Report SFE: Manipulating a cable-suspended object with multiple UAVs and environment contacts in 2D

Author:
M. Leonardo Mouta Pereira Pinheiro

Advisors:
Pr. Yves Brière (ISAE)
Dr. Simon Lacroix (LAAS)

November 25, 2022

Acknowledgments

First and foremost, I'd like to thank God for all the opportunities given to me so far.

I'd also like to thank ISAE-SUPAERO for the education I have been given these past two years. It has been a fantastic experience to participate in your double degree program. I would like to specially thank Dr. Yves Brière, who was my advisor for this report and whose classes in ECAS-210 started my interest in UAVs, and the Autonomous Systems domaine and Fluid Dynamics filière faculty as a whole.

LAAS is also to thank for this amazing internship, where I was wonderfully advised by Dr. Simon Lacroix and Dr. Hai-Nguyen (Hann) Nguyen. Also co-advising this work was Dr. Christopher Shneider Cerqueira from ITA-Brazil. I'd also like to thank Dr. Dario Sanalitro, Dr. Martin Jacquet and PhD student Gianluca Corsini, who were very friendly desk neighbors and who greatly contributed in providing ideas for this work, and my co-interns Colomban Le Falher and Arthur Lotz, who were great to work with.

Finally, thanks is also due to my family and to some special colleagues: my girlfriend Ana Santos, without whom I'd probably have flunked some courses, Mariana Alves, my steadfast binôme for all the BEs, and all my friends from the ISAE Brazillian community.

Contents

List of Figures

List of acronyms

CNRS	*National Center for Scientific Researc*
LAAS	*Laboratory of Analysis and Architecture of Systems*
RIS	*Robotics and Interactions*
SE(2)	*Special Euclidean group of order 2*
SO(2)	*Special orthogonal group of order 2*
UAV	*Unmanned Aerial Vehicule*

Chapter 1

Introduction

The objective of the report is to detail the activities performed during my end of studies internship, as required by ISAE-SUPAERO. Said internship took place between April 2022 and September 2022 at the Laboratory of Analysis and Architecture of Systems (LAAS), in Toulouse, and was focused on the domain of robotics.

The mission of this internship was to study the manipulation of cable-suspended objects, using both a fleet of unmanned aerial vehicles (UAV) and interactions with the environments, in order to arrive at the desired poses. This approach relies on methods developed previously for finger manipulators. After an initial formulation of the mathematics of the problem and of its solution algorithm, this solution was then studied in computer simulated environments so that its quality could be assessed.

As for the report's structure, I will first describe the role and the history of LAAS, followed by the specific role of the team of which I was part. Then a description of the mathematical and computational formulations of the problem will be presented, followed by a chapter discussing the results obtained.

Chapter 2

Context

2.1 LAAS-CNRS: A Research Lab

The Laboratory of Analysis and Architecture of Systems (LAAS) is a public research lab which was founded in 1968 as a part of the National Center for Scientific Research (CNRS). In its inception, founder Jean Lagasse dedicated the lab to the study of automation, back then a newborn field, but this notion was surpassed when the growing complexity of tasks led to the rise of the concept of "systems". It was soon realized that automation was but a part of a much larger context. Nowadays, the lab's mission consists of modeling, designing and controlling various sorts of complex systems, from microelectronics to human-robot interactions. More precisely, the research developed at LAAS is split into six main areas: networks/IT, robotics, decision and optimization, microwaves, energy management, and "MicroNanoBio" Technologies, with each area being comprised of several different teams. All the research done at LAAS is pointed towards five goal lines, which are shown in Fig. 2.1 [LAAS 2022].

