped Spot robot) and enables it to land safely on this target robot at the end, after zeroing the relative distance between them. Thus, to pursue this goal a position-tracking controller is considered as one of the main contributions needed in this algorithm.

In this study, the algorithm is designed and implemented using a PID controller for position-tracking iterations in a simulated environment, composed of the ROS-based quadrotor UAV simulator (djisim) provided by WARA-PS framework, together with a new built Gazebo-ROS simulation model for a two-wheeled mobile robot.

During this work, we had the opportunity to "traverse" many literature related to this topic in robotics. Hence, To assess the effectiveness, a second approach using

MPC scheme is also investigated and presented in this thesis that aims at reinforcing the performance of our algorithm in suggested future work.

Summarising, we outline the main contributions, achievements and future directions work in the following:

- *Simulations*: This thesis develops a URDF-based two-wheeled robot (TWR) using Gazebo plugins to model the movement of Spot robot in the simulation experiments. Details of building this TWR and interacting with the WARA-PS framework software were covered in this work. ROS is used for control and message passing to the simulated robots involved in this Rendezvous mission.

  Moreover, WARA-PS simulator (djisim) allows us to develop our Rendezvous algorithm and test it. This djisim simulator is in fact a useful tool, especially for new pilots of quadrotor UAV. In particular, the use of this simulator can save much time in developing, testing and debugging our algorithm and flying missions in simulated experiments before the real UAV drone is launched in the real environment with the same commands and encoded scripts. In this way, our quadrotor can be flown in this simulated environment without any risk for real crashes.

- *Algorithm structure*: The algorithm makes use of the *ROS interface* (including services, subscribed topics, and published topics) and runs a four-dimensional PID *position-tracking controller* implemented in Python to control the four basic movements of the quadrotor UAV: thrust, roll, pitch and yaw. The objective of this controller is to "iteratively" compute and send correction commands to guide the UAV drone to track the target robot platform on the ground, as well as to accomplish the alignment of the acting unmanned vehicles in the horizontal plane and the landing operation at the end. This implementation runs in the outer loop of the quadrotor hierarchical (cascaded) control structure assumed in this work and in the djisim simulator as well.

- *Second approach using model-based control*: the theoretical development of this problem is carried out and presented in this study. Searching the literature, we can find that MPC is considered a robust control strategy for this type of problems for ensuring a soft and safe landing manoeuvre. Therefore, another approach for our algorithm based on non-linear MPC control is also studied and presented in this thesis, loosely based on several similar works presented in [Kamel et al., 2017b; Kamel et al., 2017a; Bereza et al., 2020; Persson et al., 2017; Persson and Wahlberg, 2019; Persson, 2021]. Its implementation might be an interesting direction in future work to improve the rendezvous landing performance especially in real-world flight experiments.

  Furthermore, to be able to perform a soft landing in real-world experiments in future work, the motion of the quadruped Spot (Q-UGV) can be tracked with

higher precision relative positioning using RTK-GNSS system. An overview about this system is discussed as well.

- *Evaluation*: Simple simulated experiments was performed and tested using the WARA-PS simulator (djisim) in which the quadrotor UAV could simulate the Rendezvous landing manoeuvre on a simulated two-wheeled moving a fixed distance in straight line path.

  As a result, the performance of this manoeuvre was evaluated through these simulated experiments by plotting the *error convergence graph* showing that the relative distance converged successfully towards zero over time.

  Having this Rendezvous landing simulation raises the questions of how well this simulation results can represent the real world and how easy the transition from simulation to real UAVs is. This transition questions will be in future work, in which a practical (experimental) evaluation on the real platforms will be conducted using the real hardware on DJI M100 and Spot robot.

8.2 AI Task Planning Problem

In the second part of this study, this thesis faces an interesting problem in AI and robotics. Its goal is to generate a high-level AI task plan composed of a sequence of actions to be executed by an *interconnected multi-agent system* in the automated robotic inspections (ARI) domain. In fact, the goal pursued in this entire work is to exploit AI planning algorithms in order to assign several tasks to different robots involved in an inspection mission. In summary,

- A PDDL-model is implemented to generate a *mission plan* in robotic inspection domain to achieve a set of inspection points. The goal is to navigate an industrial/construction site and performing visual inspection tasks at predefined inspection points (e.g., using pictures or camera pointing for video live-streaming).

- The retrieved mission solution plan is first obtained using an off-the-shelf PDDL-based planner. Then, it is translated to Task Specification Tree (TST) using WARA-PS framework, which decomposes/expands each high-level (abstract) action into low-level functional commands (concrete actions).

- In future work, the resulting candidate plan can be executed and tested on the real robotic platforms, Spot and DJI M100, in certain inspection scenario in construction/industrial environment.

8.3 Industry Digitalization: Benefits of Robotic Industrial Inspections

Robotic industrial inspection is an important component of the growing field of *Industry Digitalization*, also often referred to as *Industry 4.0*, i.e., fourth industrial revolution. Consolidated information about this Industry 4.0 technologies can be found on IBM website [1]. In summary, industry digitalization is a transformation that relies on the *interconnectivity* of digital technologies, devices, and processes. This digitalization ultimately aims at enabling *autonomous operation and decision-making systems* for manufacturing, process plants, logistics (e.g. inspection and maintenance tasks), transportation, energy systems, and other industry sectors with minimal human involvement. For completeness, it is worth noting that integration of *Operational Technology* (OT) and *Information Technology* (IT) is one of the main characteristics of digital transformation in smart factory that ensures that the data can flow (securely and reliably) from the factory floor (OT) to the top floor (IT) and vice versa.

In particular, with growing concern about workforce shortages and employee safety, industrial teams looks to make use of *automated robotic solutions* to improve operations in industrial environments. Nowadays, mobile robots (like Spot) act as roving internet of things (IoT) sensing platforms, performing remote and autonomous tasks like thermal inspection, gauge reading, acoustic imaging, and more.

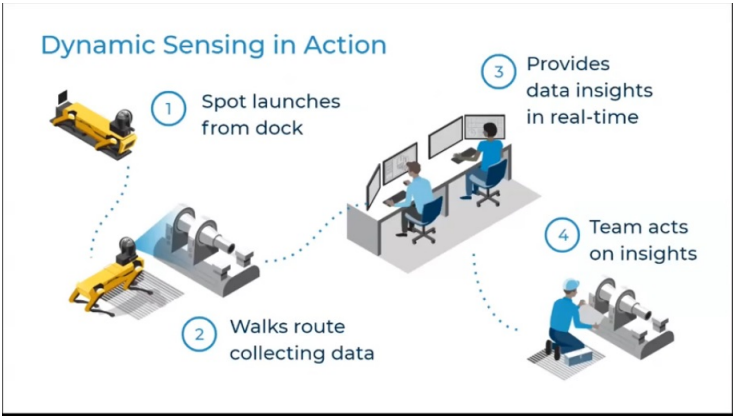


Figure 8.1 Application example about dynamic sensing in smart factory from the machine level to the cloud (from OT to IT). Real-time data collected (using mobile robots like Spot) from embedded sensors, devices and interconnected machinery on the factory floor can produce a significant amount of big data for industry. (Figure courtesy: Boston Dynamics)

[1] https://www.ibm.com/topics/industry-4-0#:~:text=Industry%204.0%20is%20revolutionizing%20the,facilities%20and%20throughout%20their%20operations.

According to *Boston Dynamics* study, automating inspections with agile mobile robots can bring many benefits to industry by improving (1) safety, (2) efficiency and (3) predictability. In fact, robotic inspections remove personnel from dangerous, electrified, radioactive, explosive environments by collecting data remotely and identifying risks without the human exposure to hazardous conditions. Then, efficiency is also improved by reducing time to inspection and improving uptime of the machinery, as well as by redistributing labor to focus on critical decision making work areas.

Furthermore, these smart robotic platforms allow collecting the critical data at the right time to power effective *predictive maintenance* programs. In fact, over the last years industries have integrated new technologies into their production facilities and throughout their operations, including *Internet of Things (IoT)*, *cloud computing and analytics*, and *AI* and *machine learning*. In this way, sensors can be triggered autonomously and data can be exported to analysis software, as well as shared across other components in the enterprise software stack, where the predictability can be improved using the obtained boosted data collection (100x) .

In general, according to IBM, using high-tech IoT devices in smart factories leads to higher productivity and improved quality. Replacing manual inspection business models with AI-powered visual insights reduces manufacturing errors and saves money and time.

