Rendezvous Landing using ROS, and AI Planning of Robotic Inspection Missions using PDDL

Deploying Ground and Aerial Robots in Automated Industrial Inspections

Degree Project Manuscript

by Hicham Mohamad

Supervisor: Prof. Anders Robertsson

May 29, 2023

LUND
UNIVERSITY

Department of Automatic Control
LTH - Lund University, Sweden

Rendezvous Landing using MPC, and
PDDL-Based AI Planning of Robotic Inspection Missions:
Deploying Ground and Aerial Robots in Automated Industrial Inspections

By Hicham Mohamad

Contents

Chapter 1

Introduction

By introducing the new technology, it becomes usual the generation and execution of automated mission and coordination of tasks involving multiple agents. In addition, it becomes of growing interest and useful the cooperation between UAV drones and mobile ground robots to carry out important inspections and monitoring in industry and in the areas of public safety. In fact, this multiagent cooperation will reduce the time to perform inspection, surveying and maintenance tasks in industry and will also minimize the challenges to accomplish urgent tasks in hazardous and difficult-to-access environments. Especially the deployment of the UAV flying robots have also demonstrated great success in remote sensing and visual inspection tasks.

Figure 1.1: The quadruped Spot robot teaming with three UAV quadcopters.

1.1 objectives

The aim of this project consists of two distinct problems to solve. These two following objectives will be accomplished using the *WARA-PS Core System* [1], which is a ROS-based software framework.

- **Rendezvous landing problem**: The objective of the first problem is to design and implement a suitable controller to perform a *autonomous Rendezvous landing* of the UAV quadrotor **DJI Matrix 100** on top of the quadruped **Spot** robot, "safely we hope".
- **AI planning problem**: In the second task, we will deal with the important topic of *automating industrial inspections with agile mobile robots* like Spot. Then, the objective would be to develop a PDDL-based task planning, i.e., an Artificial Intelligence (AI) automated planning of *robotic inspecting missions* for Spot robot together with the UAV quadrotor. As such, it will be possible to automatically search for a way of combining actions into a "strategic plan" $\pi = a_0, \ldots, a_n$ that can achieve a desired goal in a certain area of inspection scenario.

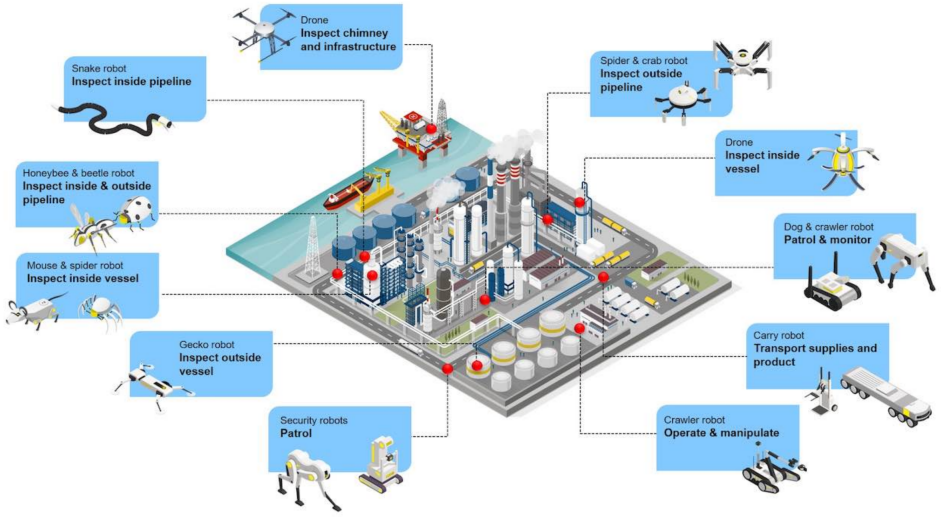


Figure 1.2: An overview of different collaborative robot systems necessary to carry out important inspections and monitoring tasks in industry. Courtesy: Yokogawa Automation.

1.2 Visual inspection scenario (Future Work)

Experimental evaluation of the coordination and execution of inspection missions by Spot and the UAV quadrotor will be performed in a small practical scenario that consists of visual inspection tasks in 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, on Spot and the DJI quadrotor.

In case Spot robot is incapable to reach some difficult-to-access inspection points, the UAV drones "DJI M100" will intervene and accomplish such unreachable tasks having the ability of flying and seeing, using remote sensing and visual inspection.

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 components, detailing, dimensions and so on.

In general, this degree project will contribute to the development of **collaborative robot systems** and **Automating of industrial inspection** necessary to carry out important inspections and monitoring in industry and in the domain of public safety. In Figure 1.2, we can see several illustrative examples of industrial inspections from Yokogawa Automation.

Chapter 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 of Åström and Murray in [9]. 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.

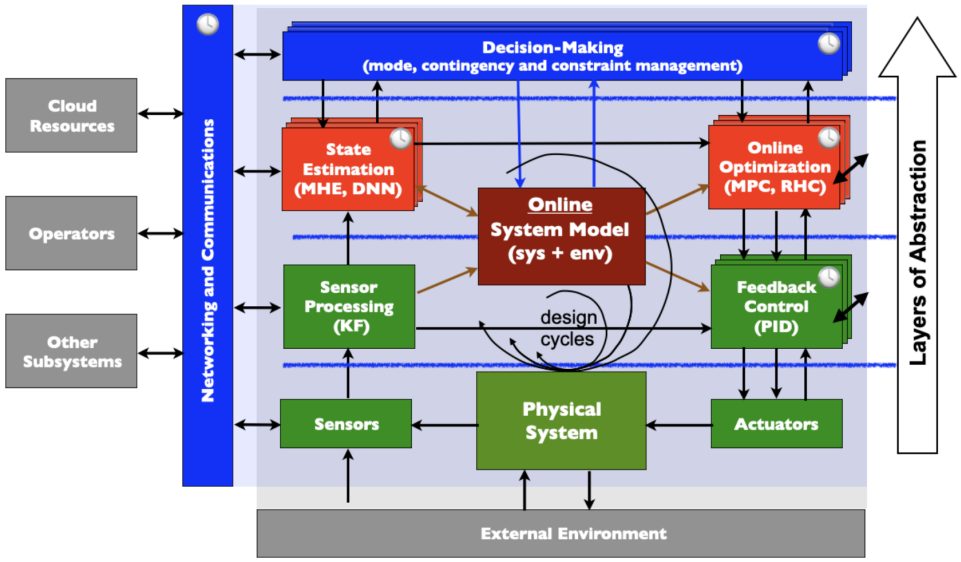


Figure 2.1: Layered control systems. The system consists of four layers: the physical layer (lowest), the feedback regulation layer (green), the trajectory generation layer (red), and the decision-making layer (blue). Networking and communications allows information to be transferred between the layers as well as with outside resources (left). The entire system interacts with the external environment, represented at the bottom of the figure [10] [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 [9]. 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 3.5 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 [10] [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 [9] and [10], 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 [Blösch et al., 2010] [17] , [Kamel, Stastny et al., 2017] [18] and [Kamel, Burri, et al., 2016] [19], 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 [16] [Mahony, Kumar and Corke, 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 discussed in the following sections and illustrated in Figures 5.4 and 5.6. 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).

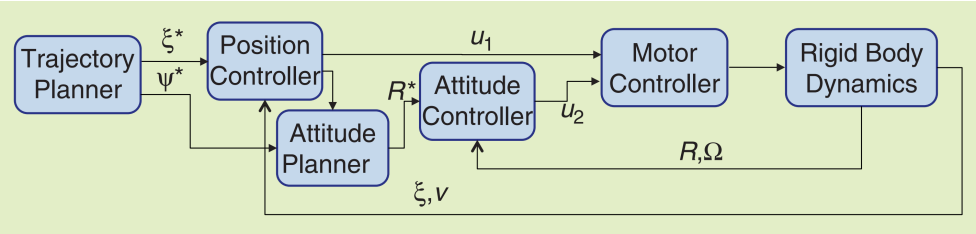
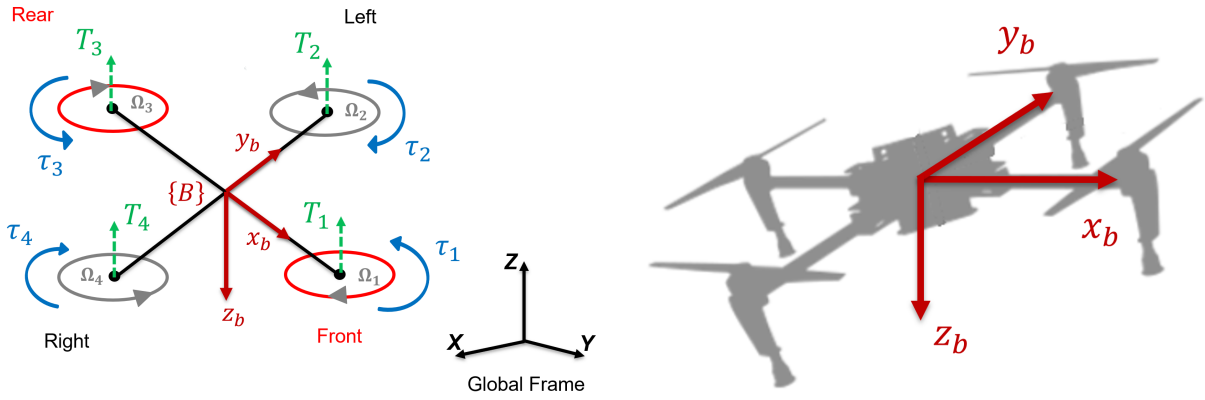


Figure 2.2: A common hierarchical control approach for VTOL quadrotors: The innermost motor control loop, the intermediate attitude control loop, and the outer position control loop. These levels form nested feedback loops. ***Rotor speed*** *drives the dynamic model of the vehicle, so high-quality control 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 [16] [Mahony, 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.



(a) (b)

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 [14], [16], [25]. 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 $\mathbf{u} = (T_\Sigma, \boldsymbol{\tau})$. 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 [13] [Bouabdallah, 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.

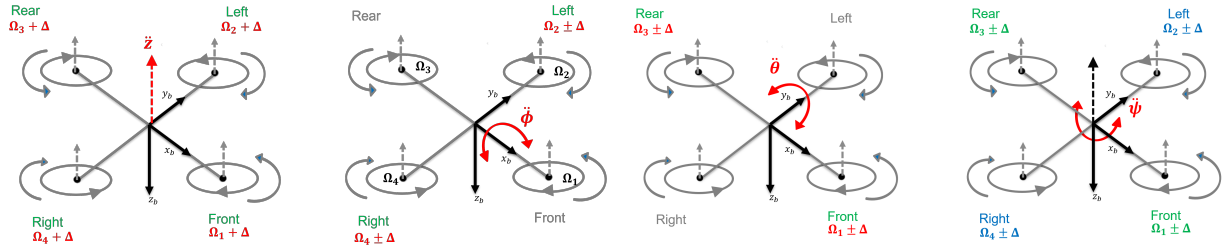


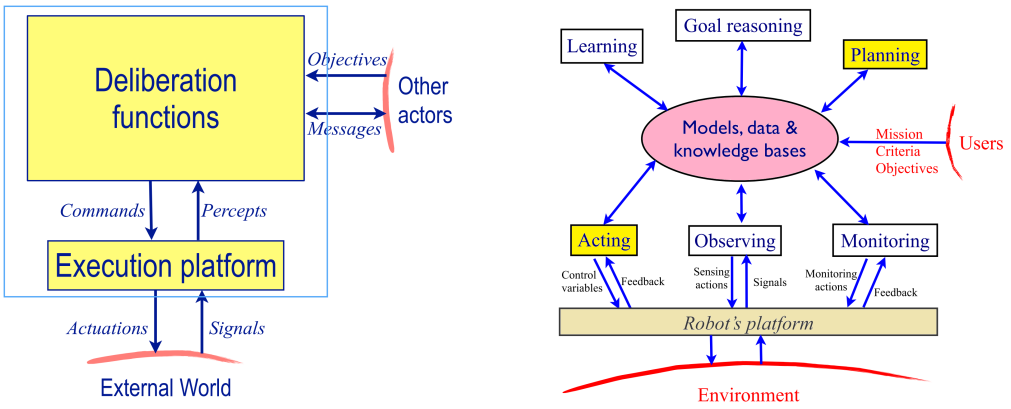
Figure 2.4: Characteristics of the quadrotor movement: Depending on the speed rotation of each propeller it is possible to identify the four basic movements of the quadrotor. In hovering, all the propellers rotate in the same speed Ω to counterbalance the gravity acceleration g. (a) ***Throttle input*** *(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: Deliberative Functionalities, Main Features, Representation, 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* [2, 3, 4].

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 [68].



(a) (b)

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 [62]. 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. [68]. 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 [62, 68], 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 [68], 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 [68, 62]. "Automated Planning" textbook of (Ghallab et al., 2016) [62] 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 [68]. LaValle's textbook [60] "Planning Algorithms" (2006) covers both classical and stochastic planning, with extensive coverage of robot motion planning.

2.4.1 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, et al. 2014] [68].

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] [46]. 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] [68]. 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 [Robin R. Murphy, 2019] [4]. 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] [80]

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 [Francesco Riccio, 2018] [70].

2.4.2 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. [47].

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 [47].

First, we argue that the concept of planning conceived as *the model-based approach to action selection*, as stated in [Geffner and Bonet, 2016][47], 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 [47]

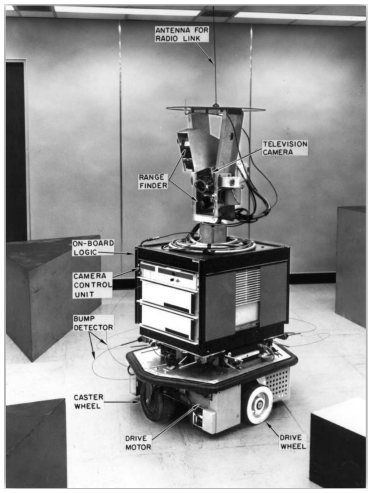
- *Programming-based approach* - Specify control by hand: the controller that prescribes the action to do next is given by the programmer, usually in a suitable high-level language.

- *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.

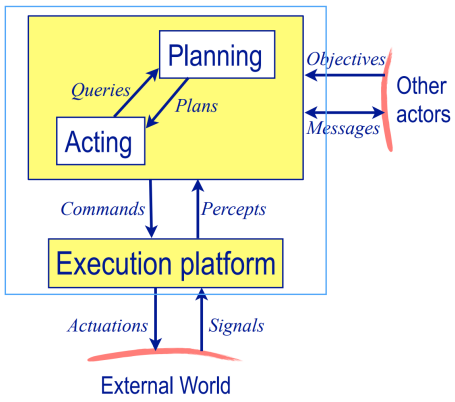
AI Planning Framework: Historical Background According to historical notes in [2], AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains. The Stanford Research Institute Problem Solver 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] [65], 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, 1988] [66, 67], relaxed some of the STRIPS restrictions and made it possible to encode more realistic problems.

The *Problem Domain Description Language*, or PDDL (Ghallab et al., 1998) [63], 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., 2016] [62]:* ***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).*

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

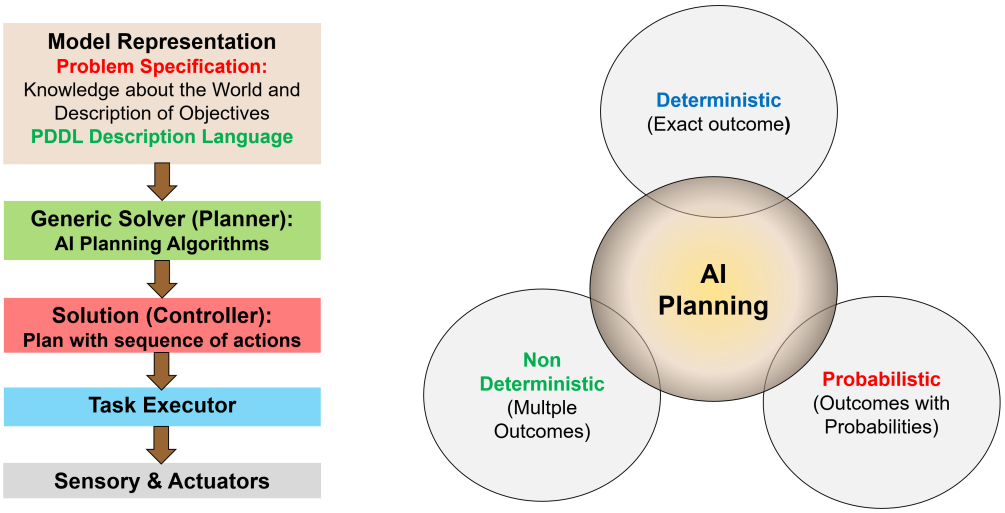
Virtually all work in automated planning has been concerned with ways of representing and solving what might be called the *classical planning problems*. The so-called "classical planning problem" is normally considered the simplest form of planning problem, the restricted conceptual model, which assumes a deterministic, discrete and essentially non-temporal world model [69]. This framework has typically been used to model real-world problems and is usually composed of the following three components, as stated in [47]

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" [69].