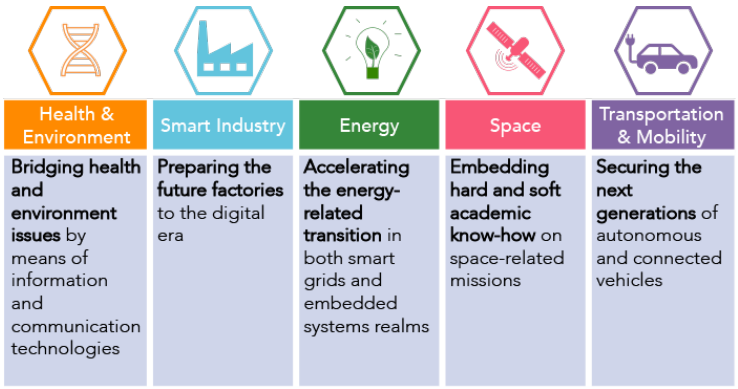


Figure 2.1: Main applications areas of the LAAS's research. Source: [LAAS 2022]

As for the structure of the teams mentioned, the organizational chart of the lab can be seen in Fig. 2.2. The activities of this internship took place in the midst of the RIS team, which is a part of the robotics research area.

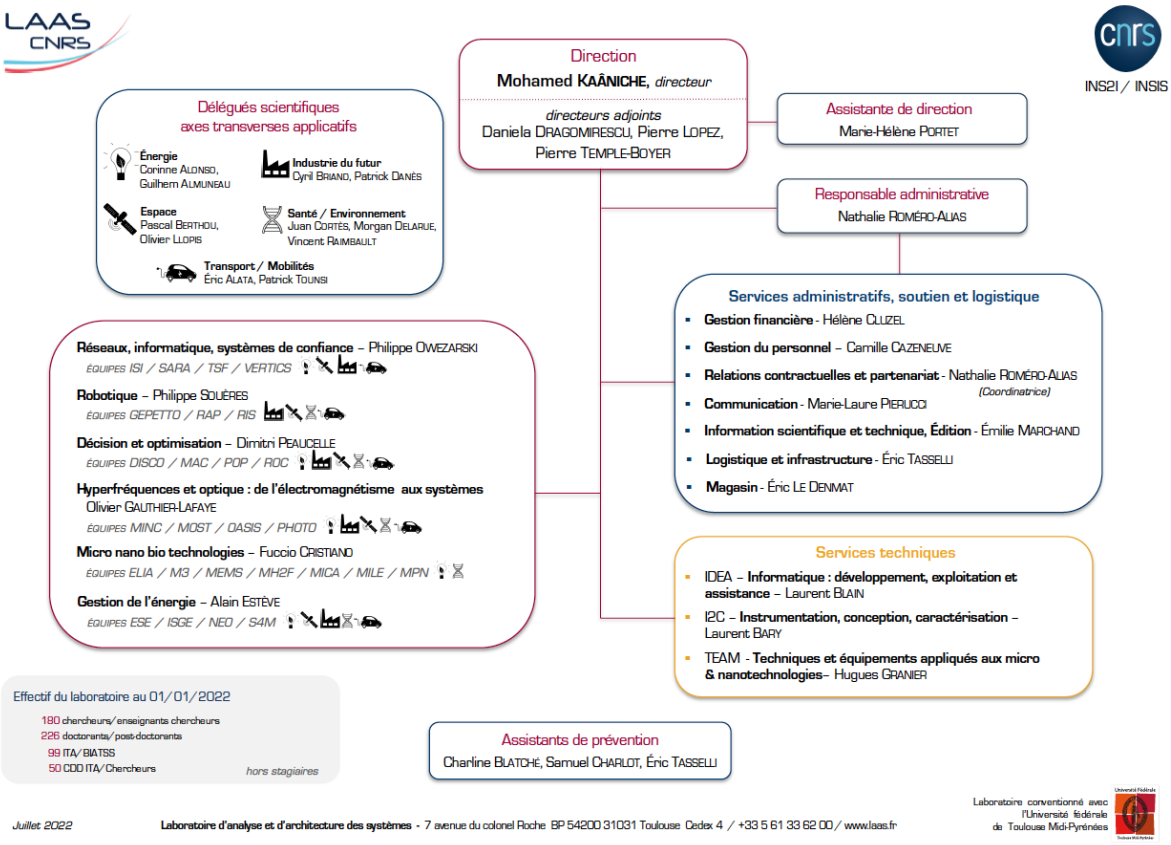


Figure 2.2: Organizational chart of the LAAS. Source: [LAAS 2022]