9

Appendix

9.1 Quadruped UGV: Spot Platform

In this section, we give a brief overview of the agile legged robot Spot characteristics and its functionalities. After attending different valuable technical webinars organized by Boston Dynamics, through the technical team composed of chris Bentzel, Bryce Beddard, Bob Ochiai, Caleb Sylvester, Vatche Arabian, Marco da Silva, Devin Billings, Kathleen Brandes and Piro Lera, we can summarize the main characteristics of this UGV quaruped as follows:

- *Easy to operate*: with Spot it is possible to perform manual and autonomous operation. Spot can also itself makes decisions on foot and body placement. This legged robot is designed to handle extreme terrains and capture data in unstructured and dangerous environments. In addition, it enjoys of 360 degrees perception and dynamic stabilization.

- *Industry friendly*: Spot is described as an "agile mobile robot". In fact, it can navigate over stairs and keep distance of 30 cm from objects, as well as can avoid obstacles. It is water and dust resistant, with the standard Ingress Protection code IP54.

- *Customizable*: Spot is able to carry sensors up to 30 lbs /14 kg (payload capacity). Spot platform provides a powerful Python Software Development Kit (SDK) [1] and flexible application ecosystem using a layered Application Programming Interface (API).

These characteristics give this flexible platform the ability to automate sensing and inspection, and consequently to be operational in different industrial facilities for performing fundamental inspection tasks including: remote inspection, thermal inspection, leak detection, radiation detection, gauge reading, gas detection, noise anomaly detection and digital twin creation.

[1] Spot SDK: `https://dev.bostondynamics.com`

Spot API Architecture

The Spot SDK includes APIs, client libraries, and examples that support the development of *autonomous navigation behaviors* for the Spot robot.

The provided API lets applications control Spot, read sensor information and integrate with payloads via `gRPC`. The Spot API follows a *client-server model*, where client applications communicate to services running on Spot over a network connection [2].

As illustrated in Figure 9.1, Spot API consists of the following layers: Data, Autonomy, Movement and Base. This flexible layered API can be used to develop tailor-made payloads, controls and behaviors. For instance, we can add cameras and sensors, autonomously trigger sensors and send data to analysis software, as well as we can create unique controls and autonomy systems for the robot and sensors. In this section, we outline the principal "ingredients" of these layers. For more detailed information the reader is referred to the documentation available on the Boston Dynamics developer site.

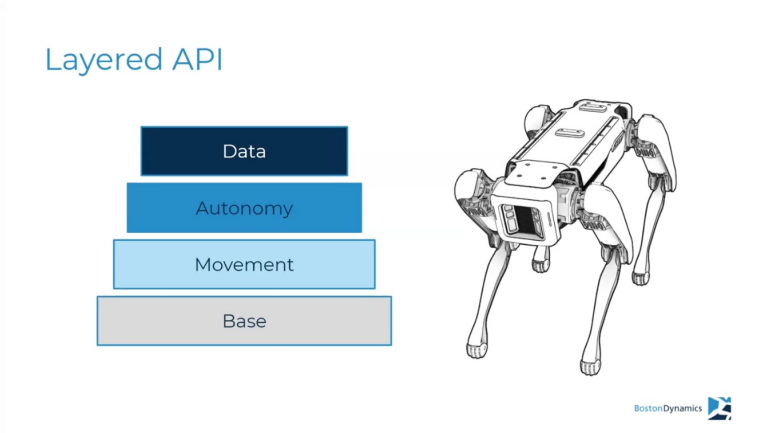


Figure 9.1 Spot API consists of the layers: Data, Autonomy, Movement and Base layers. This flexible layered API can be used to develop tailor-made payloads, controls and behaviors. Courtesy: Boston Dynamics

Base API: This base layer includes the network services and the Boston Dynamics Spot SDK:

- *Network services*:
  - built on top of a number of standards existing in the industry like `gRPC`, `WebRTC`, `HTTP/2`.

[2] Spot API Protocol: `https://dev.bostondynamics.com/docs/protos/readme`

  - Secure: encrypted, authenticated, authorized
  - Flexible Computation: on-robot, on-network clients, cloud hosted

- *Software Development Kit (SDK)*:
  - Python Library
  - Documentation
  - Examples
  - Customer Support

Movement API: This layer includes Robot state, odometry, sensor data, terrain and environment. An SDK example regarding Base and Movement layers in the SDK is `hello_spot`, which is an introductory programming example demonstrating: Stand up, strike a pose, stand tall capture an image, sit down.

Autonomy API: This layer includes autonomous rounds and readings with `Autowalk` feature [3] and builds on *Navigation* and *Missions* portions of autonomy layer. Collectively, this service is referred to as `GraphNav`. Moreover, by leveraging GraphNav and Mission APIs we can also record a Graph and/or Mission, replay portions of a Graph and create custom missions. SDK examples about Autonomy layer in the SDK are `graph_nav_command_line`, `build_mission` and `mission_recorder`.

- GraphNav:
  - Map (or graph) of complex spaces
  - Localize where Spot is
  - Navigate across large areas
- Missions:
  - Behavior trees for complex sequences of actions
  - Runs without client involvement
  - Supports complex behavior

[3] `https://support.bostondynamics.com/s/article/Getting-Started-with-Autowalk`

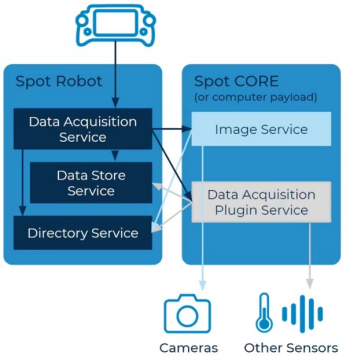


Figure 9.2 Spot Data acquisition (DAQ) architecture used to create a data acquisition plugin service and run the service to communicate with the data acquisition service on-robot. DAQ functionality is a powerful tool to use in "photogrammetry applications" in Agriculture, Building & Construction, Geospatial and other areas. Courtesy: Boston Dynamics

Data layer: This layer includes visual, audio and analysis. A related SDK example is `data_acquisition_service`, that demonstrates how to create a data acquisition plugin service and run the service to communicate with the data acquisition service on-robot.

- Data Acquisition (DAQ)
  - Integrate data capture devices
  - Display during teleop
  - Acquire data during Autowalk
- Network Compute Bridge
  - Computer vision plugin
  - Configure TensorFlow models to analyze sensor data
  - Use for analysis and control

(a) **(b)**

Figure 9.3 (a) Flexible communications architecture: This provide flexible communications and clients have options to talk to payload computer using the robot's Wi-Fi or any IP addressable device on the payload subnet. (b) Spot custom payloads applications example: NASA JPL, NeBula-Spot together with TEAM CoSTAR Autonomous rover to explore martian-like Caves. Courtesy: Boston Dynamics and NASA

Developing Payloads: Integrating and connecting with Spot API

In this section, an overview of how to develop a payload or integrate a sensor and connect it to the Spot API software is given by a useful webinar consisting of two parts. In the intro session, organised by Boston Dynamics and presented by Marco da Silva and Devin Billings, they explained the task of "developing a payload" in which they introduce the steps to interfacing with Spot mechanically and electrically [4].

From the content of this webinar, it is worth to highlight that payloads can talk to the robot directly through the API or provide the API with additional sensor data like images. In addition, payloads can talk to each other, and on robot port forwarding to talk to our rear or front payload through the robot, shown in Figure 9.3a.

In the second part of Developing a payload part, they address the *integrating of payloads with Spot software*, introduced by Kathleen Brandes and Piro Lera [5]. Thus, having a payload or sensor integrated with the robot we look to connect it to Spot's API, using the example `payloads.py` in Spot SDK.

From this second session of the webinar, we highlight the fact that we can talk to Spot robot by using `ping 192.168.80.3` after installing the Spot SDK on our

[4] `https://www.bostondynamics.com/resources/webinar/developing-payload-part-i`
[5] `https://www.bostondynamics.com/spot/resources/developing-a-payload-part-ii`

computer and opening it. Moreover, we can get access to the robot webpage by digiting the same IP address in the browser.

In the API a payload can be either a sensor like a webcam, a compute unit like Spot CORE or a developer laptop. Here, we summarize the generalized steps to follow when connecting some payload to Spot API, as stated in the webinar:

1. Installed *Spot SDK* on development machine
2. Wrote code to register a payload and `ImageService`
3. Tested and debugged on development machine
4. Deployed to *Spot CORE* which is a compute payload for Spot
5. Tested and debugged on Spot CORE
6. Used logs and payload faults to check errors

Additionally, to exercise and gain better understanding, we choose to review how to make use of the `payload.py` example in Spot SDK in order to create and register a *developer laptop* as a payload. In fact, for testing from the developer laptop, we want to register it as a *wireless payload* so that it can communicate to the services on the robot without affecting the gait because we actually don't add any weight there.

- Open `payloads.py` example in Spot SDK
- Look at `payload.GUID`, `payload_secret`, and `payload.name`
- Run this example in the terminal by writing:
  `python3 payloads.py -username tester -password password1234 192.168.80.3`
- Lastly, we need to *authorize* this new created and registered payload on the robot webpage, otherwise this laptop will not be able to communicate with different services on Spot as a payload.