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

   - *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 [2].



(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 [69]. (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 [47] 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 [67].

Indeed, a wide range of models used in planning can be understood as variations of a *basic state model* featuring $\mathcal{S} = \langle S, s_0, S_G, A, f, c\rangle$. In this way, the model underlying classical planning can be described as this state model $\mathcal{S} = \langle S, s_0, S_G, A, f, c\rangle$, 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 [67].

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 [47].*

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. States are described by a set of atoms or propositions/sentences, as shown in Figure 2.11. States change via the execution of planning actions. In fact, The plan creation aspect of AI planning deals mainly with *state transformation problems.*

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 [47]. 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 $\mathcal{S} = \langle S, s_0, S_G, A, f, c \rangle$ 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 $\mathcal{S}$ iff π generates a sequence of states $s_0, \ldots, s_{m+1}$ that represents a directed path in the weighted digraph $G = (V, E, w)$ Thus, in principle any "path-finding algorithm" over weighted directed graphs can be used for finding plans for the classical planning model.

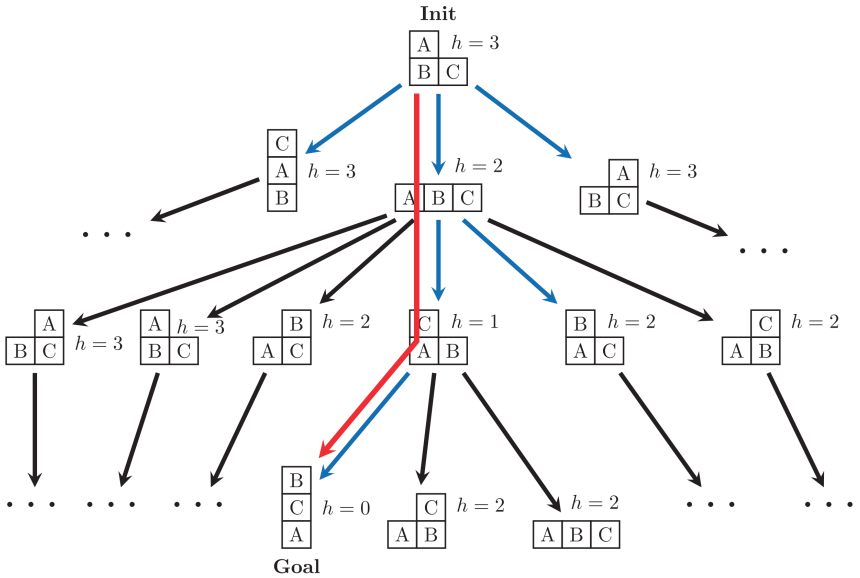


*Figure 2.10: There is a direct correspondence between **classical planning models** and "directed graphs", and between **classical plans** and certain "paths" over these graphs: A fragment of the graph corresponding to a simple **Blocks World planning problem** involving three blocks with initial and 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. [47].*

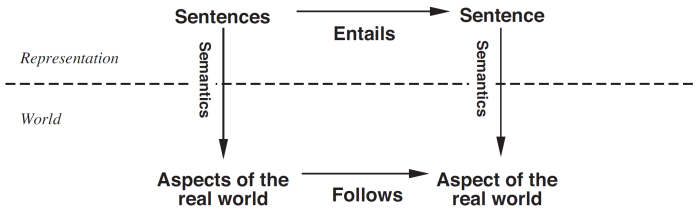


Figure 2.11: ***Correspondence between Real World and Representation****: A compact description of models (planning problems) is usually expressed in languages featuring "variables" (Strips, PDDL, PPDDL, ...), where the* ***states*** *are the possible "valuations" of the variables. Sentences are physical 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 *[50]. Courtesy: AIMA book [2]*

2.4.3 AI Planning Algorithms and Resources

In terms of automated planning, different valuable literatures, like [Russell and Norvig] [2], Ghallab [61], Lavalle [60] and some tutorials available online [47, 48], 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 performance 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 [78] [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 [69] [Haslum et al., 2019]. In fact, as mentioned by [Ghallab et al., 2016] [61] and illustrated in Figure 2.7b, Automated 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. 2016] [61]. 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] [79].

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] [2], in [61, 60] 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

[3] http://planning.domains/
[4] https://github.com/AI-Planning/classical-domains
[5] http://editor.planning.domains
[6] https://www.ida.liu.se/~TDDC17/info/labs/lab4/lab4_planning.en.shtml

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* 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 [81] [Jiang et al., 2018].

		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. [76] [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 [75, 79, 76, 78].

2.4.4 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 [69] [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 [2], 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 *pre-conditions and effects* [2]. 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

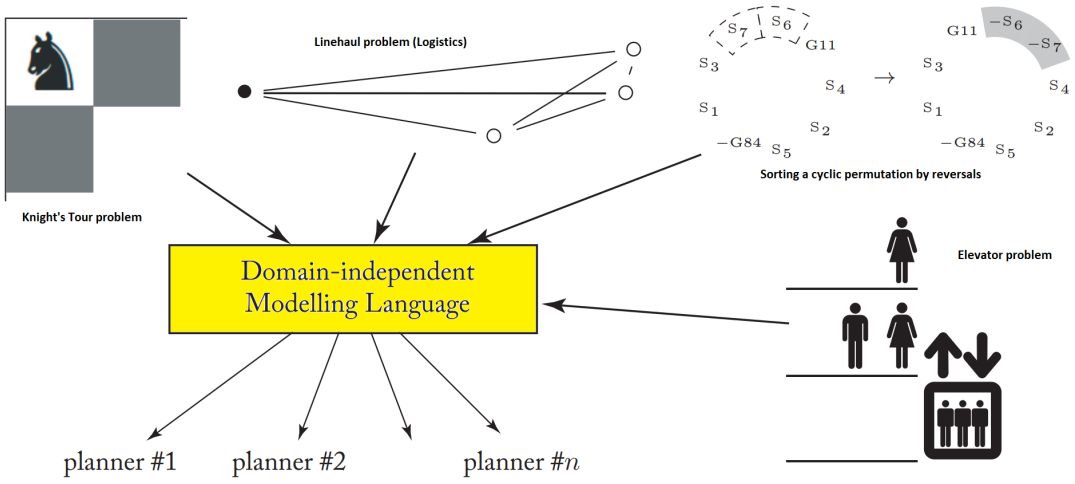


Figure 2.13: Conceptual overview of solving common planning problems: Knights Tour, Linehaul, Sorting a Cyclic Permutation by Reversals and Elevator problem in [69] [Haslum et al., 2019]. The idea of domain-independence in AI planning is to place a modelling language between the planning problem to be 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 .

2.4.5 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 nond-eterministic effects [69]. 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 [79, 76].

- *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].

- *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 [69]. 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 [86].

- *Situation Calculus.* The situation calculus, a dialect of first-order logic [59], 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* [2].

  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 [2]

- *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.

2.5 WARA-PS Core System

The core system of **WASP Research Arena Public Safety** [11], also known as WARA-PS Core System, is based on the use of ROS, Robot Operating System. 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** [1]. 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.

[7] http://www.cs.toronto.edu/cogrobo/kia/
[8] https://en.wikipedia.org/wiki/GOLOG
[9] https://en.wikipedia.org/wiki/Situation_calculus
[10] https://en.wikipedia.org/wiki/Behavior_tree_(artificial_intelligence,_robotics_and_control)
[11] https://wasp-sweden.org/research/research-arenas/wara-ps-public-safety/

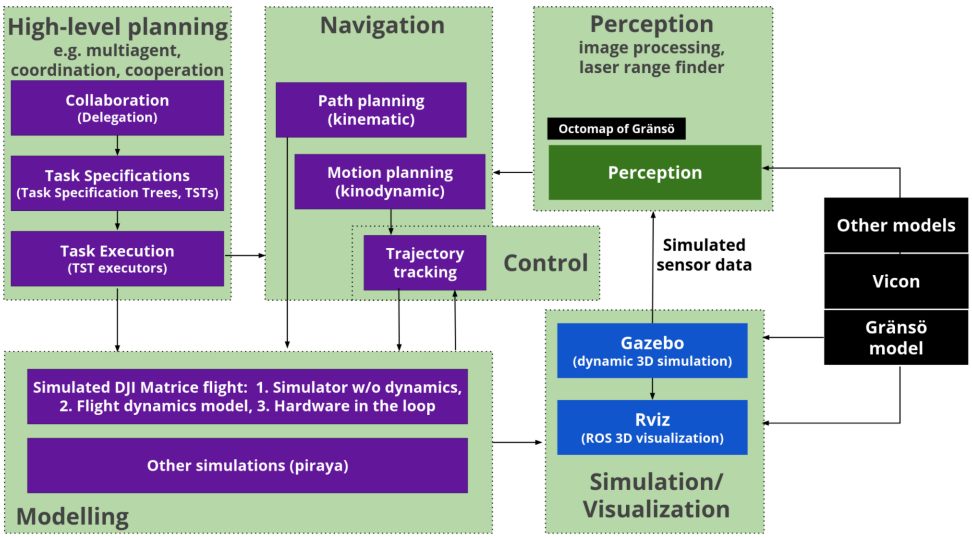


Figure 2.14: The architecture of the WARA-PS Core System: The central nodes of this ROS-based framework consist of: High-level Planning – Navigation – Perception – Control – Modelling - Simulation/Visualization. Courtesy: WARA-PS Portal (LiU/AIICS) [1]

2.5.1 Tutorials - WARA-PS Execution Environment

At **WARA-PS portal** (LiU/AIICS) [1], 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 under preparation, 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**.

2.5.2 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.

2.5.3 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.

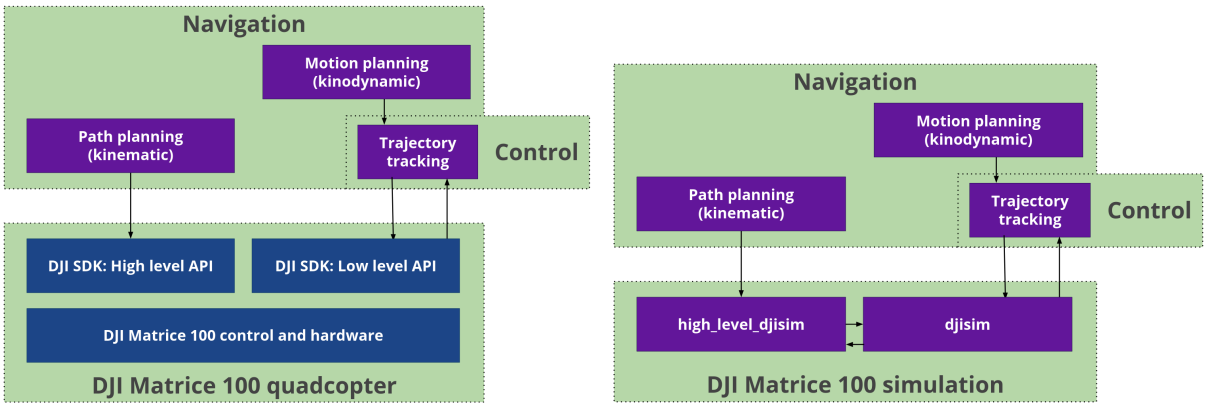


*Figure 2.15: (a) The architecture of the **ROS DJI SDK** provided by the DJI Matrice 100, through which an actual physical quadcopter can be controlled in two ways: A path planner generating a sequence of waypoints could use the high level API. Alternatively, a motion planner could generate a trajectory 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`).*

2.5.4 WARA-PS Simulators

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.

2.5.5 WARA-PS DJI Simulator – Full Dynamics with MPC

As illustrated in Figure 2.16, it is worthwhile to give a short overview of the ROS communication underlying the DJI simulator with MPC provided by WARA-PS framework. In the following we summarize the important ROS topics and nodes interconnection:

- 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.

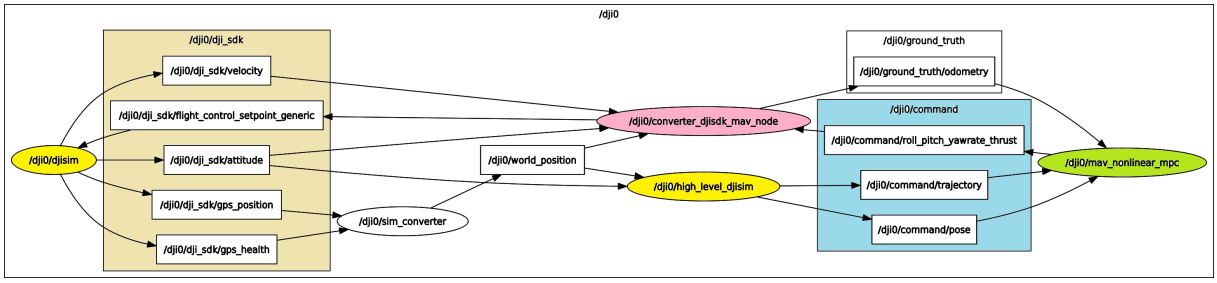


Figure 2.16: ***Simulation with MPC in WARA-PS Framework****. The generated ROS connection graph showing how the simulation is connected including the ROS nodes and topics that are involved. Since this is a closed control loop, there is no perfect place to begin. Let us decide to follow the execution 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.*

Chapter 3

Physical Modelling

A model of a system is a fundamental tool in any project. Modelling is necessary to help with the development of the autonomous landing control using 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.

Before discussing the model of our system in this project, it is worth to present the main characteristics of our rendezvous landing 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 MPC control approach.
- 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.
- Drone/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/simplification methods to obtain it.

3.1 Process approximation approach

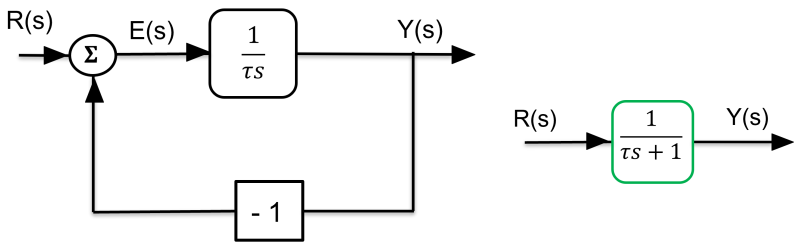
Since design information/parameters cannot be available for us, it is not possible to use the classic Newton-Euler methods to find the necessary kinematic/dynamic model of the robots involved in the project. In order to tackle this issue, a practical approximation technique, based on low-pass filter (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.

3.1.1 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:

$$\tau \frac{dy}{dt} + y = Kr \tag{3.1}$$



(a) (b)

Figure 3.1: (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

$$\tau Y(s) + Y(s) = KR(s) \implies G_{LPF}(s) = \frac{Y(s)}{R(s)} = K\frac{1}{\tau s + 1} \tag{3.2}$$

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

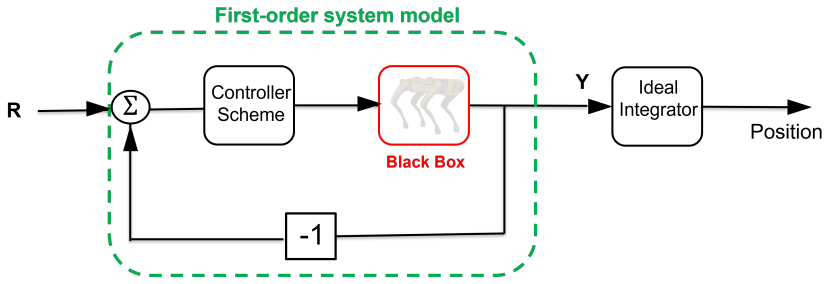
- *Time constant* is a measure of how quickly a first-order system response to a unit step input.
- *DC gain* of the system is the ratio between the steady state value of output and the input signal.
- The obtained transfer function in 3.2 represents a first-order *low-pass filter*.
- *Smoothing*: In 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.

3.2 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 3.2a, 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 $\mathbf{v_r}$ and its true velocity $\mathbf{v}$. As such, *fast closed-loop dynamics*, from $\mathbf{v_r}$ to $\mathbf{v}$, would yield the desired approximation $\mathbf{v} \approx \mathbf{v_r}$.



(a)

(b)

Figure 3.2: (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.