2.2 Team Robotics and Interactions (RIS)

The acronym RIS stands for **R**obotics and **I**nteraction**S**, which summarizes well the scope of the team: to design autonomous machines that combine and integrate "perception, reasoning, learning, and action/reaction capabilities" [LAAS 2022]. With the growing complexity of the tasks performed by robots, the need to account for more refined interactions between the platform and its environment makes the work developed by the RIS team both fundamental and innovative. Moreover, it is not only the environment that falls under the category of "interactions": cooperation between robots, human operators and interfaces with existing systems are all examples of interactions.

The philosophy employed by the RIS team is as follows: first, a set of tools is developed.

These tools include architectures, controls, planners, etc. Then, these tools are used in the context of three context areas: multi-robot systems, cognitive and interactive robots, and molecular motion problems (which can be modeled as robotics problems) [LAAS 2022]. This structure is shown in Fig. 2.3.

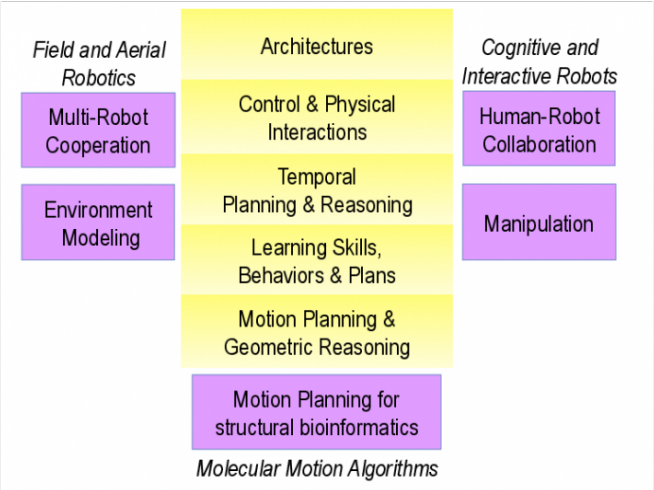


Figure 2.3: Areas and tools of the RIS team. Source [LAAS 2022]

2.3 UAV Research Internship at RIS

Autonomous aerial vehicles have become more and more common in our society, and the RIS team is working to ensure that UAVs are up to the tasks demanded of them. Over the course of the years, the paradigm of UAV usage has shifted: initially, UAVs were employed "in isolation", meaning they avoided interacting with the environment and usually worked alone. One example of such mission is aerial filming. However, current missions can require cooperation, environment interaction, and other complex tasks. Consider for example a scenario where a fleet of drones is used to do welding. Not only do the drones have to cooperate among themselves, but a high degree of "environment reading" is required. This evolution of the usage of UAVs can be summarized in Fig. 2.4 [Ollero 2021].

	First Generation	Second Generation	Third Generation
Environment	Indoor	Indoor/Outdoor	Indoor/Outdoor
Platforms	Quadrotors	Helicopter Multirotors	Multirotors fully actuated Multibody
Robotic Arms	Few DoFs	6/7 DoFs Compliance	Dual arms Hyper-redundant
Navigation	Motion tracking system	Indoor beacons GPS	SLAM
Perception	No motion tracking onboard	Onboard with markers	Onboard without markers
Planning	No planning	Ground station Basic reaction	On-board Reactivity

Figure 2.4: Evolution of UAVs. Courtesy of Antonio Franchi @ LAAS

Among the new tasks required of UAVs, we can cite the example of object transportation/manipulation, which has become ever more common with the advent of drone delivery services. Needless to say, this type of mission spans numerous applications, including placing objects in places which are dangerous or out of reach for humans, manipulating heavy objects, dealing with objects which require high precision, etc. For many applications, however, this type of mission requires more than one drone, be it because of the object's weight, size, geometry, etc. Coordinating the movement of all the drones manipulating the object in order to make it arrive at the desired pose is not a trivial matter, but this mission can be divided into three main problem subsets, such as shown in Fig. 2.5: a path planning problem on the drones' positions, a manipulation problem on the object's pose, and a control problem on the system drones-cables-object.