AI-Based Predictive Maintenance using Spot

According to Boston Dynamics follow-up, Spot is a valuable tool for automating routine inspection tasks so our human capital can focus on more critical operations.

To provide some insight, there is a blog post [6], provided by Boston Dynamics, that describes how Spot can enable the switch from *reactive* to "*AI-based predictive maintenance*". Additionally, they provide a white paper [7] that dives deeper into one

[6] `https://www.bostondynamics.com/resources/blog/new-approach-predictive-maintenance-challe`
[7] `https://resources.bostondynamics.com/hubfs/Content%20Files/Boston_Dynamics_Thermal_Inspection_Whitepaper.pdf`

of the many inspection types that Spot can perform – *thermal inspections* to identify anomalous conditions before critical failure occurs.

Specifically, as explained in the blog, agile mobile robots like Spot act as *roving IoT sensing platforms*, performing autonomous rounds and readings to collect the right data at the right time for effective predictive maintenance. Equipped with sensing payloads, a single robot can reliably and repeatedly collect multiple types of data from assets in different locations around our facility, processing the data on the robot in real-time or integrating directly with an *Enterprise Asset Management (EAM)* system to streamline predictive maintenance processes.

9.2 Quadrotor UAV: DJI Matrice 100 Platform

In this section, we start to describe the WARA-PS DJIM100 quadrotor platform. The DJI Matrice 100 is a quadcopter used in WARA-PS [WARA-PS, 2024] to collect data and perform experiments in the field. It has been used to make autonomous detections and avoid people during low altitude flights. By mounting cameras and sensors on the quadcopters, they can be used to *livestream video footage* to increase *situation awareness* and give an overview of accident sites during search and rescue missions. It can also be used to drop items such as first aids kits near people through the autonomous detection ability. Abilities and research areas of this platform can also include:

- Swarm behaviour and task delegation using the delegation framework
- Collaboration in a heterogenous system
- Search area
- Cooperative landing

Code is easily integrated using the ROS environment. This platform has an onboard **Intel NUC Mini PC** [8], which is fully complete and ready to work out of the box and allows **high-level control** and **estimation** to be done in flight by the aerial robot. In addition, the platform has onboard **IMU**, built into the **main controller MC**, which can provide feedback of acceleration and angular velocities (Valavanis, 2015, p378). DJI Matric 100 [9] spans 100cm from the tip of one propeller to the tip of the opposite propeller, has a height of 8cm, and has a weight of about 2400g including a battery. Full specification of the Matric 100 is available at the DJI site [10].

[8] https://www.intel.com/content/www/us/en/products/details/nuc.html
[9] https://portal.waraps.org/technical-specifications-of-dji-matrice-100/
[10] https://www.dji.com/se/matrice100/info

Figure 9.4 A DJI Matrice 100 (UAV) quadrotor with iPad. The DJI Matrice 100 platform is aimed at researchers and developers and is characterized by good access to the flight functionalities using the provided SDK. Its open construction provides possibilities of easy integration of additional components such as sensors or computational power. Courtesy: [WARA-PS, 2024]

Sensing and Computing

In this section, it is worthwhile to review the equipment and sensors of WARA-PS UAV, DJI Matrice 100, involved in this cooperative landing project as described on the WARA-PS portal [11].

(a) **(b)**

Figure 9.5 (a) Intel NUC computer in lightweight housing mounted on-board. (b) Zenmuse Z3 color camera with gimbal. (Courtesy: [WARA-PS, 2024])

Intel NUC Computer This platform has an onboard Intel NUC Mini PC which is fully complete and ready to work out of the box. It technical specifications can be summarized by the following points:

- The Intel NUC computer is a small-factor PC. It uses a powerful 7th generation i7 CPU (Kaby Lake) which contains 4 physical cores and 4 additional threads through the use of hyper-threading.

[11] https://portal.waraps.org/technical-specifications-of-dji-matrice-100/

- It is equipped with 16GB of RAM and 500GB solid state drive.

- The NUC is the host for the **ROS-based software system** and interfaces with the platform's autopilot using RS-232 to USB FTDI adapter.

DJI Zenmuse Cameras To the DJI Matrice platform, it is also easy to integrate the Zenmuse thermal or color cameras with gimbal [12] characterized by the following:

- The Zenmuse Z3 provides high resolution video streams at high rates.

- The gimbal provides state-of-the-art mechanical stabilization.

- ROS-based driver for grabbing images using the HDMI/USB frame grabber is available.

- Additionally, the video and photo data can be recorded on an SD-Card.

Communication and Multimedia using WiFi and HDMI

At the ground station, it is possible to communicate to the platform and analyse the capture images/videos using the following equipment, as illustrated in Figure 9.6:

- USB Capture HDMI Gen 2 from Magewell [13]: Universal HDMI frame grabber which interfaces with the host PC using USB3.0 interface can be connected to the DJI remote controller and used to capture live video on the ground station computer.

- Ubiquity Bullet Titanium M5HP WiFi access point [14]: is used to provide WiFi connectivity in the field. The WiFi base station can be operated using the enclosed 6000 mAh LiPo battery (full day operation) or using the main power (230V) where available.

[12] `https://www.dji.com/zenmuse-xt`
[13] `http://www.magewell.com/usb-capture-hdmi`
[14] `https://www.ubnt.com/airmax/bulletm/`

(a) **(b)**

Figure 9.6 (a) DJIM100 Ground Station Equipment. Left: Bullet Titanium M5HP Access point. Middle: Ubiquity complete base station to provide WiFi connectivity. Right: Laptop with screen sun protection. (b) Universal HDMI frame grabber which interfaces with the host PC using USB3.0 interface. (courtesy: [WARA-PS, 2024])

Hardware Interconnections

In this section, we review the types of connections used between the hardware components in the DJI M100 platform, as illustrated in Figure 9.7 below.

The **airborne part** of the system consists of the DJI Matrice platform (including its autopilot), the optional DJI Manifold and Guidance systems, the NUC computer and a suite of possible sensors. The CAN bus interface can only be used for DJI components, i.e. additional sensors cannot use this bus.

The **ground equipment** includes the Remote controller used by the backup pilot. It is equipped with the DJI Go app running on an iPad tablet. It is used for monitoring flight telemetry data as well as a possible live video feed. Additionally, the WiFi base station is present and can be used to connect any number of computers. One of which runs the main ground station software (ROS-based).

If the images are grabbed on-board using the DJI Manifold computer, the HDMI connection to the ground computer is not required (but still possible).

Communication between the airborne and the ground systems is realized using two technologies:

1. Standard WiFi 802.11a/n at 5GHz

2. DJI proprietary link at 2.4 GHz and 5 GHz.

From the experience so far the two types of links do not interfere with each other, as researchers stated in [WARA-PS, 2024].

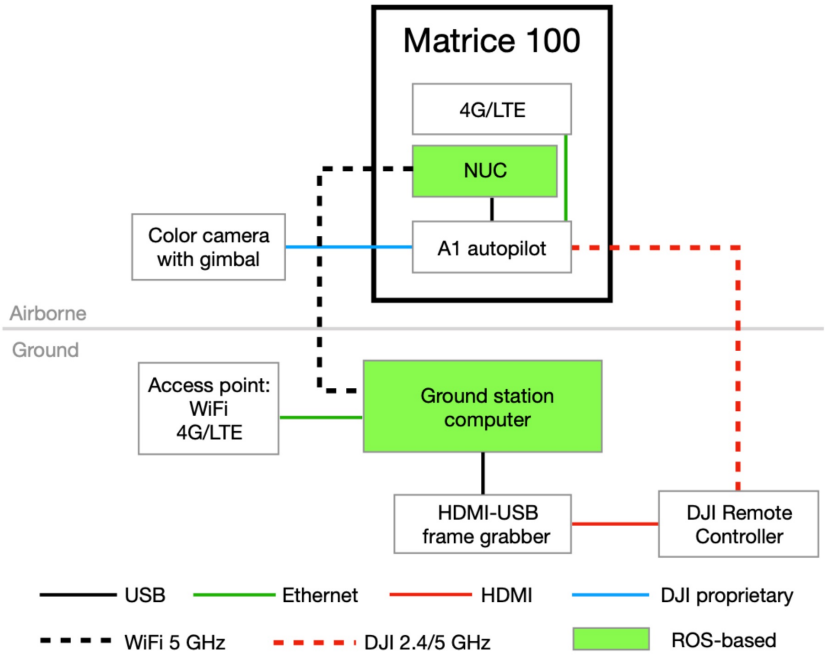


Figure 9.7 Schematic of the airborne and ground hardware components and their interconnections including wireless links. Additionally, one of the main open source software for our project is the DJI Software Development Kits (SDK) in green. The SDKs are designed to allow "registered developers" to fully access the capabilities of the quadrotor by running their own applications on the drone. The two SDKs that can be used are the **Onboard-SDK (OSDK)** for the onboard NUC computer which is mounted on the DJI platform and **ROS-SDK** which is used for the base station. The ROS SDK works "in conjunction" with the OSDK and allows developers to access all of ROS capabilities through its messages and services. (Image Courtesy: [WARA-PS, 2024])

DJIM100 Platform Initial Setup

Initial setup is required in order to operate the DJIM100 platform. Specifically, it is necessary to create the following two accounts on Apple and DJI:

1. **Apple Id Account**: to setup the iPad and download the required *DJI Go* app we need to register at the following Apple web-site: `https://appleid.apple.com/account`
2. **DJI Developer Account**: to be able to use the *Onboard SDK* we need to register at the following DJI web-site: `https://developer.dji.com/`

In this way, the DJI developer account will be needed for these important issues:

- to obtain the "developement key" (see: `key_you_get_when_you_register` in the LRS developer environment [15] indications).
- Additionally, it is required to login to the same developer account in the DJI Go app in order to "activate the platform".