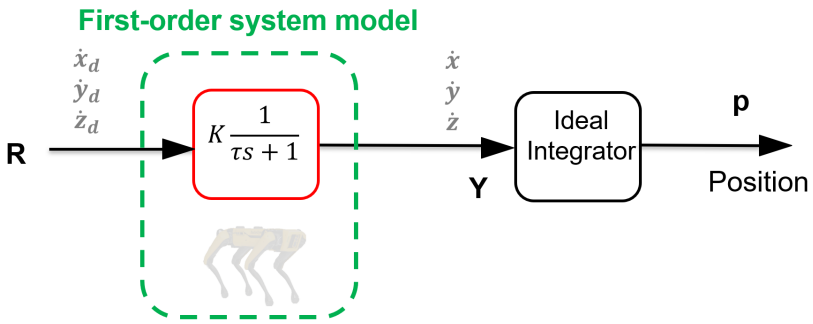


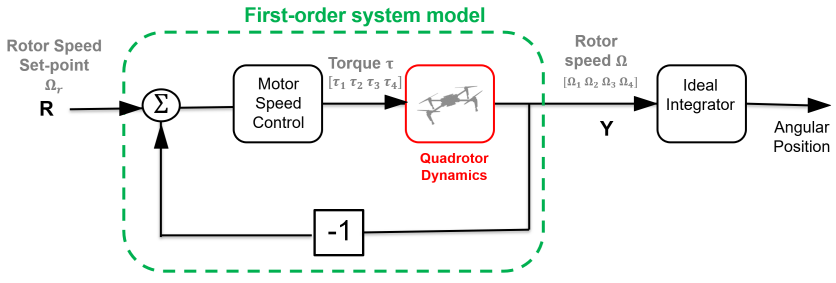
Figure 3.3: Schematic view of a first-order low-pass filter approximation, where R denotes the Cartesian velocity reference. The resulting transfer function describes the relationship between output and input.

NOTE 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.

3.3 Modelling of the Quadrotor Platform: DJI Matrix 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 3.4a.



(a)

(b)

Figure 3.4: (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 $\mathbf{\Omega} = (\Omega_1, \Omega_2, \Omega_3, \Omega_4)$ and its true ones. (b) The DJI Matrice 100 used in WARA-PS to collect data and perform experiments in the field. It has been used to make autonomous detections and avoid people during low altitude flights.

However, we are going to make use of the common "quadrotor nonlinear model" derived from various research works at ETH Zurich and well reviewed in [Bouabdallah, 2007] [23], [Blösch et al., 2010] [17] and [Kamel et al., 2017] [18].

In particular as shown in [Kamel et al., 2017] [18], we assume that the dynamics of the *internal low-level attitude* controller, already provided by the quadcopter manufacturer, to behave as a first-order system and are included in the model, as illustrated in Figure 3.5. This low-level controller will track the reference/desired roll and pitch angles (ϕ_d, θ_d) with first order behaviour.

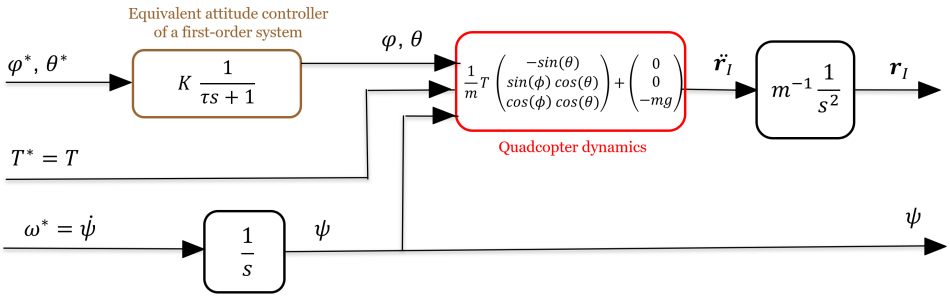


*Figure 3.5: Model of the entire quadrotor platform with the roll angle ϕ^*, the pitch angle θ^*, the total thrust T^* and the yaw rate ω^* as inputs. The outputs are the position of the quadcopter r_I and the yaw angle ψ. The dynamics of the internal **low-level attitude controller** are assumed to 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 $\mathbf{\Omega} = (\Omega_1, \Omega_2, \Omega_3, \Omega_4)$. Note that time delay, external disturbances and noise are not included in the model sketch.*

In summary, the ordinary differential equations (ODE) 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 $\boldsymbol{\xi} = (x, y, z)$ denoting the position of the center of mass of the quadrotor relative the inertial (fixed) frame, and another dynamics for rotational coordinates $\boldsymbol{\eta} = (\phi, \theta, \psi)$ representing the orientation of the quadrotor. In fact, the second subsystem in 3.4 describes the inner-loop attitude dynamics. The overall system model is expressed by the following equations:

$$\dot{\mathbf{p}} = \mathbf{v}(t) \tag{3.3a}$$

$$m\dot{\mathbf{v}}(t) = T \begin{pmatrix} -\sin(\theta) \\ \sin(\phi)\cos(\theta) \\ \cos(\phi)\cos(\theta) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} - \boldsymbol{A}\mathbf{v}(t) + \mathbf{d}(t) \tag{3.3b}$$

$$\dot{\phi}(t) = \frac{1}{\tau_\phi}(K_\phi \phi_d(t) - \phi(t)) \tag{3.4a}$$

$$\dot{\theta}(t) = \frac{1}{\tau_\theta}(K_\theta \theta_d(t) - \theta(t)) \tag{3.4b}$$

$$\dot{\psi}(t) = \omega_d \tag{3.4c}$$

where the vehicle configuration is described by the position $\mathbf{p} = (x, y, z)$ and velocity $\mathbf{v}$ 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 $\boldsymbol{R}_{IB} \in SO(3)$. Additionally, it is common to extend the model with the aerodynamic drag and external disturbance (wind) by adding the drag coefficients matrix $\boldsymbol{A}$ and $\boldsymbol{d}$, respectively, as shown in Equation 3.3.

The first-order approximation that describes the inner-loop attitude dynamics in 3.4 would provide sufficient information to the high-level MPC controller to take into account the low-level controller behavior. This is crucial in order to achieve an accurate trajectory tracking.

Note that the vehicle heading/yaw angular rate $\dot{\psi}$ of the vehicle is controlled such that it is approximately equal to a desired angular velocity $\dot{\psi} = \omega_d$, 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 [19].

NOTE 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

Chapter 4

Rendezvous Problem: UAV Quadrotor Control

Work in autonomous vehicles has grown dramatically in the past several years due to advances in *computing* and *sensing* technologies. The main components of a modern autonomous vehicle are *localization*, *perception*, and *control* [33] [Konget al. 2015].

As mentioned in [12] [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, due to high sea states for example as shown in Figure 4.1b.

Our rendezvous landing problem in this work is in itself a "trajectory tracking" problem before performing the landing manoeuvre at the end of the mission. In fact, 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 that other robot, before zeroing this relative distance in a safe and soft way.

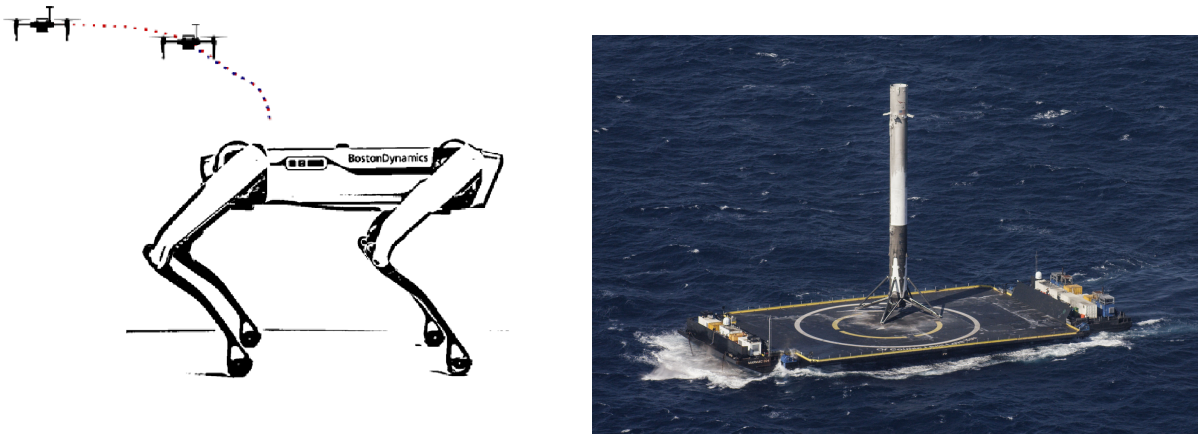
Many researchers have been working on the control problem for quadrotor aerial robotic vehicles, and they found that *the control problem to track smooth trajectories is challenging* for several reasons we quickly summarize in the following:

- First, *the system is underactuated*: there are four inputs $\mathbf{u} = (T_\Sigma, \tau_x, \tau_y, \tau_z)$, while $(\mathbf{R}, \boldsymbol{\xi}) \in SE(3)$ 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 $\boldsymbol{\xi}$ denotes the position vector of the quadrotor in the global (inertial) frame.

- Second, the *aerodynamic model* described above is only an approximate.

- Finally, *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 ($\boldsymbol{\tau}$). 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 [Mahony, Kumar and Corke, 2012] [16].

A commonly used "hierarchical" controller structure, similar to [Blösch et al., 2010] [17], [Kamel et al., 2017] [18] and many other related works, is proposed to follow in this project for solving the rendezvous landing problem. This controller will be separated into an *attitude controller* running on the micro controller (autopilot) of the DJI quadrotor and another controller, PID or Model Predictive Control (MPC), for *position tracking. This separation is useful to add robustness with respect to time delays in on-board estimation*, as stated by [Blösch et al., 2010]. Then, for better tracking performance, the closed loop dynamics of the attitude controller will be included in the system model of the quadrotor. Similar formulation is also used in WARA-PS framework.



(a) (b)

Figure 4.1: A problem of paramount importance in autonomous flight is the design of autopilots for landing in uncertain conditions. a) Schematic overview of our 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".

This section and the next one present an overview about the methods and techniques used in order to achieve our two main objectives in this work, that is, Rendezvous Landing of VTOL quadrotor drone atop Spot robot and AI Planning in robotic inspection domain. At the moment, **WARA-PS framework** includes existing tools, explained 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 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 found/read in the work of [Doherty et al., 2013] in [46].
3. **RTK-GNSS System:** To be able to perform a safe and soft landing in the experiments on the real robots, 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 maneuver, we can use a combination of Real Time Kinematic (RTK) technology with Global Navigation Satellite Systems (GNSS) augmented by common inertial sensors, for estimating the position.

4.1 Optimization-Based Control (OBC) for UAV Quadrotor Control

As mentioned in various literature, like [18, 20, 19], aerial vehicles behavior is better described by a set of *nonlinear differential equations* to capture the aerodynamic and coupling effects. In this section, a continuous time Nonlinear Model Predictive Control (NMPC) controller that considers the "full system dynamics" will be reviewed and formulated for completeness and clarity purpose. Next, we formulate the *Optimal Control Poblem* (OCP) and then we review the technique discussed in [18] that is often employed to solve the OCP to achieve real-time implementation.

In terms of autonomous cooperative landing problem, numerical papers and thesis ([29], [32], [34], [40], [41], [43]) 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, the "UAV-UGV rendezvous landing" problem will be formulated as an on-line *constrained optimization problem*, as shown in Figure 4.2.

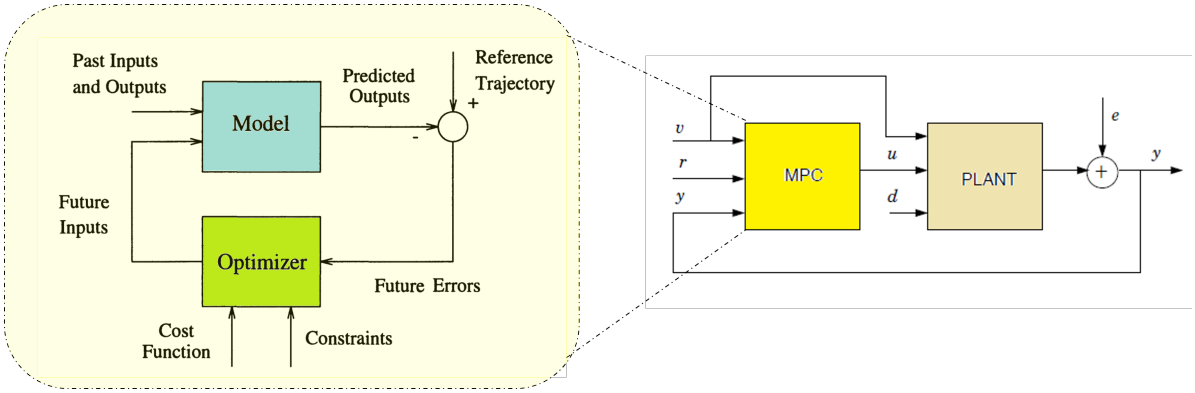


Figure 4.2: General schematic of a process controlled by MPC approach, where v,d and e denote the disturbances and r is the reference value. In the MPC basic structure in yellow, the idea of model predictive control, linear or nonlinear, is to utilize a model of the process in order to predict and optimize the future system behavior, as a function of possible optimized control actions.

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 [10].

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 [19] [Kamel, Burri and Siegwart, 2016].

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 (also called MPC) [10, 9].

4.1.1 MPC scheme: Receding Horizon Control

Receding horizon control (RHC) is another term often used alternatively to MPC. receding horizon 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 [36].

As illustrated in Figure 4.3 and explained in [7], the idea of MPC is to utilize a *model of the process* in order to predict and optimize the f*uture system behavior*:

- 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

$$x_u(k+1) = f\left(x_u(k), u(k)\right) \quad \text{where} \quad x_u(0) = x_0 \tag{4.1}$$

  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 moving horizon is the second key feature of model predictive control (MPC) [7] [Grune and Pannek, 2011].

Furthermore, by explicitly taking the *dynamics* into account in this MPC control framework, we can choose the inputs to avoid *overshoots* and *constraints violations*. The controller should also take into account potential *disturbances* to some degree [21] [Furrer et al., 2016].

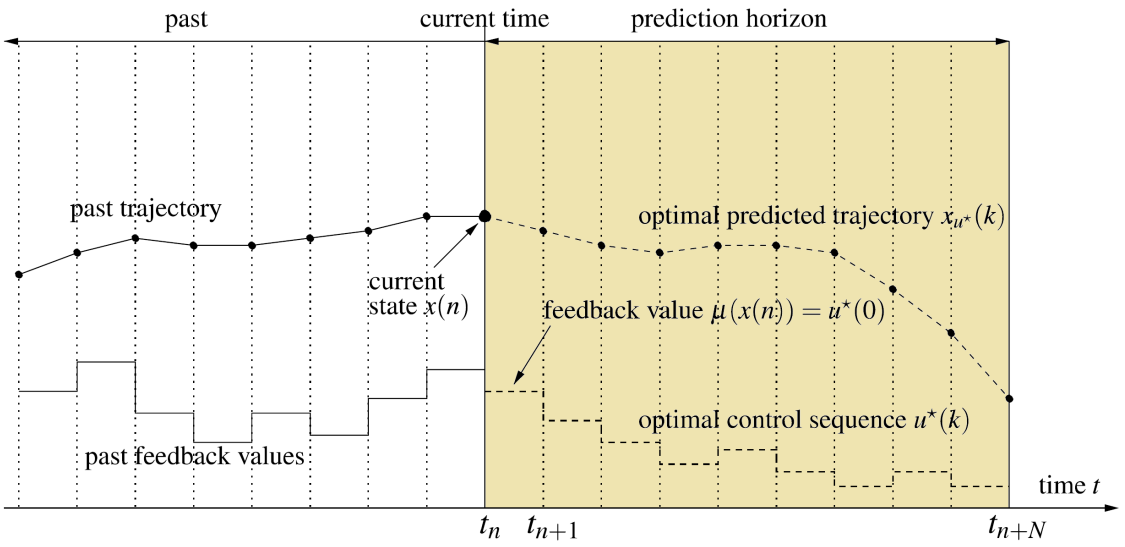


*Figure 4.3: Illustration of the model predictive control (MPC) step at time t_n. The idea of MPC is to utilize a model of the process in order to predict and optimize the future system behavior: at each sampling instant we optimize the predicted future behavior of the system over a finite time horizon 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 $k = 0, \cdots, N$, 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 moving horizon is the second key feature of model predictive control. [7] [Grune and Pannek, 2011].*

4.1.2 Nonlinear MPC Controller Formulation

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 [9, 10]

$$J = \int_{t_i}^{t_i+T} L(x,u)\ dt\ +\ V(x(t_i+T)) \tag{4.2}$$

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

$$u_{[t_i,t_i+\Delta T]} = \arg\min \int_{t_i}^{t_i+T} L(x,u)\ dt\ +\ V(x(t_i+T)) \tag{4.3}$$

$$\text{subject to} \quad \dot{x} = f(x,u) \tag{4.4}$$

$$x(t_i) = \text{current state} \tag{4.5}$$

$$g_j(x,u) \leq 0, \quad j = 1,\ldots,r \tag{4.6}$$

$$\psi_k(x(t_i+T)) = 0, \quad k = 1,\ldots,q. \tag{4.7}$$

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 and inputs 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, the constraints and the cost function. 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 $\boldsymbol{Q}_x$, $\boldsymbol{R}_u$, and $\boldsymbol{P}$. As such, the optimal control problem (OCP) can be formulated as follows

$$\min_{U} = \int_{t=0}^{T} \left(\Delta \boldsymbol{x}^T \; \boldsymbol{Q}_x \; \Delta \boldsymbol{x} \; + \; \Delta \boldsymbol{u}^T \; \boldsymbol{R}_u \; \Delta \boldsymbol{u}\right) \; dt \; + \; \Delta \boldsymbol{x}^T(T) \; \boldsymbol{P} \; \Delta \boldsymbol{x}(T) \tag{4.8}$$

$$\text{subject to} \quad \boldsymbol{x}(0) = \boldsymbol{x}(t_0) \tag{4.9}$$

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \tag{4.10}$$

$$\boldsymbol{u}(t) \in \mathbb{U}_C \tag{4.11}$$

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 4.8 using the error terms $\Delta \boldsymbol{x} = \boldsymbol{x}(t) - \boldsymbol{x}_{ref}(t)$, and $\Delta \boldsymbol{u} = \boldsymbol{u}(t) - \boldsymbol{u}_{ref}(t)$, where $\boldsymbol{x}_{ref}$ and $\boldsymbol{u}_{ref}$ 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 rendezvous state $\boldsymbol{x}$ and control input $\boldsymbol{u}$. Every time step t, the following **optimal control problem (OCP)** is solved repeatedly in real-time, which it is not a trivial task:

$$\min_{U} = \int_{t=0}^{T} \left(\left|\left| \boldsymbol{x}(t) - \boldsymbol{x}_{ref}(t) \right|\right|_{\boldsymbol{Q}_x}^2 \; + \; \left|\left| \boldsymbol{u}(t) - \boldsymbol{u}_{ref}(t) \right|\right|_{\boldsymbol{R}_u}^2 \right) \; dt \; + \; \left|\left| \boldsymbol{x}(T) - \boldsymbol{x}_{ref}(T) \right|\right|_{\boldsymbol{P}}^2 \tag{4.12}$$

$$\text{subject to} \quad \boldsymbol{x}(0) = \boldsymbol{x}(t_0) \tag{4.13}$$

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) \tag{4.14}$$

$$\boldsymbol{u}(t) \in \mathbb{U}_C \tag{4.15}$$

where $\boldsymbol{f}$ is composed of the ordinary differential equations (ODE) of the "full dynamics" model, described in 4.18. Regarding the weighting matrices, $\boldsymbol{Q}_x \geq 0$ is the penalty on the state error, $\boldsymbol{R}_u \geq 0$ is the penalty on control input error and $\boldsymbol{P} \geq 0$ is the terminal state error penalty. These weights are tuned to suit the high-level *objectives* and levels of *uncertainty* of a given application [28].