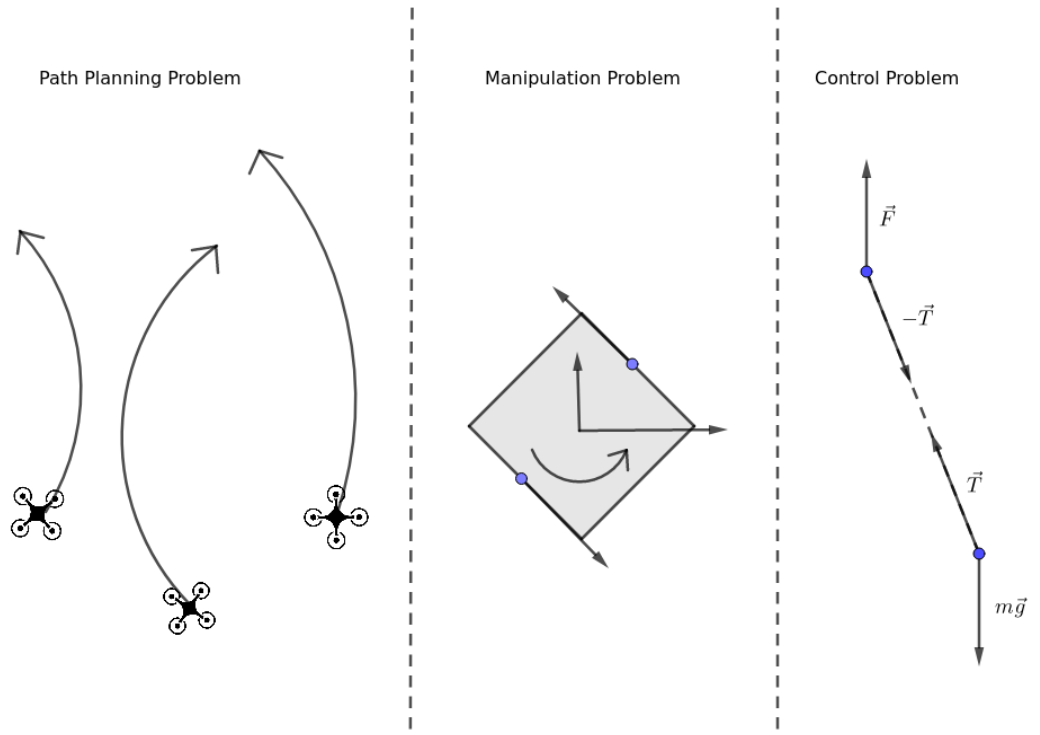


Figure 2.5: Decomposition of the UAV manipulation problem

Several difficulties arise when dealing with these scenarios. First of all, let us consider the problem of dealing with the control of the drones, which are now subject to varying loads. This problem has been tackled in the works of [Sreenath 2013], for the case of a single UAV, and of [Sanalitro 2020], for the case with multiple UAVs. As for the the problem of generating the drones' trajectories, it is clear that these trajectories are not known *a priori* when considering the initial and goal poses of the object. When generating the trajectories, part of the difficulty lies in the fact that there are many discontinuities involved in the drones'/object's path. In Fig. 2.6, one can see examples of such discontinuities: when the object touches the surface, when the cable attaching a drone to the object is relaxed, etc. These discontinuities and environmental interactions are not necessarily undesirable. As an example, Fig. 2.7 shows a case where a drone could use the friction between the object and the surface in order to change the object's orientation.