9.3 Domain File - Robotic Inspection Task

Listing 9.1 The domain file for the robotic inspection problem

```
1
2  (define (domain inspections3)
3
4      (:requirements :strips :typing :fluents :action-
           costs)
5
6      (:types
7          ;; declare two different object types: robots
               and the their locations
8          ;; as well as the locations of the inspection
               points
9          spot uav position - object
10     )
11
12     (:predicates
13         ;; Reachability of locations for Spot robot
14         (spot-reachable ?from-spot-pos ?to-spot-pos -
               position)
15
16         ;; Visibility: when objective/inspection point
               is visible for Spot
17         (spot-inspectable ?spot-pos ?inspect-pos -
               position)
18
19         ;; Sometimes some inspection points are
               partially or completely non-inspectable
               with Spot.
20         ;; That can be caused for example by high/
               vertical and/or occult construction which
               is
```

[15] https://gitlab.liu.se/lrs/lrs_devenv_common

```
21          ;; difficult-to-access for Spot robot. Thus,
                UAV should intervene to inspect instead of
                Spot.
22          ;; uav-inspectable(from-pos, inspect-pos) with
                the "intended semantics" that it can take-
                off,
23          ;; fly, inspect, fly-back and land as one uav-
                do-inspection action.
24          (uav-inspectable ?spot-pos ?inspect-pos -
                position)
25
26          ;(spot-at ?spot-pos - position)
27          (spot-at ?spot - spot ?spot-pos - position)
28
29          ;; predicate to express the goal:
30          (inspected ?inspect-pos - position)
31
32          (uav-at ?uav - uav ?inspect-pos - position)


33
34          ;; state when uav is at the top of Spot
35          (attached ?uav - uav ?spot - spot)
36
37      )
38
39      (:functions
40          (total-cost) - number
41
42          ; Operation time (optimization metric) [s], the
                total time consumed for moving
43          ;(total-time) - number
44          ;(time ?from ?to - position)  ; the duration of
                one specific transit/move
45      )
46
47      ; Action schema (parameterised action definition)
            of "move-spot" action
48      ; NOTE: The values that parameters can assume
            correspond to objects declared
49      ; in the PDDL problem definition.
50      (:action move-spot
51          :parameters (?spot - spot
52                       ?uav - uav
```

```
53                      ?from-spot-pos - position
54                      ?to-spot-pos - position)
55
56          :precondition (and
57              (attached ?uav ?spot)
58                        (spot-at ?spot ?from-spot-pos)
59                        (spot-reachable ?from-spot-pos
                              ?to-spot-pos))
60
61          :effect (and (spot-at ?spot ?to-spot-pos)
62                      (uav-at ?uav ?to-spot-pos)
63                      (not (spot-at ?spot ?from-spot-pos
                            ))
64                      (increase (total-cost) 1))
65                      ;; Increase the total time by the
                            transit/move duration
66                      ;(increase (total-time) (time ?
                            from ?to))
67
68      )
69      (:action inspect-spot
70          :parameters (?spot - spot
71                      ?spot-pos - position
72                      ?inspect-pos - position)
73
74          :precondition (and
75                        (spot-inspectable ?spot-pos ?
                              inspect-pos)
76                        (spot-at ?spot ?spot-pos)
77                        (not (inspected ?inspect-pos)))
78
79          ; This effect makes predicate (inspected ?
              inspect-pos) true
80          ; so that the robot won't return to this point
              later in the plan.
81          :effect (and (inspected ?inspect-pos)
82                      (increase (total-cost) 2))
83
84      )
85      (:action inspect-uav
86          :parameters (?dji - uav
87                      ?dji-pos - position
88                      ?inspect-pos - position)
```

```
89
90          :precondition (and (uav-inspectable ?dji-pos ?
                inspect-pos)
91                             (uav-at ?dji ?dji-pos)
92                             (not (inspected ?inspect-pos
                                   )))
93                             ;(not (spot-reachable ?spot-
                                   pos ?inspect-pos))
94                             ;(not (spot-inspectable ?
                                   spot-pos ?inspect-pos)))
95
96
97          :effect (and (inspected ?inspect-pos)
98                       (increase (total-cost) 1))
99      )
100 )
```

9.4 Problem File - Robotic Inspection Task

Listing 9.2 A small problem instance file of the robotic inspection problem

```
1
2  ( define (problem inspect03)
3      (:domain inspections3)
4
5      (:objects
6          spot1 - spot
7          dji1 - uav
8          sp0 sp1 sp2 sp3 sp4 sp5 sp6 - position
9          ip0 ip1 ip2 ip3 ip4 ip5 ip6 - position
10
11     )
12
13     ; The :init section lists the facts that are "true"
           in the initial state
14     (:init
15         ;; An initial state: The state of the world
               that we start in
16
17         ; The starting position of Spot1 and dji1
18         (spot-at spot1 sp0)
```

```
19          (uav-at dji1 sp0)
20          (attached dji1 spot1)
21
22          ;; initialise total cost to zero:
23          (= (total-cost) 0)
24          ;; Initialize total transition/moving time
25          ;(= (total-time) 0)
26
27          ;; Defining the topology of the problem: The
                way in which constituent parts
28          ;; are interrelated or arranged, i.e.,
                connections between inspection points
                reachable for Spot.
29          ;; This means that we have to define all nodes/
                inspection points and their connections.
30          ;; In other words, what nodes are adjacent with
                 each other. In this way, we can generate
31          ;; the information of the route between tasks/
                goals.
32
33          ;; The task-planner is responsible of ordering
                the tasks/goals to minimize the distance
34          ;; travelled for the final plan.
35
36          (spot-reachable sp0 sp3)
37          ;(= (time sp0 sp3) 2)        ;; Initialize the
                duration
38                                      ;; for transiting/
                                           moving between
                                           insp. points
39          (spot-reachable sp0 sp5)
40          ;(= (time sp0 sp5) 1)
41
42          (spot-reachable sp1 sp2)
43          ;(= (time sp1 sp2) 2)
44          (spot-reachable sp1 sp3)
45          ;(= (time sp1 sp3) 1)
46          (spot-reachable sp1 sp4)
47          ;(= (time sp1 sp4) 2)
48
49          (spot-reachable sp2 sp1)
50          ;(= (time sp2 sp1) 1)
51          (spot-reachable sp2 sp5)
```

```
52          ;(= (time sp2 sp5) 2)
53
54          (spot-reachable sp3 sp0)
55          ;(= (time sp3 sp0) 1)
56          (spot-reachable sp3 sp1)
57          ;(= (time sp3 sp1) 2)
58          (spot-reachable sp3 sp4)
59          ;(= (time sp3 sp4) 1)
60
61          (spot-reachable sp4 sp1)
62          ;(= (time sp4 sp1) 2)
63          (spot-reachable sp4 sp3)
64          ;(= (time sp4 sp3) 1)
65          (spot-reachable sp4 sp5)
66          ;(= (time sp4 sp5) 2)
67
68          (spot-reachable sp5 sp0)
69          ;(= (time sp5 sp0) 1)
70          (spot-reachable sp5 sp2)
71          ;(= (time sp5 sp2) 2)
72          (spot-reachable sp5 sp4)
73          ;(= (time sp5 sp4) 1)
74
75          ;(not(spot-inspectable sp1 ip1)) ; we don't
                need negated literal,
76                                          ; it is false
                                                by default
                                                ,
77                                          ; have only a
                                                list of
                                                facts that
                                                are TRUE
78          (spot-inspectable sp2 ip2)
79          (spot-inspectable sp3 ip3)
80          (spot-inspectable sp4 ip4)
81          (spot-inspectable sp5 ip5)
82
83          ; Let's consider the case where ip6 is not
                reachable by Spot
84          (uav-inspectable sp5 ip6)
85
86      )
87
```

```
88      ; The :goal section lists the "conjunction of facts
            " that must be
89      ; "true" for the goal to be achieved.
90      (:goal
91          (and (inspected ip2) (inspected ip3) (
                inspected ip4)
92              (inspected ip5) (inspected ip6))
93
94      )
95
96      (:metric
97          minimize (total-cost)
98      )
99
100     ;; Minimize the transit/move time
101     ;(:metric minimize (total-time))  ;; Minimize the
            transit/move time
102 )
```

9.5 Implementation of nonlinear MPC using ACADO

ACADO provides an example on how to formulate a nonlinear OCP in C++. The tutorial in ACADO Toolkit [16] explains how to setup a basic MPC controller. As an example, a simple actively damped **quarter car model** is considered and the corresponding code is illustrated in Listing 9.3.