As we have reviewed above, to complete the formulation of the nonlinear MPC, we first need to define the state vector and the control input vector, as well as the predictive model represented by the ordinary differential equation of the form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$. By using Z-Y-X Euler angles, we define $\boldsymbol{R}(\psi, \theta, \phi)$ 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

$$\boldsymbol{x} = \begin{pmatrix} \boldsymbol{p} & \boldsymbol{v} & \phi & \theta & \psi \end{pmatrix}^T \tag{4.16}$$

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

$$\boldsymbol{u} = \begin{pmatrix} \phi & \theta & \dot{\psi} & T \end{pmatrix}^T \tag{4.17}$$

As described previously in 3.3 and 3.4, the overall system model representing the nonlinear "full system dynamics" of the quadrotor (meaning not linearised) can be expressed here again, for clarity purpose, by the following ordinary differential equations (ODE):

$$\dot{\mathbf{p}} = \mathbf{v} \tag{4.18a}$$

$$m\dot{\mathbf{v}} = T \begin{pmatrix} -\sin(\theta) \\ \sin(\phi)\cos(\theta) \\ \cos(\phi)\cos(\theta) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} - \boldsymbol{A}\mathbf{v} + \mathbf{d} \tag{4.18b}$$

$$\dot{\phi} = \frac{1}{\tau_\phi}(K_\phi \phi_d - \phi) \tag{4.18c}$$

$$\dot{\theta} = \frac{1}{\tau_\theta}(K_\theta \theta_d - \theta) \tag{4.18d}$$

$$\dot{\psi} = \dot{\psi}_d = \omega_d \tag{4.18e}$$

where we add the aerodynamic drag coefficients matrix $\boldsymbol{A}$ and the estimated external disturbance $\boldsymbol{d}$. 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 [43].

Additionally, as described in [6, 18], to achieve *offset-free tracking* under model mismatch, the system model is augmented with additional disturbances state $\boldsymbol{d}(t)$ to capture this model mismatch. An *observer* is employed to estimate $\boldsymbol{d}(t)$ 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 [20] [Sa, Kamel, Khanna 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 [19].

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 4.12 and can be expressed by:

$$\begin{bmatrix} \phi_{min} \\ \theta_{min} \\ \psi_{min} \\ T_{min} \end{bmatrix} \leq \boldsymbol{u} = \begin{bmatrix} \phi_{ref}(t) \\ \theta_{ref}(t) \\ \psi_{ref}(t) \\ T(t) \end{bmatrix} \leq \begin{bmatrix} \phi_{max} \\ \theta_{max} \\ \psi_{max} \\ T_{max} \end{bmatrix} \tag{4.19}$$

In the following table 4.1, we summarize the relevant numerical specifications of the UAV quadrotor DJI M100, found on the corresponding website [1], showing the limitations in the input variables

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 4.1: DJI Matrice 100 relevant limitations.

4.1.3 DJIM100 Full Dynamics System Identification

As we have mentioned, to achieve high performance with the cascade approach, adopted in this project and in many different literature as well like among others [18, 19, 20, 43], it is necessary to account for the inner-loop dynamics in the MPC trajectory tracking controller.

DJI drone manufacturer does not provide the essential scientific resources such as *attitude dynamics* and the structure of underlying *autopilot controller*. Therefore, in the paper [Sa, Kamel, Khanna, Popovic, Nieto and Siegwart, 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 [20]. Then, this critical information is used to design the subsequent MPC-based horizontal position controller addressed and formulated in [18] [Kamel, Stastny et al. 2016-2017]. 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 [2].

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.

[1] https://www.dji.com/se/matrice100/info
[2] https://github.com/ethz-asl/mav_dji_ros_interface

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 [20], 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 4.2.

- *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:

$$\frac{y(s)_\phi}{u(s)_\phi} = \frac{K_\phi}{\tau_\phi s + 1} = \frac{3.544}{s + 2.118} \qquad \frac{y(s)_\theta}{u(s)_\theta} = \frac{K_\theta}{\tau_\theta s + 1} = \frac{3.827}{s + 2.43} \tag{4.20}$$

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

$$\frac{y(s)_\phi}{u(s)_\phi} = \frac{K_\phi\ \omega_\phi^2}{s^2 + 2\ \xi_\phi\ \omega_\phi\ s + \omega_\phi^2} = \frac{26.37}{s^2 + 5.32s + 27.04} \tag{4.21}$$

$$\frac{y(s)_\theta}{u(s)_\theta} = \frac{K_\theta\ \omega_\theta^2}{s^2 + 2\ \xi_\theta\ \omega_\theta\ s + \omega_\theta^2} = \frac{28.86}{s^2 + 6.00s + 27.45} \tag{4.22}$$

- *Yaw and height dynamics*: The input commands of yaw and height are rates, $u_{\dot\psi}$, $u_{\dot z}$, 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, $\dot z^*$.

$$\frac{y(s)_{\dot\psi}}{u(s)_{\dot\psi}} = \frac{K_{\dot\psi}}{\tau_{\dot\psi} s + 1} = \frac{5.642}{s + 5.268} \qquad \frac{y(s)_{\dot z}}{u(s)_{\dot z}} = \frac{K_{\dot z}}{\tau_{\dot z} s + 1} = \frac{3.342}{s + 2.99} \tag{4.23}$$

Similarly, the corresponding identified second-order dynamic models are

$$\frac{y(s)_{\dot\psi}}{u(s)_{\dot\psi}} = \frac{K_{\dot\psi}\ \omega_{\dot\psi}^2}{s^2 + 2\ \xi_{\dot\psi}\ \omega_{\dot\psi}\ s + \omega_{\dot\psi}^2} = \frac{593.3}{s^2 + 89.0s + 549} \tag{4.24}$$

$$\frac{y(s)_{\dot z}}{u(s)_{\dot z}} = \frac{K_{\dot z}\ \omega_{\dot z}^2}{s^2 + 2\ \xi_{\dot z}\ \omega_{\dot z}\ s + \omega_{\dot z}^2} = \frac{25.43}{s^2 + 7.16s + 24.8} \tag{4.25}$$

where $y(s)_{\dot\psi}$ 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 4.2: DJI Matrice 100 identified dynamics summary, found by [20] [Sa, Kamel, Khanna 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 [20].

4.1.4 MPC Controllers Implementation

We remember that in the adopted cascade control approach, the high-level MPC-based controller generates attitude commands for the low-level controller running on the autopilot, already provided on the UAV platform. A classical Linear Model Predictive Controller (LMPC), based on a linearised model of the UAV, is presented in [19] and compared against a more advanced Nonlinear Model Predictive Controller (NMPC) that considers the full system model, as the one adopted 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 [18] [Kamel, Stastny, et al. (2016-2017)].

As explained in [18, 19, 20], to implement the *linear MPC*, an efficient QP solver using `CVXGEN` code generator framework by [Mattingley and Boyd (2012)] [36] is employed to solve the optimization problem every time step.

Whereas the *nonlinear MPC* is implemented by solving the optimization problem in 4.12 every time step using an efficient solver generated using `ACADO toolkit` by [Houska et al. (2011)] [37].

Two useful examples on how to implement an MPC problem using ACADO tools and CVXGEN are reviewed in the Appendix.

4.2 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) [3].

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.

4.2.1 Setting up ROS nodes

Robot Operating System (ROS) is 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 project. WARA-PS system is also based on the use of ROS, where many aspects of the system require "publish/subscribe" communication channels, that is, *ROS topics*. Specifically, functionalities are divided into nodes that can call each others through "remote procedure calls", that is, *ROS services*.

4.2.2 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, Kamel et al. 2017] [20], it is strongly not recommended to use `ROS services` for sending control commands as designed by the manufacturer [4]. 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 [20]. Furthermore, with the OSDK, the developers can

[3] https://jack-kawell.com/2020/06/12/ros-wsl2/
[4] http://wiki.ros.org/ROS/Patterns/Communication

control the UAV without a remote controller, install third party sensors, and execute precise trajectories [31].

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 [20, 31].

Chapter 5

Rendezvous Simulation: Design and Implementation

Automatic position control is needed to control our UAV (DJI Matrix 100) in the rendezvous landing tssk, one plan is to follow the same type of control prepared by Wolfgang Hoenig in his success with the crazyflie metapackage [1]. This package was developed as part of his research at the ACT Lab at the University of Southern California.

Another plan would be to use MPC design of which the implementation will be considered in future works, because of lack of time. In non-linear MPC we have the advantage of incorporating a plant model with non-linear dynamics, a cost function with high interpretability and non-linear constraints.

In principle, Rendezvous problem, here cooperative autonomous landing between the UAV quadrotor and Spot robot, is defined to be achieved when zeroing all the following *relative position error equations*

$$\begin{aligned} \Delta x &= x_{uav} - x_{spot} \\ \Delta y &= y_{uav} - y_{spot} \\ \Delta h &= h_{uav} - h_{spot} \end{aligned} \tag{5.1}$$

In the MPC control methods, we would make use of the nonlinear MPC "trajectory tracking" controller based on the paper [18] [Kamel et al., 2017] at ETH Zurich, and of which a similar one is used in the WARA-PS simulator. This nonlinear MPC controller consists of a cascade structure as illustrated in Figure 5.4. In this case, we need to make some modifications by replacing the error vector in the objective function of the optimization problem in MPC scheme by the error vector defined in 5.1. *In fact, we are going to replace the reference x_{ref} in Figure 5.4 by the position of Spot robot.* As such, the quadrotor UAV need to track Spot position and perform the landing successfully.

In general, precise trajectory tracking is a demanding feature for aerial robots (UAV) in order to successfully perform the required rendezvous task in the experiments, especially when operating in realistic environments where external disturbances may heavily affect the flight performance and when flying in the vicinity of some structure or the ground robot Spot itself.

In the experimental work 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* in 5.1 defined between the current pose of the quadruped and the pose of the quadrotor.

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

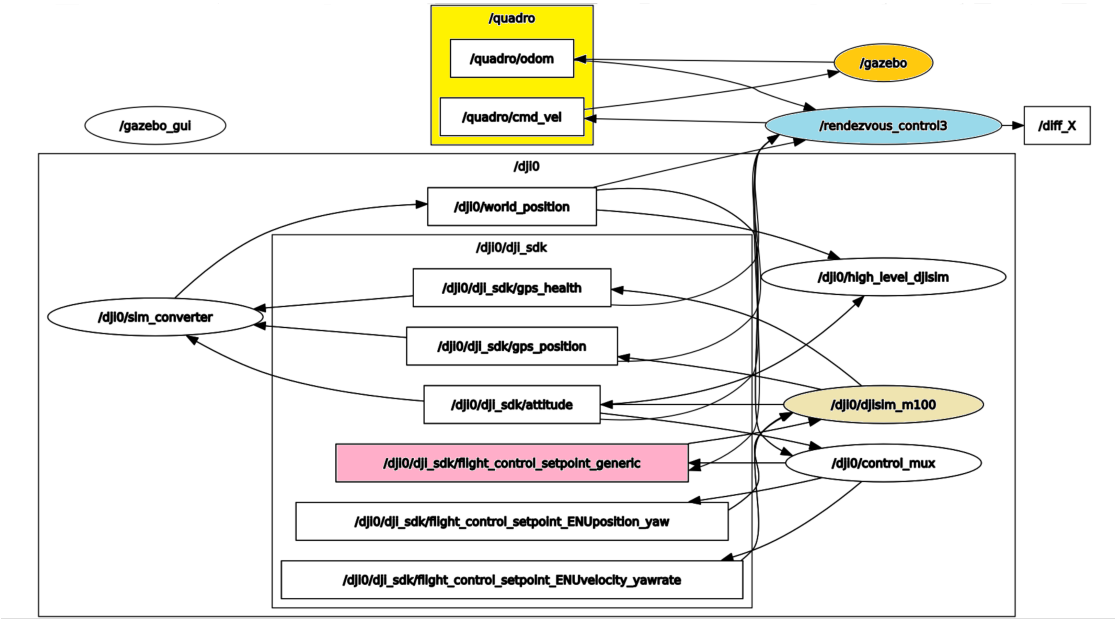


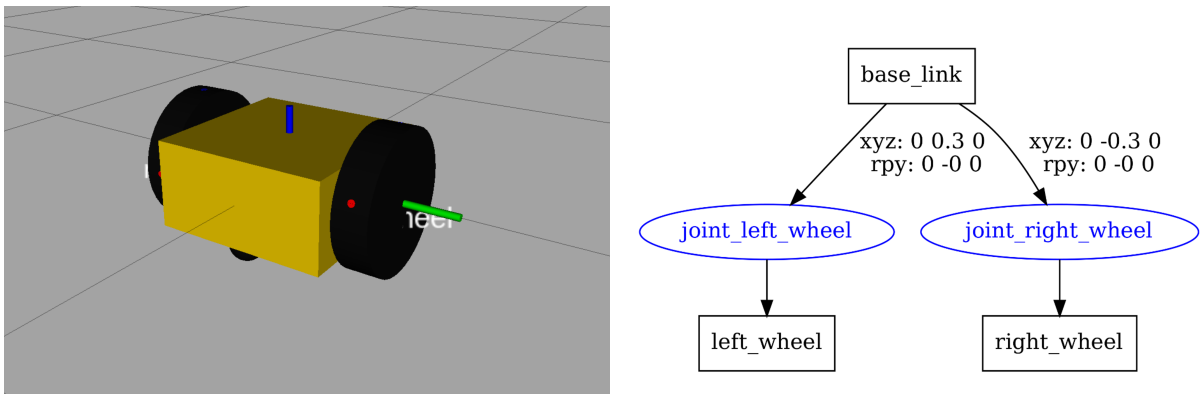
Figure 5.1: RQT graph visualizing the different nodes of our rendezvous simulation and their communication in ROS using `rqt_graph`

.

5.1 Simulation using Gazebo-ROS and WARA-PS Framework

When the WARA-PS DJI simulator and development environment is 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 controller (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 5.1 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 5.2: (a) Our 2-wheeled robot named "Quadro" in RViz. 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:* `urdf_to_graphiz quadro.urdf` *and* `evince quadro.pdf`

5.1.1 Designing of ground mobile robot simulation model using URDF

As a two-wheeled mobile robot, we build a Gazebo-ROS simulation model using URDF for substituting Spot robot model in order to simulate the movement of Spot in the rendezvous maneuver. This robot model inherits most of the necessary functionality from [Carol Fairchild and Thomas Harman, 2017] book: *ROS Robotics by Example*, 2 Edition [5] and from .

Our 2-wheeled robot named "Quadro" is basically a robot composed by 3 links and 2 joints, as shown in Figure 5.2a. In URDF every robot needs a base link, in this case, the gold chassis is in charge of connecting the two wheels of the robot.