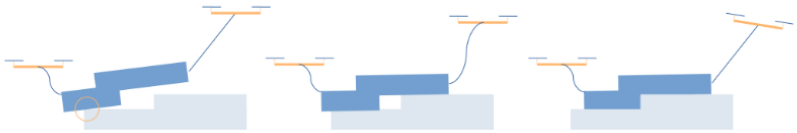


Figure 2.6: Sometimes multiple drones have to coordinate their movement and account for environment interactions in order to successfully manipulate an object. Courtesy of Hann Nguyen @ LAAS

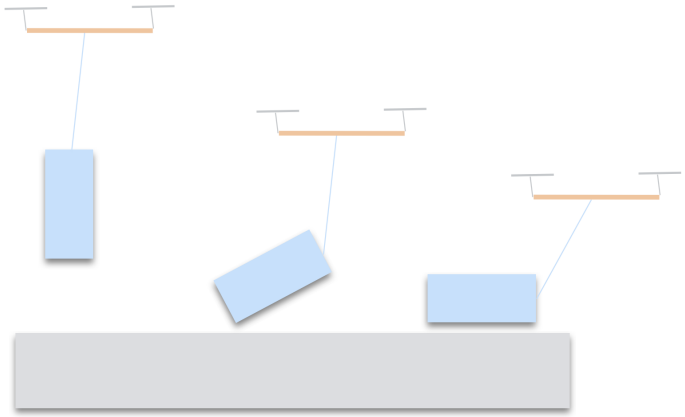


Figure 2.7: A drone can make use of the environment to achieve its goal. Courtesy of Hann Nguyen @ LAAS

As for the manipulation problem, previous solutions for situations involving discontinuities consisted of proposing several high-level primitives, such as "grasping", "pushing", "pivoting", etc. The work by [Byrne 2001], for example, observes 72 manipulation primitives in gorillas, which means that this method of enumerating and designing motion primitives might get out of hand fast if one deals with more complex manipulation problems.

In more recent work by [Cheng 2021b] a different approach has been proposed: instead of defining motion primitives, it is possible to list different "modes" for the object's contact points, with each one of these modes being related to a set of constraints on force and velocity. These constraints can be than added to a path planning algorithm, such as the CBiRRT proposed in [Berenson 2009], in order to generate the trajectories that must be followed by the drones. What was proposed by LAAS as the objective of the internship was to adapt these tools to the context of UAVs in order to try and solve the problems of path planning and object manipulation together. We shall explore the formalization of this problem in the next chapter.

Chapter 3

Formulation of the UAV-Object Manipulation Problem

3.1 Formal Description of the Problem

To begin our description of the problem, it is important to describe the physical elements involved, which are listed below and illustrated on Fig. 3.1, following the same rationale of [Cheng 2021b].

- Object $\mathcal{O}$: Rigid polygonal object. We assume the object is homogeneous in density and that its friction coefficient μ_{obj} is the same throughout its surface. This is the payload that will be manipulated by the drones.

- Environment $\mathcal{E}$: Collection of static, impenetrable polygonal shapes in the movement region. Each shape of the environment can have a different friction coefficient μ_{env}.

- Drones d_i: Each drone d_i is represented as a fully actuated punctual object with mass m_i. We have a total of N_{uav} drones involved. Also, as a further restriction, each drone is connected to exactly one point on the object and no point on the object is connected to more then one drone. The lengths of the cables connecting each drone to the object are known and the cables are ideal.

- Contact points c_i: points where the object is either touching one of the surfaces of the environment or is attached to one of the cables.