Mathematical Formulation of Model Predictive Control

Problem formulation: Let x denote the states, u the control input, p a time-constant parameter, and T the time horizon of an MPC optimization problem. We are interested in **tracking MPC problems**, which are of the general form:

$$
\begin{aligned}
\min_{x(\cdot),u(\cdot),p} \quad & \int_{t_0}^{t_0+T} \|h(t,x(t),u(t),p)-\eta(t)\|_Q^2 \,\mathrm{d}t + \|m(x(t_0+T),p,t_0+T)-\mu\|_P^2 \\
\text{subject to:} \quad & x(t_0) = x_0 \\
\forall t \in [t_0,t_0+T]: \quad & 0 = f(t,x(t),\dot{x}(t),u(t),p) \\
\forall t \in [t_0,t_0+T]: \quad & 0 \geq s(t,x(t),u(t),p) \\
& 0 = r(x(t_0+T),p,t_0+T)
\end{aligned}
$$

[16] https://acado.sourceforge.net/doc/html/d4/d26/example_013.html

Here, the function f represents the **model equations**, s the **path constraints** and r the **terminal constraints**. Note that in the online context, the above problem must be solved iteratively for changing x_0 and t_0. Moreover, we assume here that the **objective** is given in "least square form". Most of the tracking problems that arise in practice can be formulated in this form with η and μ denoting the tracking and terminal reference.

Implementation of an MPC Controller for a Quarter Car

The following piece of code shows how to implement an MPC controller based on this quarter car model. It comprises six main steps:

1. Introducing all variables and constants.
2. Setting up the quarter car *ODE model*.
3. *Setting up a least-squares objective function* by defining the five components of the measurement function h and an appropriate weighting matrix.
4. *Defining a complete optimal control problem (OCP)* comprising the dynamic model, the objective function as well as constraints on the input.
5. *Setting up a RealTimeAlgorithm* defined by the OCP to be solved at each sampling instant together with a sampling time specifying the time lag between two sampling instants. Moreover, several options can be set and plot windows flushed.
6. *Setting up a Controller by specifying a control law*, i.e. the real-time algorithm solving our OCP in this case, and a reference trajectory to be tracked. In this example, the reference trajectory is read from a file where the value of all components are defined over time. (Note that the reference trajectory can be left away when calling the `Controller` constructor which is equivalent to all entries zero over the whole simulation horizon.)

Listing 9.3 MPC controller based on quarter car model

```
1
2  #include <acado_toolkit.hpp>
3  #include <include/acado_gnuplot/gnuplot_window.hpp>
4
5  int main( )
6  {
7      USING_NAMESPACE_ACADO
8
9
10     // INTRODUCE THE VARIABLES:
```

```
11      // ------------------------
12      DifferentialState xB;
13      DifferentialState xW;
14      DifferentialState vB;
15      DifferentialState vW;
16
17      Control F;
18      Disturbance R;
19
20      double mB = 350.0;
21      double mW = 50.0;
22      double kS = 20000.0;
23      double kT = 200000.0;
24
25
26      // DEFINE A DIFFERENTIAL EQUATION:
27      // -------------------------------
28      DifferentialEquation f;
29
30      f << dot(xB) == vB;
31      f << dot(xW) == vW;
32      f << dot(vB) == ( -kS*xB + kS*xW + F ) / mB;
33      f << dot(vW) == ( -kT*xB - (kT+kS)*xW + kT*R - F )
            / mW;
34
35
36      // DEFINE LEAST SQUARE FUNCTION:
37      // -----------------------------
38      Function h;
39
40      h << xB;
41      h << xW;
42      h << vB;
43      h << vW;
44      h << F;
45
46      // LSQ coefficient matrix
47      Matrix Q(5,5);
48      Q(0,0) = 10.0;
49      Q(1,1) = 10.0;
50      Q(2,2) = 1.0;
51      Q(3,3) = 1.0;
52      Q(4,4) = 1.0e-8;
```

```
53
54      // Reference
55      Vector r(5);
56      r.setAll( 0.0 );
57
58
59      // DEFINE AN OPTIMAL CONTROL PROBLEM:
60      // ----------------------------------
61      const double tStart = 0.0;
62      const double tEnd   = 1.0;
63
64      OCP ocp( tStart, tEnd, 20 );
65
66      ocp.minimizeLSQ( Q, h, r );
67
68      ocp.subjectTo( f );
69
70      ocp.subjectTo( -200.0 <= F <= 200.0 );
71      ocp.subjectTo( R == 0.0 );
72
73
74      // SETTING UP THE REAL-TIME ALGORITHM:
75      // -----------------------------------
76      RealTimeAlgorithm alg( ocp,0.025 );
77      alg.set( MAX_NUM_ITERATIONS, 1 );
78      alg.set( PLOT_RESOLUTION, MEDIUM );
79
80      GnuplotWindow window;
81          window.addSubplot( xB, "Body Position [m]" );
82          window.addSubplot( xW, "Wheel Position [m]" );
83          window.addSubplot( vB, "Body Velocity [m/s]" );
84          window.addSubplot( vW, "Wheel Velocity [m/s]" )
                ;
85          window.addSubplot( F,  "Damping Force [N]" );
86          window.addSubplot( R,  "Road Excitation [m]" );
87
88      alg << window;
89
90
91      // SETUP CONTROLLER AND PERFORM A STEP:
92      // ------------------------------------
93      StaticReferenceTrajectory zeroReference( "ref.txt"
            );
```

```
94
95      Controller controller( alg, zeroReference );
96
97      Vector y( 4 );
98      y.setZero( );
99      y(0) = 0.01;
100
101     controller.init( 0.0, y );
102     controller.step( 0.0, y );
103
104
105     return 0;
106 }
```

9.6 Implementation of OCP in Python using CVXPY

CVXPY provides an example on how to formulate an OCP in Python [17]. **Convex optimization** can be used to solve many problems that arise in control. In this example, we show how to solve such a problem using **CVXPY**. We have a system with a **state** $x_t \in \mathbb{R}_n$ that varies over the time steps $t = 0, \cdots, T$, and **inputs** or actions $u_t \in \mathbb{R}_n$ we can use at each time step to affect the state. For example, x_t might be the position and velocity of a rocket and u_t the output of the rocket's thrusters. We model the evolution of the state as a **linear dynamical system**, i.e.,

$$x_{t+1} = Ax_t + Bu_t \tag{9.1}$$

where $A \in \mathbb{R}^{nxn}$ and $B \in \mathbb{R}^{nxm}$ are known matrices.

Our goal is to find the **optimal actions** $u_0, \cdots, u_{T-1}$ by solving the **optimization problems**

$$\begin{aligned} \min \quad & \sum_{t_0}^{T-1} l(x_t, u_t) + l_T(x_T) && (9.2)\\ \text{subject to:} \quad & x_{t+1} = Ax_t + Bu_t && (9.3)\\ & (x_t, u_t) \in \mathscr{C},\ x_T \in \mathscr{C}_T && (9.4) \end{aligned}$$

where $l : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is the stage cost, l_T is the terminal cost, $\mathscr{C}$ is the state/action constraints, and $\mathscr{C}_T$ is the terminal constraint. The optimization problem is convex if the costs and constraints are convex.

[17] https://nbviewer.org/github/cvxgrp/cvx_short_course/blob/master/intro/control.ipynb

Example

In the following code we solve a control problem with $n = 8$ states, $m = 2$ inputs, and horizon $T = 50$. The matrices A and B and the initial state $x0$ are randomly chosen (with $A \approx I$). We use the (traditional) stage cost $l(x,u) = \|x\|_2^2 + \|u\|_2^2$, the input constraint $\|u_t\|_\infty \leq 1$, and the terminal constraint $x_T = 0$.