After having a working URDF model of our two-wheeled robot, we need to launch it into Gazebo and move it around. First, we must make some modifications to our URDF file to add simulation-specific tags so that it properly works in Gazebo. Gazebo uses SDF, which is similar to URDF, but by adding specific Gazebo information, we can convert our quadro model file into an SDF-type format [5].

5.1.2 Adding Gazebo Plugins: Setting up Differential Drive Controller

Gazebo plugins need to be added to the robot model code to enable control of the robot movement in simulation. Gazebo plugins are needed to simulate controllers that publish `Twist` ROS messages for motor commands. This Gazebo plugin is implemented for setting up `differential_drive_controller` that will publish necessary values using different topics. The `cmd_vel` is the topic in which we send Twist messages in ROS (`geometry_msgs`), by reading from the keyboard or by implementing a python node and then publishing to Twist. This twist message contains two subsections: linear velocity and angular velocity.

In ROS conventions, we use the `rostopic pub` 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. The Python script send a `Twist` message on the `cmd_vel` topic to move Quadro robot forward 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 the robot movement is the same as in turtlesim in ROS tutorials [2]. In this tutorial series, one can learn how to create python scripts to move the turtle, in order to practice the ROS basics.

5.2 PID Rendezvous Control with ROS

A first important step for autonomous flight of a quadcopter is hovering in place [44]. This also requires the ability to take off from the ground and land after the flight. All these basic motions require a *position controller*, which takes the UAV's current position as input in order to compute new commands for the DJI quadrotor. Hence, this position controller replaces the teleoperating human.

Within the new created `quadro_sim` package, we implement a controller in Python that uses 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` in the folder `quadro_sim/config`.

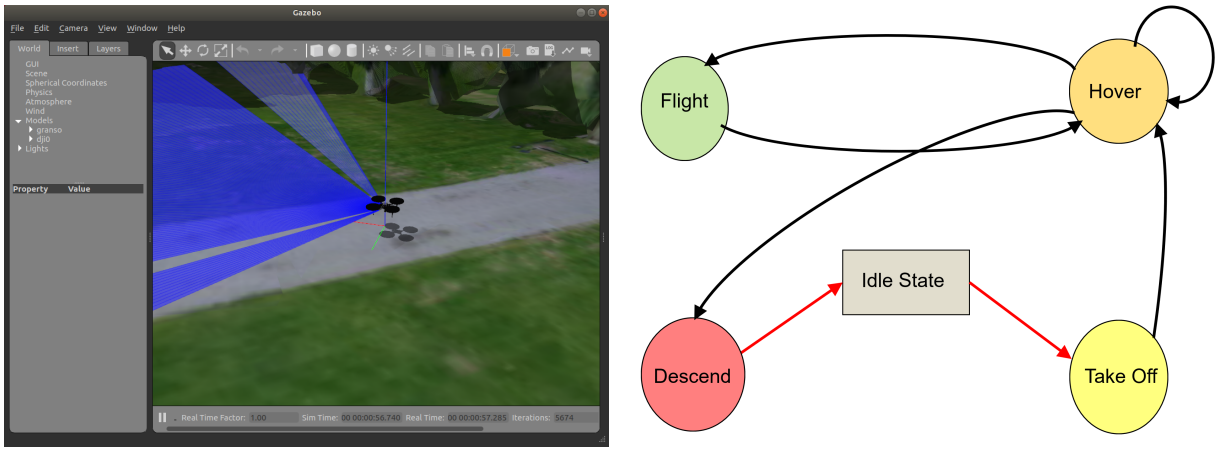
We remember that the DJI M100 is controlled by a cascaded design controller, where the inner attitude controller is part of the firmware. Our `rendezvous_control` node run a separated outer controller (PID or MPC) which takes the current and target positions as input and produces a setpoint, attitude and thrust, for the inner controller.

In this project, the target position refers to the current position of the simulated ground mobile robot that can be received from the robot position feedback topic: `/quadro/odom`, of the built-in message type in ROS `nav_msgs/Odometry`. In ROS, we can get the definition of odometry from the following command:

```
rosmsg show nav_msgs/Odometry
```

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

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.



(a) (b)

Figure 5.3: (a) Our WARA-PS UAV DJI simulated and loaded in Gazebo. (b) In the controller 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 included idle, takeoff, hover, flight, and descend.

5.2.1 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 `/dji_sdk/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" [3] 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:

```
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:

```
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
```

5.2.2 Rendezvous control ROS node

In the Rendezvous controller node, using PID control or MPC, the controller software computes the difference between the DJI's current position and the goal position on top of Quadro, either hovering or landing target, to send correction commands to the UAV quadrotor to fly closer to the goal position.

[3] http://wiki.ros.org/dji_sdk

For our rendezvous mission, the Python script *control_rendezvous.py* creates the *rendezvous_control* node to handle the process of identifying the locations of the DJI quadrotor and the ground mobile 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 quadro mobile robot. 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
```

The output topic is

```
/dji0/dji_sdk/flight_control_setpoint_generic
```

where the **control flag** value to control roll, pitch, thrust and yaw rate, should be

```
0x00 | 0x20 | 0x08 | 0x00 | 0x00 = 0x2A
```

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`.

5.2.3 Using state machine to perform Rendezvous control

Finite-state machines are powerful mechanisms for controlling the behavior of a system, especially robotic systems [5]. ROS has implemented a state machine structure and behaviors in a Python-based library called SMACH [4]. However, for prototyping our Rendezvous control ROS node we implement our simple state machine in Python without using SMACH library.

Adapting to the work in [44], 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. 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.

This *iteration* continues every 20 milliseconds with a "new position" for DJI detected and a "new correction" computed and sent. This is a *closed-loop system* that computes the difference between positions, and then commands the UAV to fly in the direction of the goal position. 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.

5.2.4 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, and the same can be done for descend flight state, where this is done using `Empty` type for both services and publishing,

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`:

[4] http://wiki.ros.org/smach/Tutorials

```
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 controller, 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, 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`

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

5.3 Possible Extensions: Nonlinear MPC Control Methods, Two Strategies

For landing of the UAV quadrotor on top of Spot robot, we are going to review and investigate two alternative landing control strategies using MPC. Both these two MPC controllers will guide the drone to track Spot robot at the landing task. However, these two methods are different such that one of them is "more robust" and take in consideration the safety of the *touchdown step*, whereas in the other strategy only the *tracking and landing* are considered. If the experimental/simulated results does not show satisfactory performance with the first method then the second method will be adopted and implemented.

At the end, we wish to control the position of the VTOL quadrotor and this problem requires stabilization and control of the attitude dynamics. For this reason, we need to make use of the "inner/outer loop control architecture".

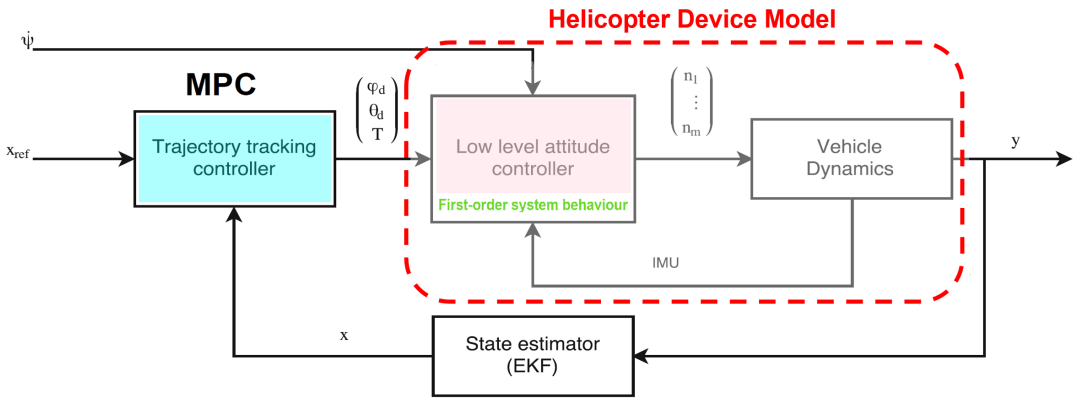


Figure 5.4: The UAV controller sketch for precise trajectory tracking based on NMPC in [18]: A cascade control approach is used where a high-level NMPC generates attitude commands for a low-level controller running on the autopilot. The dynamic behavior of the attitude controller is considered as a first-order 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, that are tracked by an inner attitude tracking controller.

5.3.1 Method 1: ETHZ/WARA-PS based approach

In this method, we will make use of the ETH-based nonlinear MPC "trajectory tracking" controller, also used by WARA-PS simulator, after adjusting the error vector related to the relative positon. This modified nonlinear MPC controller consists of a cascade structure with a centralized position control, as illustrated in Figure 5.5.

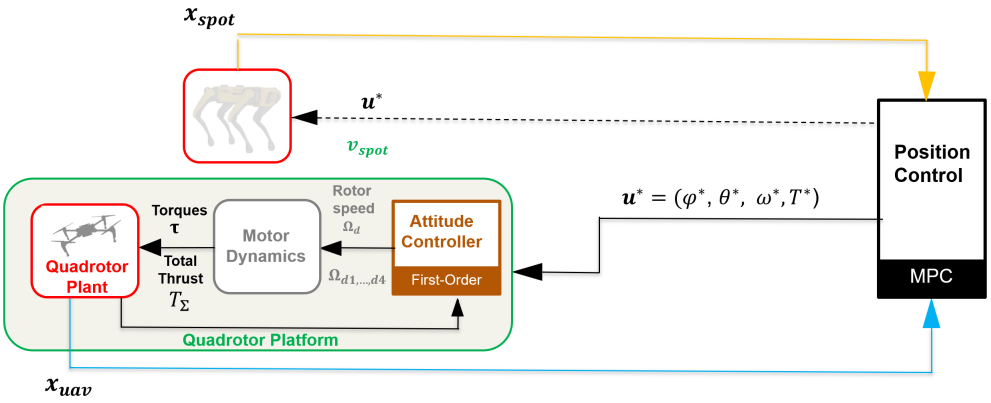


Figure 5.5: The centralized MPC architecture: To generate feasible rendezvous trajectories for the vehicles, we use Model Predictive Control (MPC). The optimal solutions from the MPC, $\mathbf{u}^* = (\phi^*, \theta^*, \omega^*, T^*)$, are sent to ***low-level autopilots*** *that compute the actuator inputs. Usually, for multi-rotor system Cascade control approach is used. As such, 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.*

5.3.2 Method 2: Separated MPC approach

In this method, robustness issues can be taken into account for achieving a safe/soft touchdown at the landing maneuver. This is similar to several works at KTH in [40, 41, 43], as well in [Mellinger, Shomin, Kumar, 2010][14] and [Lee, Ryan, Kim, 2012] [24].

In particular, the method presented in [Linnea Persson, 2021] PhD thesis [43] 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 Δx, Δy is completed, but still can guarantee that Δh will not go to zero before it is safe to do so.

Objectives and Constraints The same here, Rendezvous problem between the UAV quadrotor and Spot robot is defined to be achieved when zeroing all the relative position error equation shown in 5.1.

The "planned trajectory" can be executed using RTK GPS, where the measurements are fed back to the MPC control to compute an appropriate **error vector** defined between the current pose of the quadruped and the pose of the quadrotor. In addition to this objective, we need also to consider the following objectives and constraints:

- 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.
- The vehicles should 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}.

The requirement that $(\Delta x, \Delta y) \to 0$ before $\Delta h \to 0$ is crucial for the safety of the maneuver. Otherwise there is a large risk of the vehicles crashing into each other.

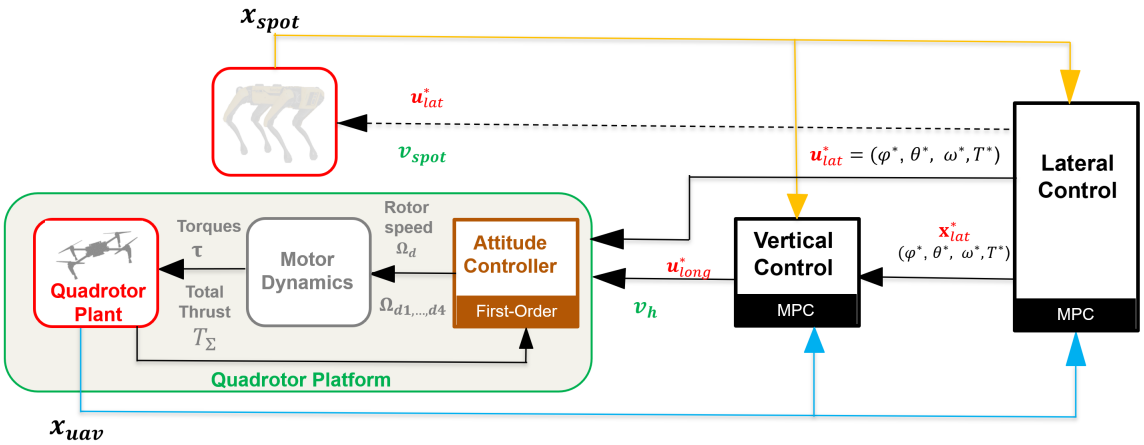


*Figure 5.6: The separated MPC architecture: To generate feasible rendezvous trajectories for the vehicles, we use Model Predictive Control (MPC). However, the MPC is now separated into two parts, one acting in the horizontal plane (lateral MPC controller) and one in the vertical (vertical MPC controller). Then, the optimal solutions from the MPC (u^*_{lat}, u^*_{long}) are sent to **low-level autopilots** that compute the actuator inputs. Usually, for multi-rotor system, 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, a.k.a Cascade control approach*

Low-level Control In the MPC implementations, we choose to control the attitude (roll and pitch), yaw rate and vertical velocity. It is assumed that the "low-level controllers" use the roll angle ϕ^*, pitch angle θ^*, yaw rate $\dot{\psi}^*$, and vertical velocity $\dot{h}^*$ as "control inputs". The same here in this method, the dynamics of the internal low-level controllers are included in the model.

As shown in [43] [Persson, 2021, p 43], the attitude control dynamics, already provided by the quadrotor manufacturer, are assumed to be approximated as second-order systems. For roll and pitch, 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

$$G_\phi(s) = \frac{K_\phi \ \omega_\phi^2}{s^2 + 2 \ \xi_\phi \ \omega_\phi \ s + \omega_\phi^2} \qquad G_\theta(s) = \frac{K_\theta \ \omega_\theta^2}{s^2 + 2 \ \xi_\theta \ \omega_\theta \ s + \omega_\theta^2} \tag{5.2}$$

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

$$\Psi(s) = \frac{1}{s} \ G_\psi(s) = \frac{1}{s} \ \frac{K_\psi}{\tau_\psi \ s + 1} \dot{\Psi}^*(s) \tag{5.3}$$

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)

Table 5.1: Horizontal control.

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

$$\dot{H}(s) = \frac{K_v}{\tau_v \ s + 1} \dot{H}^*(s) \tag{5.4}$$

Inputs	Variables	Limits
Vertical speed	$\dot{h}$	-5 to +5 m/s
Height	h	0 to 120 m
Thrust	T	0% to 100%

Table 5.2: Vertical control.

5.4 Robust Rendezvous Landing

Sequence of the controllers can be designed to robustly land on a small surface on Spot, similar to [Mellinger, Shomin, Kumar, 2010][14] and [Lee, Ryan, Kim, 2012] [24]. .

We assume the position and velocity of the quadrotor are available for the feedback MPC controller 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 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 5.7. First, the quadrotor is controlled to fly/align above the desired landing location on Spot. 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 labeled 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 fly/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.

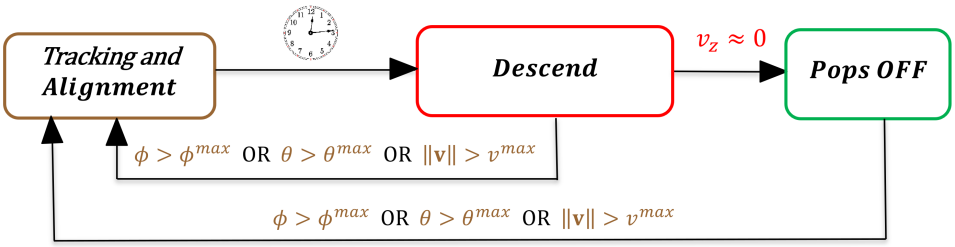


Figure 5.7: Control strategy for robust rendezvous landing: Note that event-triggered transitions are labeled with the event and time-triggered transitions are denoted by clocks. While in the Descend state the quadrotor waits to sense an event before transitioning to the next state. If an error is sensed the quadrotor is controlled to fly/align at its original location.

5.5 Estimating Position - 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 [16] [Mahony, Kumar, Corke, 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 [24].

Instead, for achieving high-accuracy positioning in the landing maneuver, we are going to use a combination of Real Time Kinematic (RTK) technology [5] with Global Navigation Satellite Systems (GNSS), such as GPS, GLONASS, Galileo, BeiDou, and other constellation systems. As mentioned in WARA-PS

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

portal [1], the system can easily be integrated with different types of vehicles and platforms, making them compatible with different types of UAVs in the project.

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].

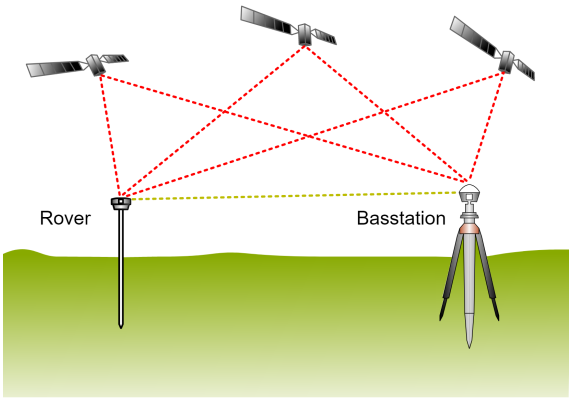


Figure 5.8: The Real-Time Kinematic (RTK) concept: RTK is the application of surveying (examination) to correct for common errors in current satellite navigation (GNSS) systems. ***Carrier-phase tracking****: RTK follows the same general concept in GNSS but uses measurements of the phase of the satellite 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.*

The demand for scalable "high precision technology" is growing rapidly, as evident in the automotive world with next generation ADAS (Advanced driver-assistance systems) and V2X (Vehicle-to-everything) applications, and in robotics with applications such as UAVs and robotic lawnmowers [7].

NOTE This RTK-GNSS system in WARA-PS comes from another project that Combitech is a part of, called DRIWS (Digital Runway Incursion Warning System). Here are some related links about RTK algorithm to integrate in the project, provided by my supervisor Prof. Anders Robertsson at the department of Automatic Control at LTH:

- https://gitlab.control.lth.se/elt14mn1/gnss
- https://gitlab.liu.se/lrs/lrs_wara_rtk

[6] https://inertialsense.com/how-rtk-provides-gps-centimeter-level-accuracy/
[7] https://www.u-blox.com/en/technologies/high-precision-positioning

Chapter 6

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 using domain-independent planning techniques [62]. A sample of abstraction hierarchy for our robotic inspection mission is illustrated in Figure 6.1.

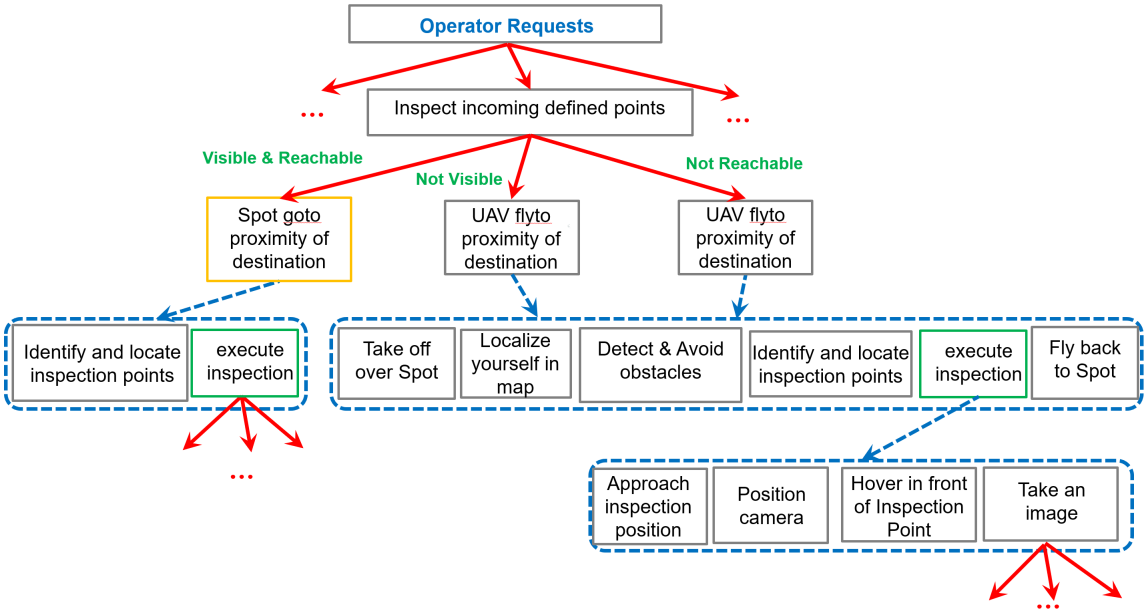


*Figure 6.1: Multiple levels of abstraction for the inspection domain: Each solid red arrow indicates a "refinement" of an abstract action into more concrete ones. Each dashed blue arrow maps a task into a **plan of actions**. For unmanned aircraft, actions may for example include taking off and landing, flying between waypoints or to a distant destination, hovering, positioning cameras and taking pictures.*

6.1 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 [61].

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 6.1. 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 6.8.

as illustrated in Figure 6.3a, 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.

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

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.

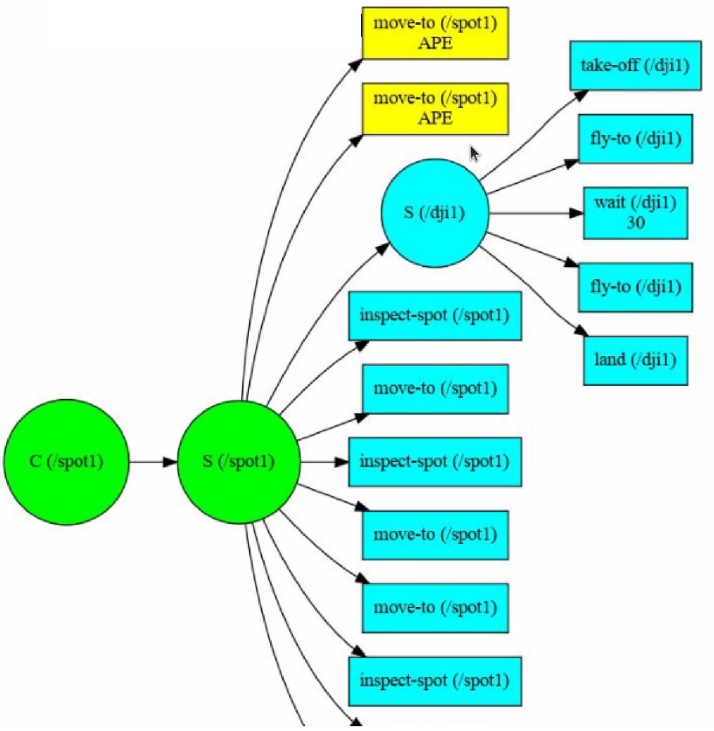
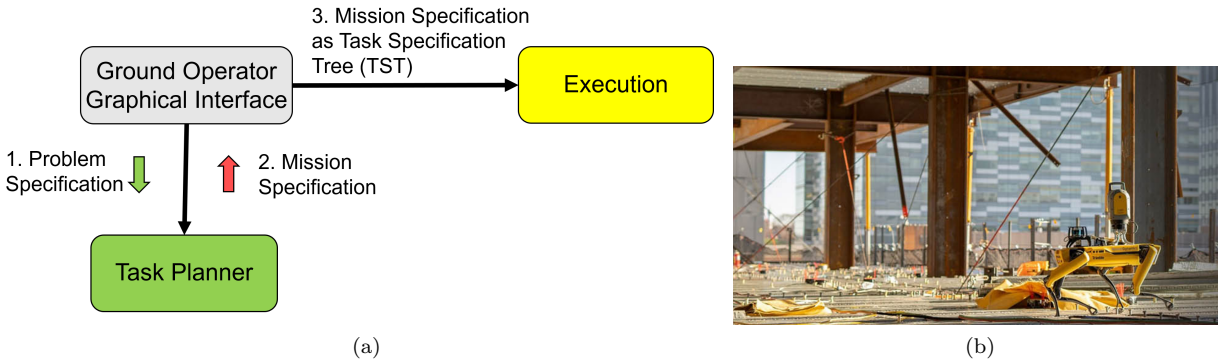


Figure 6.2: A fragment of the Task Specification Tree (TST) converted from the plan solution using TST factory in the WARA-PS framework: The corresponding TST uses elementary action nodes, corresponding to elementary actions of type `inspect-spot`, `move-to`, `fly-to`, *etc. Furthermore, it requires 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 6.3: (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)*

6.2 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 [51] [Ulam et al., 2006], 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 6.4 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 6.1. Again as stated in [62], 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 [76] [Torreno 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.

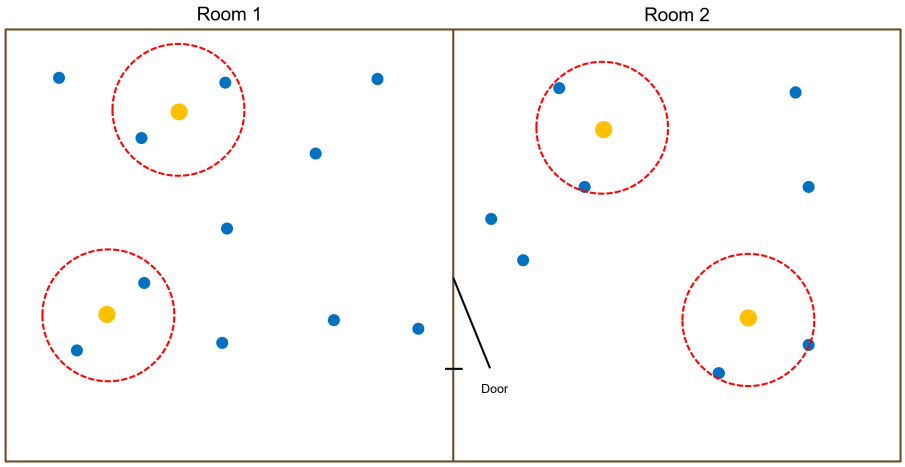


*Figure 6.4: Conceptual overview depicts the **automated robotic inspections (ARI) world**: "Room1" and "Room2" formalize the space to accommodate robots and inspection points and allow the actions needed to accomplish the goal of inspection missions. Yellow circles represent Spot robot, whereas the 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.*

6.2.1 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 [71].

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 [78] [Komenda et al., 2016]

Another important feature of these PDDL planning models, according to various literature like in [45, 69, 76], PDDL separates the model of the planning problem in two major parts as follows, defined by the respective main ingredients shown in Figure 6.5:

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 6.5a.

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 6.5b.