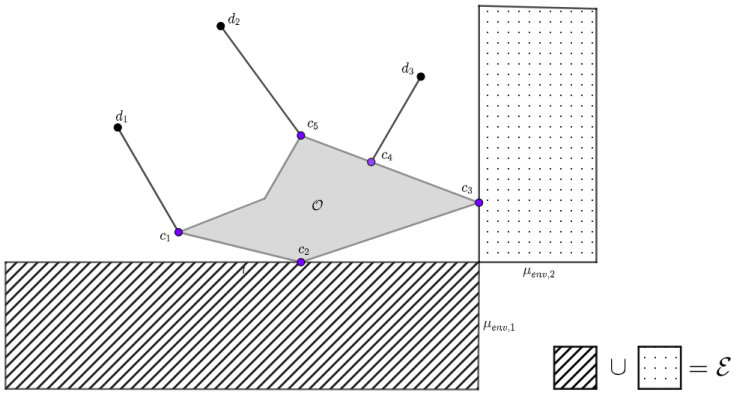


Figure 3.1: Physical elements used in the manipulation problem. Numbering is arbitrary for this example

The existence of these objects allow us to define the frames of reference which are relevant to the problem, as shown in Fig. 3.2. The world frame W is an inertial frame fixed somewhere in the world; the body frame B is fixed at a point on the manipulable object (not necessarily its center of mass); finally, each contact point has its own contact frame C_i, which is a right-handed frame of reference where the y direction is defined by the contact normal that points towards the interior of the object.

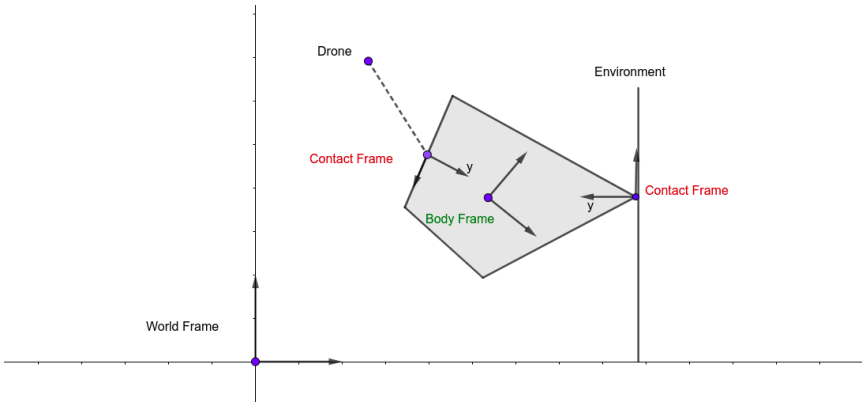


Figure 3.2: Frames of reference involved in the problem

With these frames in mind, together with the previous definitions presented, we can define some key terms that will be used when solving this problem, while also further formalizing some of the previously shown concepts.These also follow the same rationale of [Cheng 2021b].

- Object configuration q: An object configuration is the tuple (x, y, θ) that defines the translation and rotation of frame B w.r.t. the frame W.

- Drone configuration $\vec{d}$. Each drone has a position $\vec{d_i} = \begin{bmatrix} d_{ix} \\ d_{iy} \end{bmatrix}$ w.r.t. frame W[1]. Combining all drone positions, we get the drone configuration $\vec{d} = \begin{bmatrix} \vec{d_1} \\ \vec{d_2} \\ \dots \\ \vec{d}_{N_{uav}} \end{bmatrix}$. Following the nomenclature in [Fink 2011], the problem of finding $\vec{d}(q)$ is called the inverse configuration problem, while finding $q(\vec{d})$ is called the direct configuration problem.

- Contact points c_i: Each contact point is defined by three attributes: its position, normal, and type. The position $\vec{p_i} = \begin{bmatrix} p_{ix} \\ p_{iy} \end{bmatrix}$ and unitary normal $\vec{n}_i = \begin{bmatrix} n_{ix} \\ n_{iy} \end{bmatrix}$, $\|\vec{n}_i\| = 1$ are written w.r.t. frame B. It is worth reinforcing the point that the normal points towards the interior of the object. Property type $\in$ {environment, attachment} defines the nature of the contact point.

- Contact mode m_i: each contact point c_i is assigned a contact mode m_i which will describe the interaction between the object and the surface/cable at that point. For contact points of type "environment", we have $m_i \in$ {fixed, separate, right-slide, left-slide}. For contact points of type "attachment", we have $m_i \in$ {strained, loose}. The contact mode will define which constraints are acting on the contact point.