Listing 9.4 MPC control problem with 8 states 2 inputs and horizon 50

```
1
2  # Generate data for control problem.
3  #-------------------------------------
4
5  import numpy as np
6
7  np.random.seed(1)
8  n = 8
9  m = 2
10 T = 50
11 alpha = 0.2
12 beta = 3
13 A = np.eye(n) - alpha * np.random.rand(n, n)
14 B = np.random.randn(n, m)
15 x_0 = beta * np.random.randn(n)
16
17 # Form and solve control problem.
18 # ------------------------------
19
20 import cvxpy as cp
21
22 x = cp.Variable((n, T + 1))
23 u = cp.Variable((m, T))
24
25 cost = 0
26 constr = []
27 for t in range(T):
28     cost += cp.sum_squares(x[:, t + 1]) + cp.
          sum_squares(u[:, t])
29     constr += [x[:, t + 1] == A @ x[:, t] + B @ u[:, t
          ], cp.norm(u[:, t], "inf") <= 1]
30 # sums problem objectives and concatenates constraints.
31 constr += [x[:, T] == 0, x[:, 0] == x_0]
32 problem = cp.Problem(cp.Minimize(cost), constr)
33 problem.solve()
```

We display the results below as a 4-high stack of plots showing u_1, u_2, x_1, and x_2 vs t. Notice that u_t is saturated (i.e., $\|u_t\|_\infty = 1$) initially, which shows that the input constraint is meaningful.

Listing 9.5 4-high stack of plots showing u1 u2 x1 and x2 vs t

```
1
2  # Plot results.
3  #----------------
4
5  import matplotlib.pyplot as plt
6
7  %config InlineBackend.figure_format = 'svg'
8
9  f = plt.figure()
10
11 # Plot (u_t)_1.
12 ax = f.add_subplot(411)
13 plt.plot(u[0, :].value)
14 plt.ylabel(r"$(u_t)_1$", fontsize=16)
15 plt.yticks(np.linspace(-1.0, 1.0, 3))
16 plt.xticks([])
17
18 # Plot (u_t)_2.
19 plt.subplot(4, 1, 2)
20 plt.plot(u[1, :].value)
21 plt.ylabel(r"$(u_t)_2$", fontsize=16)
22 plt.yticks(np.linspace(-1, 1, 3))
23 plt.xticks([])
24
25 # Plot (x_t)_1.
26 plt.subplot(4, 1, 3)
27 x1 = x[0, :].value
28 plt.plot(x1)
29 plt.ylabel(r"$(x_t)_1$", fontsize=16)
30 plt.yticks([-10, 0, 10])
31 plt.ylim([-10, 10])
32 plt.xticks([])
33
34 # Plot (x_t)_2.
35 plt.subplot(4, 1, 4)
36 x2 = x[1, :].value
37 plt.plot(range(51), x2)
38 plt.yticks([-25, 0, 25])
```

```
plt.ylim([-25, 25])
plt.ylabel(r"$(x_t)_2$", fontsize=16)
plt.xlabel(r"$t$", fontsize=16)
plt.tight_layout()
plt.show()
```

Bibliography

Ahmadzadeh, H. and E. Masehian (2015). "Modular robotic systems: methods and algorithms for abstraction, planning, control, and synchronization". *Artificial Intelligence 223*.

Andersson, O., P. Doherty, M. Lager, J.-O. Lindh, L. Persson, E. A. Topp, J. Tordenlid, and B. Wahlberg (2021). "WARA-PS: a research arena for public safety demonstrations and autonomous collaborative rescue robotics experimentation". *Autonomous Intelligent Systems 1:9*. URL: https://doi.org/10.1007/s43684-021-00009-9.

Åström, K. J. and R. M. Murray (2020). *Feedback Systems: An Introduction for Scientists and Engineers*. Second Edition, Princeton University Press.

Bereza, R., L. Persson, and B. Wahlberg (2020). "Distributed model predictive control for cooperative landing". *Proceedings of the 2020 IFAC World Congress*.

Bloesch, M., S. Weiss, D. Scaramuzza, and R. Siegwart (2010). "Vision based mav navigation in unknown and unstructured environments".

Borrelli, F., A. Bemporad, and M. Morari (2017). "Predictive control for linear and hybrid systems". *First Edition, Cambridge University Press*. URL: http://www.mpc.berkeley.edu/mpc-course-material.

Bouabdallah, S. (2007). *Design and control of quadrotors with application to autonomous flying*. PhD thesis. Ecole Polytechnique Federale de Lausanne.

Bouabdallah, S. and R. Siegwart (2007). "Full control of a quadrotor". *International Conference on Intelligent Robots and Systems, San Diego, CA, USA*.

Brachman, R. and H. Levesque (2004). *Knowledge Representation and Reasoning*. 1st Edition, Morgan Kaufmann.

Caicedo-Nunez, C. H. and M. Zefran (2008). "Consensus-based rendezvous". *IEEE International Conference on Control Applications*. URL: https://doi.org/10.1109/CCA.2008.4629611.

Cashmore, M., M. Fox, T. Larkworthy, D. Long, and D. Magazzeni (2013). "Planning inspection tasks for AUVs". *OCEANS 2013 - San Diego, CA, USA*, pp. 1–8. DOI: 10.23919/OCEANS.2013.6741058.

Cashmore, M., M. Fox, T. Larkworthy, D. Long, and D. Magazzeni (2014). "AUV mission control via temporal planning". *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6535–6541. DOI: 10.1109/ICRA.2014.6907823.

Coles, A., A. Coles, M. Fox, and D. Long (2010). "Forward-chaining partial-order planning". *Proceedings of the International Conference on Automated Planning and Scheduling* 20, pp. 42–49. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13403.

Doherty, P., F. Heintz, and J. Kvarnström (2013). "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation". *Unmanned Systems*. URL: https://www.ida.liu.se/divisions/aiics/publications/US-2013-High-Level-Mission.pdf..

Fairchild, C. and T. Harman (2017). *OS Robotics by Example*. Packt Publishing, 2nd edition.

Ferber, P. and J. Seipp (2022). "Explainable planner selection for classical planning". *Proceedings of the 36th AAAI Conference on Artificial Intelligence, Association for the Advancement of Artificial Intelligence*.

Fikes, R. E. and N. J. Nilsson (1971). "STRIPS: A new approach to the application of theorem proving to problem solving". *Artificial Intelligence, 2nd IJCAI, Imperial College, London, England*.

Furrer, F., M. Burri, M. Achtelik, and R. Siegwart (2016). "RotorS — A modular gazebo MAV simulator framework". *Springer, In Koubaa, A. (eds) Robot Operating System (ROS), Volume 1, 2016. Studies in Computational Intelligence*. URL: https://doi.org/10.1007/978-3-319-26054-9_23.

Geffner, H. and B. Bonet (2013). *A concise introduction to models and methods for automated planning*. Morgan & Claypool.

Geffner, H. and B. Bonet (2016). "Introduction to planning models and methods". *IJCAI Tutorial*.

Gerevini, A. and I. Serina (2002). "Lpg: a planner based on local search for planning graphs with action costs". *American Association for Artificial Intelligence*.

Ghallab, M., A. Howe, C. Knoblock, D. McDermott, A. Ram, M. M. Veloso, D. Weld, and D. Wilkins (1998). "PDDL - the planning domain definition language". *AIPS-98 Planning Committee*.

Ghallab, M., D. Nau, and P. Traverso (2004). *Automated Planning: Theory and Practise*. Morgan Kaufmann Publishers, San Fransisco, CA.

Ghallab, M., D. Nau, and P. Traverso (2016a). *Automated Planning and Acting*. Cambridge University Press.

Ghallab, M., D. Nau, and P. Traverso (2016b). *Deliberative planning and acting, ijcai tutorial*. URL: http://projects.laas.fr/planning/ijcai-2016-tutorial/index.html (visited on 2024).

Grune, L. and J. Pannek (2011). *Nonlinear model predictive control*.

Hartley, E. N. (2015). "A tutorial on model predictive control for spacecraft rendezvous".

Hartley, E. N., M. Gallieri, and J. M. Maciejowski (2013). "Terminal spacecraft rendezvous and capture with lasso model predictive control". **86**:11, pp. 2104–2113. URL: https://doi.org/10.1080/00207179.2013.789608.

Haslum, P., N. Lipovetzky, D. Magazzeni, and C. Muise (2019). *An Introduction to the Planning Domain Definition Language*. Morgan and Claypool.

Helmert, M. (2006). "The fast downward planning system". *Artificial Intelligence Research* 26, pp. 191–246. URL: https:/doi.org/10.1613/jair.1705.

Hoenig, W. and N. Ayanian (2017). "Flying multiple uavs using ros". In: *Robot Operating System ROS*. University of Southern California Springer.

Hoffmann, J. (2003). "The Metric-FF planning system: translating ignoring delete lists to numeric state variables". *Journal of Artificial Intelligence Research* 20, pp. 291–341. URL: https://doi.org/10.1613/jair.1144.