```
(define (domain DOMAIN_NAME)
  ( :requirements [:strips] [:equality] [:typing] [:adl])

  ( :predicates (PREDICATE_1_NAME ?A1 ?A2 ... ?AN)
                (PREDICATE_2_NAME ?A1 ?A2 ... ?AN)
                ...)

  ( :action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]
  )

  ( :action ACTION_2_NAME
     ...)
...)
```

(a)

```
(define (problem <title>)
        (:domain <domain-name>)
        ( :objects
                <object-list>
        )

        ( :init
                <predicates>
        )
        ( :goal
                <predicates>
        )
)
```

(b)

*Figure 6.5: (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.

6.2.2 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 [69].

In the following, we formulate a small-scale practical *robotic inspection* problem in a basic PDDL encoding. As shown in Listing 6.1, 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 [63]. 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 [69]. In fact, a domain set of requirements allow a planner to quickly tell if it is likely to be able to handle the domain [63].

Listing 6.1: The planning variables declaration in the domain file - Robotic inspection task

```
(define (domain inspections2)

    (:requirements :strips :typing :fluents :action-costs)

    (:types
        spot uav position - object
    )
```

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 [63].

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 [69].

Our predicates in Listing 6.2 indicate if Spot and UAV are at a particular 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 6.2: The predicates in the domain file - Robotic inspection task

```

    (:predicates
        ;; Reachability of locations for Spot robot
```

```
        (spot-reachable ?from-spot-pos ?to-spot-pos - position)

        ;; Visibility: when one objective is visible for Spot
        (spot-inspectable ?spot-pos ?inspect-pos - position)

        (uav-inspectable ?spot-pos ?inspect-pos - position)

        (spot-at ?spot - spot ?spot-pos - position)

        ;; predicate to express the goal:
        (inspected ?inspect-pos - position)

        (uav-at ?uav - uav ?inspect-pos - position)

        (attached ?uav - uav ?spot - spot)

    )
```

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 6.3, 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 6.3: The functions in the domain file - Robotic inspection task

```

    (:functions
        ;; Actions cost
        (total-cost) - number

        ;; Operation time (optimization metric) [s]
        ;(total-time) - number
    )
```

In Listing 6.4 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 [69] 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 6.4: The allowed actions in the domain file - Robotic inspection task

```

    (:action move-spot
        :parameters (?spot - spot
```

```
                    ?uav - uav
                    ?from-spot-pos - position
                    ?to-spot-pos - position)

        :precondition (and (attached ?uav ?spot)
                           (spot-at ?spot ?from-spot-pos)
                           (spot-reachable ?from-spot-pos ?to-spot-pos))

        :effect (and (spot-at ?spot ?to-spot-pos)
                     (uav-at ?uav ?to-spot-pos)
                     (not (spot-at ?spot ?from-spot-pos))
                     (increase (total-cost) 1))

    )
    (:action inspect-spot
        :parameters (?spot - spot
                     ?spot-pos - position
                     ?inspect-pos - position)

        :precondition (and (spot-inspectable ?spot-pos ?inspect-pos)
                           (spot-at ?spot ?spot-pos)
                           (not (inspected ?inspect-pos)))

        ; This effect makes predicate (inspected ?inspect-pos) true
        ; so that the robot won't return to this point later in the plan.
        :effect (and (inspected ?inspect-pos)
                     (increase (total-cost) 2))

    )
    (:action inspect-uav
        :parameters (?dji - uav
                     ?dji-pos - position
                     ?inspect-pos - position)

        :precondition (and (uav-inspectable ?dji-pos ?inspect-pos)
                           (uav-at ?dji ?dji-pos)
                           (not (inspected ?inspect-pos)))

        :effect (and (inspected ?inspect-pos)
                     (increase (total-cost) 1))
    )
)
```

6.2.3 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 6.5 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 6.6 . 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 6.5: The objects defined in a small problem instance file - robotic inspection problem

```
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 6.6, 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 [69].

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 6.6: The initialized states defined in a small problem instance file - robotic inspection problem

```
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)
```

```
        (spot-reachable sp4 sp5)

        (spot-reachable sp5 sp0)
        (spot-reachable sp5 sp2)
        (spot-reachable sp5 sp4)


        (spot-inspectable sp2 ip2)
        (spot-inspectable sp3 ip3)
        (spot-inspectable sp4 ip4)
        (spot-inspectable sp5 ip5)

        ; consider the case where ip6 is not reachable by Spot
        (uav-inspectable sp5 ip6)

    )
```

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 6.7, 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 6.7: The goal defined in a small problem instance - robotic inspection problem

```

    (:goal
        (and  (inspected ip2) (inspected ip3) (inspected ip4)
              (inspected ip5) (inspected ip6))
    )

    (:metric
        minimize (total-cost)
    )
)
```

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 [69].

6.3 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 6.8. 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 6.8. 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 6.2.

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 [62]. 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* [71].

Listing 6.8: The retrieved mission specification (planner output): The following 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.

```
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)
```

6.3.1 Quality Evaluation of Candidate Plans

In [Ghallab, Nau, Traverso, 2016] [62] and [Haslum et al., 2019] [69] 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] [77], 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 [62]. 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] [71].

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 [69], there is a clear difference in emphasis:

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

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 [71].

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 [69], 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 [47].

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)` [71].

6.4 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 [69]. 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 [72].

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 6.8, 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 and a plan, and determines if the plan solves the problem [69]. An alternative implementation of a plan validator for PDDL is *INVAL* [7].

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

After exploring different planning literature similar to our inspection domain, including exploration, survey/coverage task, mine removal, logistics and other domains and examples [45, 69, 77, 71, 76, 51], 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] [52]. The FD planning method can only handle one specific function `(total-cost)` [71] (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] [53].
- *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] [54]. However, it does not consider the optimization metric properly, according to individual results for survey domain in [71].
- *POPF*: *Partial-order planning* is an intuitively attractive strategy, but has proved difficult to achieve efficiently. With POPF [Coles et al., 2010] [55] they 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 [71]. Before that, VHPOP (Younes and Simmons 2003) was the only recent partial-order planner to exploit the approach for temporal planning[55].
- *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 et al., 2002] [56].
- *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 [71]. For more details, the reader is referred to [Benton et al., 2012] [57].

6.4.1 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" [72]. 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

[8] https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html
[9] http://prometeo.ing.unibs.it/lpg

research works in this field like the online portfolio selector of *Delfi* using Deep Learning techniques [Katz et al., 2018] [73] and the explainable planner selection of [ferber and Seipp, 2022] [72] 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.

How to create the *online portfolio planner* called *Delfi* is described in [Katz, Sohrabi, Samulowitz, Sievers, (2018)] [73]. 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 [72]. "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 $\boldsymbol{v} \in \mathbb{R}^N$ containing the values of the input features.

In conclusion, 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 [72].

Chapter 7

Deployed Platforms: Spot and DJI Matrix 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 this section, we are going to give an overview describing briefly these two platforms deployed in our project.

7.1 UGV Spot Quadruped Robot

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.

7.1.1 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].

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

As illustrated in Figure 7.1, 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.