Hoffmann, J. and B. Nebel (2001). "The FF planning system: Fast plan generation through heuristic search". *Journal of Artificial Intelligence Research* 14, pp. 253–302. DOI: https://doi.org/10.1613/jair.855.

Houska, B., H. J. Ferreau, and M. Diehl (2011). "An auto-generated real-time iteration algorithm for nonlinear mpc in the microsecond range". *Automatica* **47**:10, pp. 2279–2285.

Ingrand, F. and M. Ghallab (2014). "Deliberation for autonomous robots: a survey". *LAAS-CNRS, University of Toulouse, France, Elsevier*. URL: https://doi.org/10.1016/j.artint.2014.11.003.

Jiang, Y., S. Zhang, P. Khandelwal, and P. Stone (2019). "Task planning in robotics: an empirical comparison of pddl-based and asp-based systems". *Frontiers of Information Technology & Electronic Engineering*. URL: https://arxiv.org/abs/1804.08229.

Kamel, M., M. Burri, and R. Siegwar (2017a). "Linear vs nonlinear mpc for trajectory tracking applied to rotary wing micro aerial vehicles". URL: http://arxiv.org/abs/1611.09240v2.

Kamel, M., T. Stastny, K. Alexis, and R. Siegwart (2017b). "Model predictive control for trajectory tracking of unmanned aerial vehicles using robot operating system". *Robot Operating System (ROS) The Complete Reference (A. Koubaa, ed.), Springer*.

Katz, M., S. Sohrabi, H. Samulowitz, and S. Sievers (2018). "Delfi: online planner selection for cost-optimal planning". *IPC-9 Planner Abstracts*, pp. 57–64.

Komenda, A., M. Stolba, and D. L. Kovacs (2016). "The international competition of distributed and multiagent planners (codmap)". *AI Magazine 37*, pp. 109–115.

Kovacs, D. L. (2012). "A multi-agent extension of PDDL3.1". *Proceedings of the 3rd Workshop on the International Planning Competition (IPC)*, pp. 19–27.

Kvarnström, J. (2011). "Planning for loosely coupled agents using partial order forward-chaining". *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS). AAAI*, pp. 138–145.

LAMBREGTS, A. (1983). "Vertical flight path and speed control autopilot design using total energy principles". *AIAA, Guidance and Control Conference*. DOI: 10.2514/6.1983-2239.

Lane, D. M., F. Maurelli, P. Kormushev, M. Carreras, M. Fox, and K. Kyriakopoulos (2012). "Persistent autonomy: the challenges of the PANDORA project". *IFAC Proceedings Volumes* **45**:27. 9th IFAC Conference on Manoeuvring and Control of Marine Craft, pp. 268–273. DOI: https://doi.org/10.3182/20120919-3-IT-2046.00046.

LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

Lee, D., T. Ryan, and H. J. Kim (2012). "Autonomous landing of a VTOL UAV on a moving platform using image-based visual servoing", pp. 971–976.

Mahony, R., V. Kumar, and P. Corke (2012). **19**:3, pp. 20–32.

Manhães, M. M. M., S. A. Scherer, M. Voss, L. R. Douat, and T. Rauschenbachnd (2016). "UUV simulator: A Gazebo-based package for underwater intervention and multi-robot simulation". *OCEANS 2016 MTS/IEEE Monterey*, pp. 1–8. DOI: 10.1109/OCEANS.2016.7761080.

Marconi, L., A. Isidori, and A. Serrani (2002). *Autonomous vertical landing on an oscillating platform: an internal-model based approach*. Automatica.

Mattingley, J., Y. Wang, and S. Boyd (2011). "Receding horizon control: automatic generation of high-speed solvers". *IEEE Control Systems Magazine*. DOI: 10.1109/MCS.2011.940571.

Maurelli, F., M. Carreras, J. Salvi, D. Lane, K. Kyriakopoulos, G. Karras, M. Fox, D. Long, P. Kormushev, and D. Caldwell (2016). "The PANDORA project: A success story in AUV autonomy". *OCEANS 2016 - Shanghai*, pp. 1–8. DOI: 10.1109/OCEANSAP.2016.7485618.

Mellinger, D., N. Michael, and V. Kumar (2012). "Trajectory generation and control for precise aggressive maneuvers with quadrotors". **31**:5.

Mellinger, D., M. Shomin, and V. Kumar (2010). "Control of quadrotors for robust perching and landing". *International Powered Lift Conference, Philadelphia, PA*.

Muñoz, P., M. R-Moreno, and D. Barrero (2016). "Unified framework for path-planning and task-planning for autonomous robots". *Robotics and Autonomous Systems* **82**, pp. 1–14.

Murphy, R. R. (2019). *Introduction to AI Robotics*. 2nd Edition, MIT Press.

Murray, R. M. (2022). *Optimal-Based Control*. Control and Dynamical Systems, California Institute of Technology (Caltech).

Muskardin, T., G. Balmer, L. Persson, S. Wlach, M. Laiacker, A. Ollero, and K. Kondak (2017). "A novel landing system to increase payload capacity and operational availability of high altitude long endurance uavs". *Intelligent & Robotic Systems* 88(2), pp. 597–618.

"Optimal UAV rendezvous on a UGV" (2016). *AIAA Guidance, Navigation, and Control Conference*.

Pedersen, M. R., L. Nalpantidis, R. S. Andersen, C. Schou, S. Boegh, V. Krueger, and O. Madsen (2015). "Robot skills for manufacturing: from concept to industrial deployment". *Robotics and Computer-Integrated Manufacturing*.

Pednault, E. P. D. (1986). "Formulating multiagent, dynamic-world problems in the classical planning framework". *In Reasoning about Actions and Plans: Proc. Workshop*, pp. 47–82.

Pednault, E. (1988). "Synthesizing plans that contain actions with context-dependent effects". *Computational Intelligence* 4, pp. 356–372.

Persson, L. (2021). *Model Predictive Control for Cooperative Rendezvous of Autonomous Unmanned Vehicles*. PhD thesis. KTH Stockholm.

Persson, L., T. Muskardin, and B. Wahlberg (2017). "Cooperative rendezvous of ground vehicle and aerial vehicle using model predictive control". *IEEE 56th Annual Conference on Decision and Control (CDC)*, pp. 2819–2824.

Persson, L. and B. Wahlberg (2019). "Model predictive control for autonomous ship landing in a search and rescue scenario". *AIAA SciTech Forum, San Diego, California*.

Poole, D. L. and A. K. Mackworth (2017). *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2nd Edition.

Powers, C., D. Mellinger, and V. Kumar (2015). "Quadrotor kinematics and dynamics". In: *Handbook of Unmanned Aerial Vehicles. (Valavanis and Vachtsevanos)*. Springer Netherlands Dordrecht, pp. 307–328.

Reiter, R. (2001). "Knowledge in action : logical foundations for specifying and implementing dynamical systems". URL: `http://www.cs.toronto.edu/cogrobo/kia/`.

Riccio, F. (2018). *Spatial Representation for Planning and Executing Robot Behaviors in Complex Environments*. PhD thesis. Sapienza University of Rome.

Richter, S. and M. Westphal (2010). "The lama planner: guiding cost-based anytime planning with landmarks". *Journal of Artificial Intelligence Research* 39, pp. 127–177. URL: `https://doi.org/10.1613/jair.2972`.

Roberto Guzman Roman Navarro, M. C. and J. Ariño (2017). "Robotnik - professional service robotics applications with ros". In: *Robot Operating System (ROS)*. Vol. 2. Studies in Computational Intelligence, vol 707, Springer.

Russell, S. and P. Norvig (2010). *Artificial Intelligence: A Modern Approach, 3rd Edition*. Prentice Hall Series in Artificial Intelligence. URL: http://aima.cs.berkeley.edu.

Sa, I., M. Kamel, M. P. R. Khanna, J. Nieto, and R. Siegwart (2017). "Dynamic system identification, and control for a cost-effective and open-source multi-rotor mav". *Field of Service Robotics*.

Sharma, A. (2019). *System Identification of a Micro Aerial Vehicle*. MA thesis. Luleå University of Technology, Sweden.

Steenstra, L. (2019). *PDDL-Based Task Planning of Survey Missions for Autonomous Underwater Vehicles*. MA thesis. Delft University of Technology.

Torreño, A., E. Onaindia, A. Komenda, and M. Stolba (2017). "Cooperative multi-agent planning: a survey". **50**:6. URL: https://doi.org/10.1145/3128584.

Torreño, A., E. Onaindia, and O. Sapena (2014). "FMAP: distributed cooperative multi-agent planning". *Applied Intelligence* **41**, pp. 606–626.

Uddin, N. (2020). "System identification of two-wheeled robot dynamics using neural networks". *Journal of Physics, Conference Series 1577 012034*. DOI: 10.1088/1742-6596/1577/1/012034.

Ulam, P., Y. Endo, A. Wagner, and R. Arkin (2007). "Integrated mission specification and task allocation for robot teams – design and implementation". *ICRA*.

WARA-PS (2024). *WARA-PS core system portal, LiU/AIICS*. URL: https://portal.waraps.org/ (visited on 2024).

Wenzel, K. E., A. Masselli, and A. Zell (2011). "Automatic take off, tracking and landing of a miniature uav on a moving carrier vehicle". *Journal of Intelligent & Robotic Systems, 61(1):221–238*.

INSTITUTIONEN FÖR REGLERTEKNIK | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2025-12-19

EXAMENSARBETE UAV-UGV Landning samt AI-Uppgiftsplanering för Robotiserad Industriell Inspection
STUDENT Hicham Mohamad
HANDLEDARE Anders Robertsson, Gregory Austin, Björn Olofsson, Yiannis Karayiannidis (Reglerteknik)
EXAMINATOR Karl-Eric Årzen (Reglerteknik)

AI-baserad samverkan mellan UAV och UGV för industriell robotinspektion

POPULÄRVETENSKAPLIG SAMMANFATTNING **Hicham Mohamad**

Denna avhandling adresserar två huvudsakliga problemområden: rendezvous-landning av en quadcopter på en mobil markfarkost, samt AI-planering (även känt som uppgiftsplanering) för industriell robotinspektion.

UAV-UGV-samverkan, vilket avser samarbete mellan obemannade luftfarkoster (UAV) och obemannade markfordon (UGV), är ett kraftfullt koncept där drönare och markrobotar arbetar tillsammans för att övervinna individuella begränsningar. Detta skapar en mer robust, effektiv och mångsidig lösning för komplexa uppdrag som katastrofinsatser, jordbruk eller försvar.

I den första delen av den här studie undersöker vi hur man löser ett rendezvous-landningsproblem mellan en luftfarkost (UAV) och ett markfordon (UGV). Därefter, i den andra delen, behandlar vi konceptet intelligent robotbeteende (eller så kallad avsiktligt agerande). För att möjliggöra sådana sofistikerade deliberativa färdigheter undersöker vi således hur man utformar roboten för att korrekt lösa ett AI-planeringsproblem (även känt som uppgiftsplanering) inom domänen robotiserade industriella inspektioner.

I detta syfte är målet med projektets första del att utforma en autopilot som kan tvinga quadrotorn UAV att spåra den rörliga markroboten och därefter landa på den. Simulerade experiment utförs med den ROS-baserade UAV-simulatorn, tillhandahållen av WARA-PS ramverket (WASP Forskningsarena - Allmänna Säkerheten), tillsammans med en nykonstruerad modell för simulering av en tvåhjulig robot (TWR). De erhållna resultaten bekräftar den föreslagna algoritmens tillfredsställande prestanda genom att illustrera konvergensdiagrammet. Vidare, för att förstärka prestandan hos denna rendezvous-algoritm, undersöker vi dessutom den teoretiska utvecklingen av tre olika avancerade metoder med hjälp av den lovande reglermetoden modell-prediktiv reglerteknik (MPC).

I det andra arbetet är målet att formulera och implementera en planeringsmodell i Planning Domain Definition Language (PDDL) som kan lösa ett inspektionsplaneringsproblem genom att generera en sekvens av sammansatta abstrakta handlingar (en högnivå-målplan). Som ett resultat erhålls en möjlig lösningsplan för ett exempel på ett litet simulerat inspektionsuppdrag framgångsrikt. Därefter expanderas (förfinas) denna genererade PDDL-lösningsplan på hög nivå för att i slutändan erhålla en möjlig kooperativ målplan bestående av detaljerade elementära (konkreta) handlingar.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2026

Document Number
TFRT-6315

Author(s)
Hicham Mohamad

Supervisor
Gregory Austin, Dept. of Computer Science, Sweden
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

UAV-UGV Rendezvous and AI Planning for Robotic Industrial Inspection Based on ROS and WARA-PS

Abstract

In the past, robots were goal-based agents and experienced a more limited world. These robots needed to be instructed on each individual step or action to complete a larger task. They worked in static and well-modelled environments and run predefined behaviours where their missions consisted of repeating a single task (e.g. assembling and vacuum cleaning). Nowadays the world of robots and autonomous vehicles is growing continuously making our environment more robotised due to advanced research and development in electronics (storage and computation power) and Artificial Intelligence (AI) algorithms.

In fact, by fully exploiting the robot capabilities and taking advantage of the AI algorithms we are changing our well-being and productivity in different sectors, particularly in industry, health-care and public safety. These technological advances opened up the opportunity for new applications. For instance, we have seen an increasing number of practical applications using multi-robot systems (or robot fleet) within industry, search and rescue (SAR), space exploration and other areas, to address
specific mission scenarios where the control techniques are based on the field of multi-agent networked control in robotic networks. In particular, the powerful concept UAV-UGV cooperation where unmanned aerial vehicles (UAVs) and unmanned ground vehicles (UGVs) collaborate to create a more robust, efficient, and versatile solution for complex missions like disaster response, agriculture, or defence.

In this degree project, we present two distinct problems in the engineering fields of Automatic Control and AI Robotics. In the first part, we investigate how to solve a Rendezvous landing problem between a quadcopter (UAV) and a mobile robot (UGV). Next, in the second part we deal with the concept of intelligent behaviour that requires human-level cognitive capabilities of which a central problem is selecting the action to do next or the so-called deliberative acting. Thus, to enable such
sophisticated deliberative skills, we investigate how to design the robot to properly solve an AI planning problem (also known as Task Planning) in Robotic Inspection domain. These two problems can be related and would complement each other in certain inspection missions, for example in construction/industrial scenarios, deploying such aerial and ground vehicles.

To this end, the goal of the project first part is to design an autopilot that can execute a Rendezvous landing algorithm, which is an important problem in autonomous flight. The main steps of the algorithm consist of: tracking, alignment, descent, retardation (shutting of propeller motors) and touch-down. Specifically, the autopilot's task is to force the quadrotor UAV "DJI Matrice 100" (a common research platform) to track a quadruped robot "Spot" (from Boston Dynamics) and subsequently land on it.

In this study, the dynamics of both quadrotor and quadruped systems were examined and presented, where the models are relatively simplified or necessarily approximated as first or second-order systems, in similar fashion as in various literature. Further, the Rendezvous algorithm uses Robot Operating System (ROS) to command and control the flight of the quadrotor UAV and the movement of the mobile ground robot. Substantially, the algorithm makes use of the ROS interface (including services, subscribed topics, and published topics) and runs a fourdimensional PID position-tracking controller implemented in Python to control the four basic movements of the quadrotor: thrust, roll, pitch and yaw. The objective of this controller is to compute and send correction commands to guide the UAV drone to track the target robot platform on the ground, as well as to accomplish the alignment of the acting unmanned vehicles in the horizontal plane and the landing operation at the end.

Simulated experiments are conducted using the ROS-based UAV simulator provided by WARA-PS framework (WASP Research Arena Public Safety), as well as using a two-wheeled simulated robot (TWR), which is a newly constructed URDFbased simulation model for simulating the movement of Spot robot in the rendezvous manoeuvre. The obtained results confirm the satisfactory performance of the proposed algorithm (implemented in Python) by illustrating the convergence graph using the PID position-tracking controller.

Further, for reinforcing the performance of this Rendezvous algorithm we additionally investigate other robust control strategies using the promising Model Predictive Control (MPC) approach, where theoretical development for three different advanced MPC-based methods is presented in this report, in which interaction between platforms can also be included in cooperative manner. However, the practical implementations of this second approach using MPC, including simulations and real-world flight tests and experiments, will be addressed in future work.

In the second work, the goal is to formulate and implement a planning model in Planning Domain Definition Language (PDDL) that can solve an inspection planning problem by generating a sequence of composite (abstract) actions. That is, as a result we obtain a high-level goal plan to perform certain tasks and achieve the desired objectives (like "spot-inspect" or "uav-inspect"). In fact, one possible solution plan to a small simulated inspection mission example is successfully obtained using our implemented PDDL-model, composed of domain and problem files, as input to a suitable off-the-shelf AI-based solver (task planner). Next, this generated highlevel PDDL solution plan is expanded (refined) into executable Task Specification Tree (TST), using TST Factory inWARA-PS, to obtain at the end a possible cooperative goal plan consisting of detailed elementary (concrete) actions, like "move-to" and "fly-to". Testing the applicability of the resulting PDDL plan of actions in realworld problems will be performed in future research work.

Keywords		
Classification system and/or index terms (if any)		
Supplementary bibliographical information		
ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-182	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/