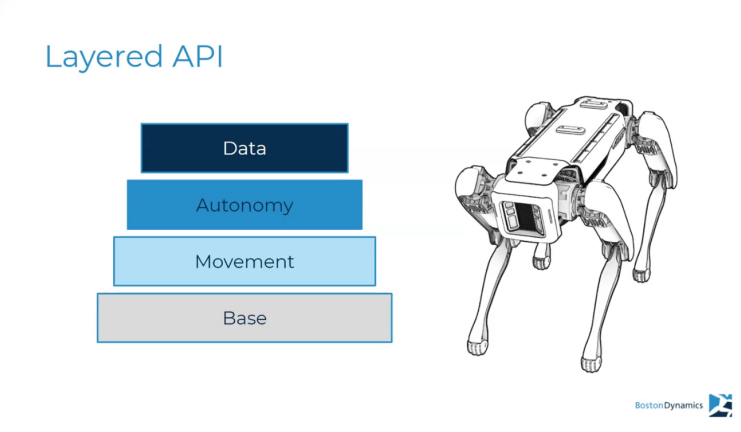


Figure 7.1: Spot API consists of the layers: Data, Autonomy, Movement and Base layers. This flexible layered API can be used to develop tailor-made payloads, controls and behaviors. Courtesy: Boston Dynamics

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`.
  - 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:

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

- Behavior trees for complex sequences of actions
- Runs without client involvement
- Supports complex behavior

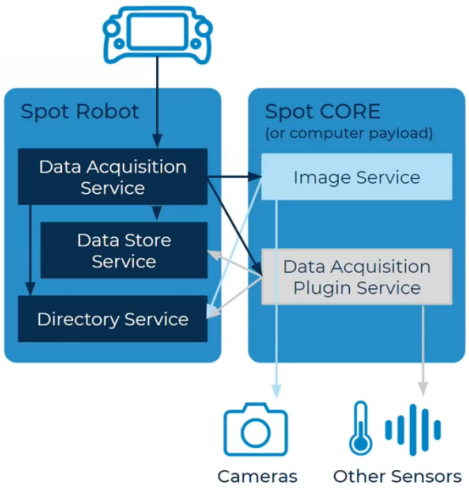


Figure 7.2: Spot Data acquisition (DAQ) architecture used to create a data acquisition plugin service and run the service to communicate with the data acquisition service on-robot. DAQ functionality is a powerful tool to use in "photogrammetry applications" in Agriculture, Building & Construction, Geospatial 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 7.3: (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

7.1.2 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 7.3a.

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 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

[4] https://www.bostondynamics.com/resources/webinar/developing-payload-part-i
[5] https://www.bostondynamics.com/spot/resources/developing-a-payload-part-ii

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.

7.1.3 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 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.

7.2 UAV Quadrotor Platform: WARA-PS DJIM100

In this section, we start to describe the WARA-PS DJIM100 quadrotor platform. The DJI Matrice 100 is a quadcopter used in WARA-PS [1] 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].

[6] https://www.bostondynamics.com/resources/blog/new-approach-predictive-maintenance-challenges
[7] https://resources.bostondynamics.com/hubfs/Content%20Files/Boston_Dynamics_Thermal_Inspection_Whitepaper.pdf
[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 7.4: 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[1]

7.2.1 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 7.5: (a) Intel NUC computer in lightweight housing mounted on-board. (b) Zenmuse Z3 color camera with gimbal.

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.
- 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.

[11] https://portal.waraps.org/technical-specifications-of-dji-matrice-100/
[12] https://www.dji.com/zenmuse-xt

7.2.2 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 7.6:

- 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.

(a) (b)

Figure 7.6: (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.

7.2.3 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 7.7 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 !?

[13] http://www.magewell.com/usb-capture-hdmi
[14] https://www.ubnt.com/airmax/bulletm/

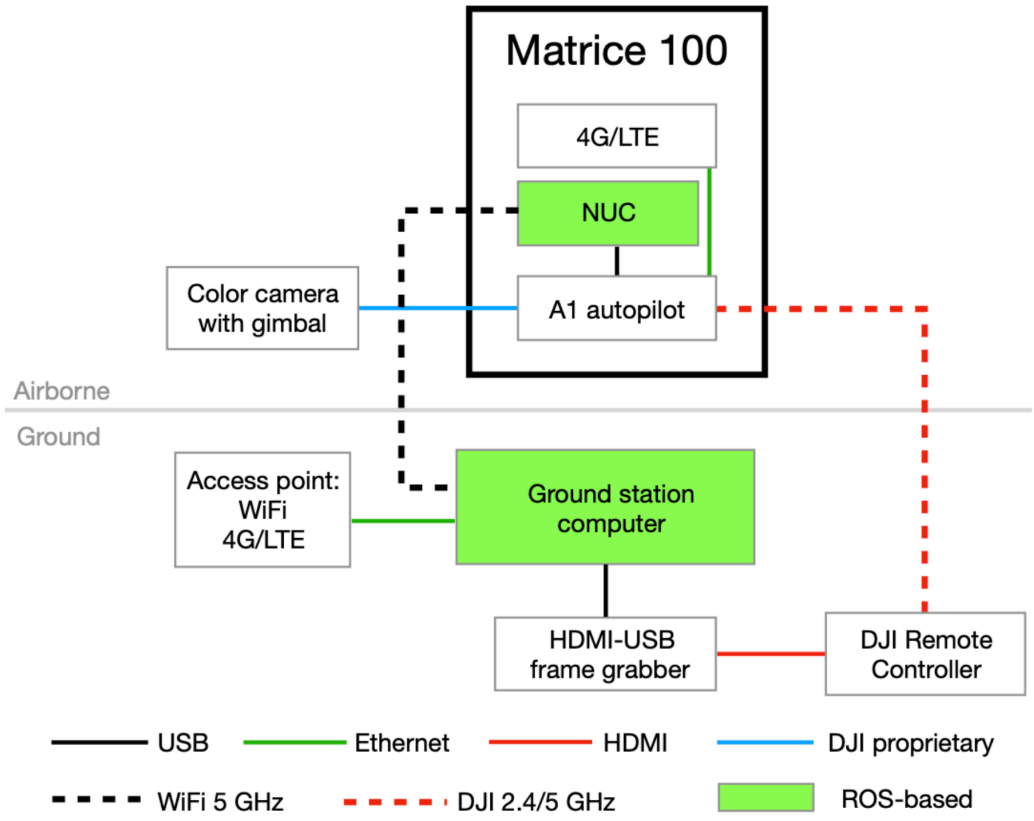


*Figure 7.7: Schematic of the airborne and ground hardware components and their interconnections including wireless links. Additionally, one of the main open source software for our project is the DJI Software Development Kits (SDK) in green. The SDKs are designed to allow "registered developers" to 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[1]*

7.2.4 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".

[15] `https://gitlab.liu.se/lrs/lrs_devenv_common`

Chapter 8

Summary, Conclusions and Future Works

This thesis work presented two different problems in the engineering fields of Automatic Control and AI Robotics. In the following, we summarize the work regarding each part. Then, we review the benefits of Robotic Industrial Inspections studied by Boston Dynamics.

8.1 Rendezvous Landing Problem

In the first main task, a rendezvous landing problem, a hierarchical control approach (cascaded structure) using PID and MPC scheme is studied for performing the rendezvous landing maneuver of UAV quadrotor DJI-M100 atop the landing platform on the quadruped Spot robot. In summary,

- The *theoretical analysis* of this problem is carried out and presented. MPC is considered a suitable control strategy for this type of problems for ensuring a soft and safe landing maneuver.

- Possible Extensions: The presented controller of the `rendezvous_control` node is a PID controller. A non-linear MPC controller using ACADO tools, adapted to the work presented in [Kamel, 2017] and discussed in this thesis, might be an interesting extension to improve the rendezvous landing performance.

- The control objective is to design applied thrust T and/or torques τ for the UAV such that the UAV tracks and lands on the desired position on the ground mobile robot Spot.

- To be able to perform a soft landing, the motion of the Q-UGV Spot can be tracked using RTK-GNSS system, with which the quadrotor may acquire information capable of identifying the *relative pose* of the quadruped robot Spot with respect to its own position. This information allows the implemented control system to plan a trajectory leading the quadrotor in an *appropriate landing configuration.*

- SIMULATIONS OF THE RENDEZVOUS LANDING MANEUVER NEED WAS PERFORMED AND TESTED USING THE LRS DEVELOPMENT ENVIRONMENT AND THE WARA-PS SIMULATOR TOGETHER WITH PID CONTROLLER, AFTER REPRESENTING SPOT IN A Gazebo-ROS SIMULATION MODEL. NEXT, THE PERFORMANCE OF THIS MANEUVER WAS EVALUATED THROUGH SIMULATED EXPERIMENTS BY OBTAINING USEFUL RELATED PLOTS showing the convergence of the relative distance towards zero.

- Simulated Experiments: Using the simulator for the DJI M100 in WARA-PS framework could actually help in the development and testing of the Rendezvous task by performing simulated experiments. IN FUTURE WORK, A PRACTICAL/EXPERIMENTAL EVALUATION ON THE REAL PLATFORMS CAN BE CARRIED OUT AS WELL, USING DJI AND SPOT.

8.2 AI Task Planning Problem

In the second task, we need to solve an AI planning problem for the automated robotic inspections (ARI) domain in construction/industrial environment. In summary,

- A PDDL-model is implemented to generate a mission plan to achieve a set of visual inspection goals (e.g., pictures or camera pointing for video live-streaming) for this robotic inspection domain. The goal is to navigate an industrial/construction site and performing visual inspection on predefined inspection points.
- 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 each action into low-level functional commands.
- IN FUTURE WORK, THIS PLAN CAN BE EXECUTED AND TESTED ON THE ROBOTIC PLATFORMS, SPOT AND DJI, IN CERTAIN INSPECTION SCENARIO.

8.3 Benefits of Robotic Industrial Inspections

With workforce shortages and employee safety a top concern, industrial teams are looking to robotic automation solutions to improve operations. A mobile IoT sensor platform, Spot can perform remote or automated tasks like thermal inspection, gauge reading, acoustic imaging, and more - collecting the critical data needed to power your predictive maintenance program.

Automating Inspections with Agile Mobile Robots according to *Boston Dynamics* studies can give the following benefits:

- *Improve Safety*: Remove personnel from dangerous, electrified, radioactive, explosive environments. Collect data remotely and identify risks from a far.
- *Improve Efficiency*: Reduce time to inspection. Improve uptime. Redistribute labor to critical decision making
- *Improve Predictability*: 100X data collection. Reduced fixed sensors. Visual inspections. Analog gauges. Gas leaks. Thermal anomalies. Acoustic anomalies. Corrosion detection.

Chapter 9

Appendix

9.1 Domain File - Robotic Inspection Task

Listing 9.1: The domain file for the robotic inspection problem which describes general aspects of the 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
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 )
```

9.2 Problem File - Robotic Inspection Task

Listing 9.2: A small problem instance file of the robotic inspection problem which contains description of the particular aspects of the problem to solve

```
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 )
```

9.3 Implementation of nonlinear MPC using ACADO

ACADO provides an example on how to formulate a nonlinear OCP in C++. The tutorial in ACADO Toolkit [1] 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.

9.3.1 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:

$$\begin{aligned}
\min_{x(\cdot),u(\cdot),p} \quad & \int_{t_0}^{t_0+T} \|h(t,x(t),u(t),p)-\eta(t)\|_Q^2 \,\mathrm{d}t + \|m(x(t_0+T),p,t_0+T)-\mu\|_P^2 \\
\text{subject to:} \quad & x(t_0) = x_0 \\
\forall t \in [t_0, t_0+T]: \quad & 0 = f(t,x(t),\dot{x}(t),u(t),p) \\
\forall t \in [t_0, t_0+T]: \quad & 0 \geq s(t,x(t),u(t),p) \\
& 0 = r(x(t_0+T),p,t_0+T)
\end{aligned}$$

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.

9.3.2 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.)

[1] https://acado.sourceforge.net/doc/html/d4/d26/example_013.html

Listing 9.3: The following piece of code shows how to implement an MPC controller based on this 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:
```

```
    //
    const double tStart = 0.0;
    const double tEnd   = 1.0;

    OCP ocp( tStart, tEnd, 20 );

    ocp.minimizeLSQ( Q, h, r );

    ocp.subjectTo( f );

    ocp.subjectTo( -200.0 <= F <= 200.0 );
    ocp.subjectTo( R == 0.0 );


    // SETTING UP THE REAL-TIME ALGORITHM:
    // ------------------------------------
    RealTimeAlgorithm alg( ocp,0.025 );
    alg.set( MAX_NUM_ITERATIONS, 1 );
    alg.set( PLOT_RESOLUTION, MEDIUM );

    GnuplotWindow window;
        window.addSubplot( xB, "Body Position [m]" );
        window.addSubplot( xW, "Wheel Position [m]" );
        window.addSubplot( vB, "Body Velocity [m/s]" );
        window.addSubplot( vW, "Wheel Velocity [m/s]" );
        window.addSubplot( F,  "Damping Force [N]" );
        window.addSubplot( R,  "Road Excitation [m]" );

    alg << window;


    // SETUP CONTROLLER AND PERFORM A STEP:
    // ------------------------------------
    StaticReferenceTrajectory zeroReference( "ref.txt" );

    Controller controller( alg,zeroReference );

    Vector y( 4 );
    y.setZero( );
    y(0) = 0.01;

    controller.init( 0.0,y );
    controller.step( 0.0,y );


    return 0;
}
```

9.4 Implementation of OCP in Python using CVXPY

CVXPY provides an example on how to formulate an OCP in Python [2]. **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.,

$$x_{t+1} = Ax_t + Bu_t \tag{9.1}$$

[2] https://nbviewer.org/github/cvxgrp/cvx_short_course/blob/master/intro/control.ipynb

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**

$$\min \quad \sum_{t_0}^{T-1} l(x_t, u_t) + l_T(x_T) \tag{9.2}$$

$$\text{subject to:} \quad x_{t+1} = Ax_t + Bu_t \tag{9.3}$$

$$(x_t, u_t) \in \mathcal{C},\ x_T \in \mathcal{C}_T \tag{9.4}$$

where $l : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is the stage cost, l_T is the terminal cost, $\mathcal{C}$ is the state/action constraints, and $\mathcal{C}_T$ is the terminal constraint. The optimization problem is convex if the costs and constraints are convex.

9.4.1 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: The following piece of code shows how to implement an MPC controller where we solve a control problem with 8 states 2 inputs and horizon T

```
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: Code for displaying the resulting output as a 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])
39 plt.ylim([-25, 25])
40 plt.ylabel(r"$(x_t)_2$", fontsize=16)
41 plt.xlabel(r"$t$", fontsize=16)
42 plt.tight_layout()
43 plt.show()
```

Bibliography

[1] WARA-PS Core System Portal (LiU/AIICS), Wallenberg AI, Autonomous Systems and Software Program

[2] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3rd Edition, Prentice Hall Series in Artificial Intelligence, 2010. Available: http://aima.cs.berkeley.edu, June 2021. [Online, visited 22-June-2021]

[3] David L. Poole, Alan K. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*, Cambridge University Press, 2nd Edition, 2017

[4] Robin R. Murphy, *Introduction to AI Robotics*, 2nd Edition, MIT Press, 2019

[5] Carol Fairchild and Thomas Harman, *ROS Robotics by Example*. Packt Publishing, 2nd edition 2017

[6] Francesco Borrelli, Alberto Bemporad, and Manfred Morari, *Predictive control for linear and hybrid systems*, First Edition, Cambridge University Press 2017. [Online, visited 22-June-2021]. Available: http://www.mpc.berkeley.edu/mpc-course-material.

[7] Lars Grune, Jurgen Pannek, *Nonlinear Model Predictive Control*, Communications and Control Engineering, Springer-Verlag London Limited 2011

[8] Torkel Glad and Lennart Ljung, *Control Theory: Multivariable and Nonlinear Methods*. Taylor & Francis 2000

[9] Karl J. Åström, Richard M. Murray, Feedback Systems: An Introduction for Scientists and Engineers, Second Edition, Princeton University Press 2020

[10] Richard M. Murray, *Optimal-Based Control*. Control and Dynamical Systems, California Institute of Technology (Caltech) 2022

[11] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo, Robotics — Modelling, Planning and Control, Springer Advanced Textbooks in Control and Signal Processing Series, London, UK, 2009. Italian edition: Robotica — Modellistica, Pianificazione e Controllo, McGraw-Hill Libri Italia, Milano, 2008

[12] Lorenzo Marconi, Alberto Isidori, and Andrea Serrani. *Autonomous vertical landing on an oscillating platform: an internal-model based approach*. Automatica, 2002.

[13] Samir Bouabdallah, Roland Siegwart, "full Control of a Quadrotor", *International Conference on Intelligent Robots and Systems*, San Diego, CA, USA, Oct 29 - Nov 2, 2007

[14] Daniel Mellinger, Michael Shomin, and Vijay Kumar, "Control of Quadrotors for Robust Perching and Landing". *International Powered Lift Conference*, October 5-7, 2010, Philadelphia, PA.

[15] D. Mellinger, N. Michael, and V. Kumar, "Trajectory Generation and Control for Precise Aggressive Maneuvers with Quadrotors," Int. J. Robotics. Research., vol. 31, no. 5, pp. 664–674, Apr. 2012.

[16] R. Mahony, V. Kumar, and P. Corke, " Multirotor aerial vehicles: Modeling, estimation, and control of quadrotor ", *IEEE Robotics Automation Magazine*, vol. 19, no. 3, pp. 20–32, Sept 2012.

[17] Michael Bloesch, Stephan Weiss, Davide Scaramuzza, and Roland Siegwart, "Vision based MAV navigation in unknown and unstructured environments", in *International Conference on Robotics and Automation (ICRA)*, 2010

[18] Mina Kamel, Thomas Stastny, Kostas Alexis and Roland Siegwart, "Model Predictive Control for Trajectory Tracking of Unmanned Aerial Vehicles Using Robot Operating System", in *Robot Operating System (ROS) The Complete Reference* (A. Koubaa, ed.), Springer 2017

[19] Mina Kamel, Michael Burri and Roland Siegwar, "Linear vs Nonlinear MPC for Trajectory Tracking Applied to Rotary Wing Micro Aerial Vehicles", *ArXiv Robotics* http://arxiv.org/abs/1611.09240v2, 2017 (version 2)

[20] Inkyu Sa, Mina Kamel, R. Khanna, M. Popovic, J. Nieto, and Roland Siegwart, "Dynamic System Identification, and Control for a cost-effective and open-source Multi-rotor MAV", in *Field of Service Robotics*, 2017.

[21] Fadri Furrer, Michael Burri, Markus Achtelik and Roland Siegwart. "RotorS — A Modular Gazebo MAV Simulator Framework". In: *Koubaa, A. (eds) Robot Operating System (ROS). Studies in Computational Intelligence, vol 625. Springer*, Cham. https://doi.org/10.1007/978-3-319-26054-9_23

[22] Kamel, M., Alonso Mora, J., Siegwart, R., Nieto, J. (2017). "Robust collision avoidance for multiple micro aerial vehicles using nonlinear model predictive control". In A. Bicchi, T. Maciejewski (Eds.), *Proceedings 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 236-243). IEEE. https://doi.org/10.1109/IROS.2017.8202163

[23] Samir Bouabdallah, *Design and control of quadrotors with application to autonomous flying*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2007.

[24] Daewon Lee, Tyler Ryan, and H. Jin Kim, "Autonomous landing of a VTOL UAV on a moving platform using image-based visual servoing," in *Robotics and Automation (ICRA), 2012 IEEE International Conference* on, pp. 971–976, IEEE, 2012.

[25] Caitlin Powers, Daniel Mellinger, and Vijay Kumar. *Quadrotor Kinematics and Dynamics*. In K. Valavanis and G. Vachtsevanos, Handbook of Unmanned Aerial Vehicles, pages 307–328. Springer Netherlands, Dordrecht, 2015.

[26] Castillo P., Dzul A. and Lozano R., "Real-time stabilization and tracking of a four rotor mini rotorcraft", *IEEE Transactions on Control Systems Technology*, Vol. 12, No. 4, pp. 510–516, July 2004.

[27] Pedro Castillo, Rogelio Lozano, Alejandro Dzul, "Modelling and Control of Mini-Flying Machines", Springer 2005

[28] Edward. N. Hartley, Marco Gallieri, and Jan M. Maciejowski, (2013). "Terminal Spacecraft Rendezvous and Capture with LASSO Model Predictive Control", *International Journal of Control, vol. 86, no. 11, pp. 2104–2113*, 2013. https://doi.org/10.1080/00207179.2013.789608

[29] Edward. N. Hartley, *A tutorial on model predictive control for spacecraft rendezvous*, in European Control Conference (ECC), July 2015.

[30] Tommaso. Bresciani, *Modelling, Identification and Control of a Quadrotor Helicopter*, Master's Thesis, Lund University, Sweden, 2008.

[31] Aman Sharma, "System Identification of a Micro Aerial Vehicle", Master's Thesis, Luleå University of Technology, Sweden 2019.

[32] Karl Engelbert Wenzel, Andreas Masselli, and Andreas Zell. *Automatic take off, tracking and landing of a miniature uav on a moving carrier vehicle*. Journal of Intelligent & Robotic Systems, 61(1):221–238, Jan 2011

[33] Jason Kong, Mark Pfeiffer, Georg Schildbach, Francesco Borrelli, *Kinematic and Dynamic Vehicle Models for Autonomous Driving Control Design*, IEEE Intelligent Vehicles Symposium (IV) 2015

[34] A. Rucco, P. Sujit, A. P. Aguiar, and J. Sousa, *Optimal UAV rendezvous on a UGV*, AIAA Guidance, Navigation, and Control Conference, 2016.

[35] S. Boyd, and L. Vandenberghe. *Convex Optimization*. Cambridge: Cambridge University Press, 2004.

[36] Jacob Mattingley, Yang Wang, Stephen Boyd. (2011). "Receding Horizon Control: Automatic Generation of High-Speed Solvers". *IEEE Control Systems Magazine*, June 2011. Digital Object Identifier 10.1109/MCS.2011.940571

[37] Boris Houska, Hans J. Ferreau and Moritz Diehl, (2011). "ACADO toolkit - An Open-Source Framework for Automatic Control and Dynamic Optimization". *Optimal Control Applications and Methods, 2011; 32:298–312*. DOI: 10.1002/oca.939

[38] Bellman, R., *Dynamic Programming*. Princeton University Press, Princeton (1957). Reprinted in 2010

[39] Pontryagin, L.S., Boltyanskii, V.G., Gamkrelidze, R.V., Mishchenko, E.F., The Mathematical Theory of Optimal Processes. Translated by D.E. Brown. Pergamon/Macmillan Co., New York (1964)

[40] Linnea Persson, Tin Muskardin, Bo Wahlberg, *Cooperative Rendezvous of Ground Vehicle and Aerial Vehicle using Model Predictive Control*. In 2017 IEEE 56th Annual Conference on Decision and Control (CDC), pages 2819–2824, Dec 2017.

[41] Linnea Persson and Bo Wahlberg. *Model predictive control for autonomous ship landing in a search and rescue scenario*. In AIAA SciTech 2019 Forum, 2019.

[42] Linnea Persson, *Autonomous and Cooperative Landings Using Model Predictive Control*, Licentiate Thesis, KTH Stockholm 2019

[43] Linnea Persson, *Model Predictive Control for Cooperative Rendezvous of Autonomous Unmanned Vehicles*, PhD Thesis, KTH Stockholm 2021

[44] Wolfgang Hoenig and Nora Ayanian, *Flying Multiple UAVs Using ROS*. University of Southern California, Los Angeles, CA, USA. Springer ROS 2017

[45] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*, Morgan & Claypool, 6/2013.

[46] Patrick Doherty, Fredrik Heintz, Jonas Kvarnström, *High-Level Mission Specification and Planning for Collaborative Unmanned Aircraft Systems using Delegation*, Unmanned Systems, Vol. 0, No. 0 (2013) 1–39: https://www.ida.liu.se/divisions/aiics/publications/US-2013-High-Level-Mission.pdf.

[47] Hector Geffner et al., *Introduction to Planning Models and Methods*, IJCAI Tutorial 2016

[48] Malik Ghallab et al., *Deliberative Planning and Acting*, IJCAI Tutorial 2016 http://projects.laas.fr/planning/ijcai-2016-tutorial/index.html, June 2021. [Online, visited 22-June-2021]

[49] Moritz Tenorth, Michael Beetz, "Representations for robot knowledge in the KnowRob framework", *Artificial Intelligence*, 2015. https://doi.org/10.1016/j.artint.2015.05.010

[50] Roland Brachman, Hector Levesque, *Knowledge Representation and Reasoning*, 1st Edition, Morgan Kaufmann, 2004

[51] P. Ulam, Y. Endo, A. Wagner, R. C. Arkin, "Integrated Mission Specification and Task Allocation for Robot Teams – Design and Implementation", in ICRA 2007

[52] Malte Helmert, "The Fast Downward Planning System". *Journal of Artificial Intelligence Research, 26:191–246*, 2006. https:/doi.org/10.1613/jair.1705

[53] Silvia Richter and Matthias Westphal, "The LAMA planner: Guiding cost-based anytime planning with landmarks". *Journal of Artificial Intelligence Research, 39:127–177, 2010*. https://doi.org/10.1613/jair.2972

[54] Joerg Hoffmann. "The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables". *Journal of Artificial Intelligence Research, 20:291–341*, 2003. https://doi.org/10.1613/jair.1144

[55] Coles, A., Coles, A., Fox, M., Long, D. (2010). "Forward-Chaining Partial-Order Planning". *Proceedings of the International Conference on Automated Planning and Scheduling, 20(1), 42-49*. Retrieved from https://ojs.aaai.org/index.php/ICAPS/article/view/13403

[56] Alfonso Gerevini, Ivan Serina. "LPG: A Planner Based on Local Search for Planning Graphs with Action Costs". American Association for Artificial Intelligence, 2002

[57] J. Benton, A. Coles, and A. Coles. (2012). "Temporal planning with preferences and time-dependent continuous costs", *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, 01 2012. Retrieved from https://ojs.aaai.org/index.php/ICAPS/article/view/13509

[58] Nils J. Nilsson, *Shakey the robot*, Technical report 323, AI Center, SRI International, Menlo Park, CA, USA, 1984. http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/shakey-the-robot.pdf

[59] Raymond Reiter, *Knowledge in action : logical foundations for specifying and implementing dynamical systems*, MIT Press, 2001, http://www.cs.toronto.edu/cogrobo/kia/

[60] Steven M. LaValle *Planning Algorithms*. Cambridge University Press, 2006

[61] Ghallab M., Nau D., Traverso P., *Automated Planning: Theory and Practise*. Morgan Kaufmann Publishers, San Fransisco, CA (2004)

[62] Ghallab M., Nau D., Traverso P., *Automated Planning and Acting*. Cambridge University Press, 2016

[63] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela M. Veloso, Daniel Weld, and David Wilkins. "PDDL - The Planning Domain Definition Language". *AIPS-98 Planning Committee*, 1998

[64] Allen, J., Hendler, J., and Tate, A. (Eds). 1990. Readings in Planning. Morgan Kaufmann, San Mateo, CA.

[65] Fikes, R. E. and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 1971

[66] Pednault, E. P. D. (1986). "Formulating Multiagent, Dynamic-World Problems in the Classical Planning Framework". *In Reasoning about Actions and Plans*: Proc. 1986 Workshop, pp. 47-82.

[67] Edwin Pednault, "Synthesizing Plans that Contain Actions with Context-Dependent Effects", *Computational Intelligence 4, 356-372*, 1988 Knowledge Systems Research Department, AT&T Bell Laboratories, USA.

[68] Felix Ingrand, Malik Ghallab, "Deliberation for Autonomous Robots: A Survey". LAAS-CNRS, University of Toulouse, France, 2014 Elsevier: https://doi.org/10.1016/j.artint.2014.11.003

[69] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, Christian Muise, *An Introduction to the Planning Domain Definition Language*, Morgan and Claypool, 2019

[70] Francesco Riccio, *Spatial Representation for Planning and Executing Robot Behaviors in Complex Environments*. PhD Thesis, Sapienza University of Rome 2018

[71] Lukas Steenstra, *SPDDL-Based Task Planning of Survey Missions for Autonomous Underwater Vehicles*. MSc Thesis, Delft University of Technology, 2019

[72] Patrick Ferber and Jendrik Seipp. 2022. "Explainable Planner Selection for Classical Planning", *Proceedings of the 36th AAAI Conference on Artificial Intelligence, Association for the Advancement of Artificial Intelligence.*

[73] Michael Katz, Shirin Sohrabi, Horst Samulowitz, Silvan Sievers. 2018. "Delfi: Online Planner Selection for Cost-Optimal Planning". In *IPC-9 Planner Abstracts, 57–64*

[74] Patrick Doherty, *TDDC17 Artificial Intelligence*, Lab4 - Planning: https://www.ida.liu.se/~TDDC17/info/labs/lab4/lab4_planning.en.shtml

[75] Jonas Kvarnstrom. "Planning for Loosely Coupled Agents Using Partial Order Forward-Chaining", *In Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS). AAAI, 138–145*, 2011

[76] Alejandro Torreño, Eva Onaindia, Antonin Komenda, Michal Stolba, "Cooperative Multi-Agent Planning: A Survey", *ACM Computing Surveys, Volume 50, Number 6, Article 84*, 2017 https://doi.org/10.1145/3128584

[77] P. Muõz, M. R-Moreno, and D. Barrero, "Unified Framework for Path-Planning and Task-Planning for Autonomous Robots", *Robotics and Autonomous Systems, vol. 82, pp. 1–14*, 2016.

[78] Antonin Komenda, Michal Stolba, and Daniel L Kovacs. 2016. "The International Competition of Distributed and Multiagent Planners (CoDMAP)". *AI Magazine 37, 3 (2016), 109–115*, 2016

[79] Daniel L. Kovacs, "A Multi-Agent Extension of PDDL3.1", *In Proceedings of the 3rd Workshop on the International Planning Competition (IPC). 19–27*, 2012

[80] Mikkel Rath Pedersen, Lazaros Nalpantidis, Rasmus Skovgaard Andersen, Casper Schou, Simon Boegh, Volker Krueger, Ole Madsen, "Robot skills for manufacturing: From concept to industrial deployment". *Robotics and Computer-Integrated Manufacturing*, Elsevier 2015

[81] Y. Jiang, S. Zhang, P. Khandelwal, P. Stone, *Task Planning in Robotics: an Empirical Comparison of PDDL-based and ASP-based Systems*: https://arxiv.org/abs/1804.08229. Frontiers of Information Technology & Electronic Engineering 2019

[82] C. Castelfranchi and R. Falcone, *Towards a theory of delegation for agent-based systems*, Robotics and Autonomous Systems, 24 (1998), pp. 141–157.

[83] Jonas Kvarnström, *Automated Planning*, WASP Lecture 2016, Department of Computer and Information Science, Linköping University

[84] Alberto Finzi, Intelligent Robotics (IROB), Lecture 15 *Task Planning*, University of Napoli 2021 http://wpage.unina.it/alberto.finzi/didattica/IROB/materialeIROB2021.html

[85] Alberto Finzi, Sistema per il Governo dei Robot (SGRB), Lecture 15 *Task Planning*, University of Napoli 2020 http://wpage.unina.it/alberto.finzi/didattica/SGRB/materialeSGRB1920.html

[86] H. Ahmadzadeh, E. Masehian, "Modular robotic systems: Methods and algorithms for abstraction, planning, control, and synchronization", Elsevier, 2015

[87] Dana Nau, Lecture slides, *Automated Planning: Theory and Practice*, University of Maryland 2013 http://www.cs.umd.edu/~nau/planning/slides/

[88] Anis Koubaa, *Robot Operating System (ROS) The Complete Reference*, Volume 1, Springer, 2016

[89] Anis Koubaa, *Robot Operating System (ROS) The Complete Reference*, Volume 2, Springer, 2017

[90] Chiara Piacentini, Varvara Alimisis, Maria Fox, and Derek Long. *An extension of metric temporal planning with application to AC voltage control*. Artificial Intelligence, 229:210–245, 2015. DOI:10.1016/j.artint.2015.08.0106

[91] Chiara Piacentini, Daniele Magazzeni, Derek Long, Maria Fox, and Chris Dent. *Solving realistic unit commitment problems using temporal planning: Challenges and solutions*. In Amanda Jane Coles, Andrew Coles, Stefan Edelkamp, Daniele Magazzeni, and Scott Sanner, Eds., Proc. of the 26th International Conference on Automated Planning and Scheduling (ICAPS), pages 421–430, AAAI Press, London, UK, June 12–17, 2016. http://www.aaai.org/ocs/index.php/ICAPS/ICAPS16/paper/view/129926

[92] Reza Olfati-Saber, J. A. Fax, and Richard M. Murray, *Consensus and cooperation in networked multi-agent systems*, Proceedings of the IEEE, vol. 95, Jan 2007.

[93] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in ICRA workshop on open source software, vol. 3, p. 5, Kobe, Japan, 2009

[94] BoonPing Lim, Menkes van den Briel, Sylvie Thiébaux, Scott Backhaus, and Russell Bent. *HVAC-aware occupancy scheduling*. In Proc. of the 29th AAAI Conference on Artificial Intelligence, pages 679–686, Austin, TX, January 25–30, 2015. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/94066

[95] ROS for Spot Robot: http://wiki.ros.org/Robots/Spot, June 2021. [Online, visited 22-June-2021]

[96] Clearpath ROS Package for Spot Robot: https://clearpathrobotics.com/spot-robot/, June 2021. [Online, visited 22-June-2021]

[97] Spot ROS User Documentation: http://www.clearpathrobotics.com/assets/guides/melodic/spot-ros/, June 2021. [Online, visited 22-June-2021]