- Contact forces $\vec{\lambda}_i$ and contact velocities $\vec{v}_{ci}$: on the object's side of the contact interaction, each contact point is subject to contact force $\vec{\lambda}_i = \begin{bmatrix} \lambda_{ix} \\ \lambda_{iy} \end{bmatrix}$ and contact velocities $\vec{v}_{ci} = \begin{bmatrix} v_{cix} \\ v_{ciy} \end{bmatrix}$, both written w.r.t. the contact frame C_i.

Given these definitions, the problem we wish to solve consists of: given an environment $\mathcal{E}$ and object $\mathcal{O}$, an initial object configuration q_{init} and a goal configuration q_{goal}, we wish to find the drones' configuration path $\vec{d}^k$, with k representing discrete steps, such that we have $q(\vec{d}^0) = q_{init}$ and $dist(q(\vec{d}^N), q_{goal}) < Q$ for some value N, distance function for $SE(2)$ $dist(q_1, q_2)$ and value Q. Moreover, for every $0 \leq k \leq N$, we must respect the constraints imposed by the environment/cables and the maximum limits admitted for the drones.

[1] I apologize for using d_i to refer to the drone and $\vec{d_i}$ to refer to the drone's position. I was running out of significant letters.

3.2 Problem Tools

In order to develop the problem, we need to use some mathematical tools. These are mostly described in [Murray 2017] and [Eade 2013], which are the main sources of this section, unless otherwise stated.

3.2.1 SE(2) Algebra

Given the definition of the object configuration, we can find a one-to-one correspondence between the configuration q and the homogeneous matrix representation g_{WB} of the transformation from frame B to W if we limit $\theta \in [-\pi, \pi)$, such as in (3.1). This is not too complicated to visualize: after all, the object's configuration q is just a shorthand description of the frame B w.r.t. frame W. Since the configuration notation and the homogeneous matrix notation are equivalent, with the latter simply being the preferred form for algebraic multiplication, they will be used interchangeably and both will be referred to as the object's pose. Also, the concept of transformation matrix, although used in (3.1) to express the relation between frames B and W, can be extended to describe the transformation between any two reference frames.

$$q = (x, y, \theta) \iff g_{WB} = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \in SE(2) \tag{3.1}$$

Considering that we have $g_{WB} \in SE(2)$, all the properties of $SE(2)$ algebra apply. For example, we can define a *twist* $\hat{\xi} \in se(2)$ such that:

$$e^{\hat{\xi}} \in SE(2)$$

Twists are of the general form shown in (3.2). One can see that any real multiple of this general form is also a general twist. Given that definition, we can define operator "vee" V which yields ξ, the so-called *twist coordinates* of $\hat{\xi}$, as shown in (3.3). In an analogous sense, it is not hard to define the inverse operator "wedge" $\hat{}$, as in (3.4), which transforms twist coordinates to a twist $\in se(2)$.

$$\hat{\xi} = \begin{bmatrix} 0 & -\omega & v_x \\ \omega & 0 & v_Y \\ 0 & 0 & 0 \end{bmatrix} \tag{3.2}$$

$$\hat{\xi}^V = \begin{bmatrix} 0 & -\omega & v_x \\ \omega & 0 & v_Y \\ 0 & 0 & 0 \end{bmatrix}^V \implies \xi = \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} \tag{3.3}$$

$$\hat{\xi} = \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}^{\wedge} \implies \hat{\xi} = \begin{bmatrix} 0 & -\omega & v_x \\ \omega & 0 & v_Y \\ 0 & 0 & 0 \end{bmatrix} \tag{3.4}$$

Using the general form of the twist from (3.2), we can calculate the value of the matrix exponential in (3.5) (which conversely allows the calculation of a matrix logarithm for $SE(2)$, such as in (3.6)). These expressions can be computed from the infinite expansion of the general matrix exponential, but this is left implicit. Also, the value of the expression for when $\omega = 0$ can be obtained from the limit, resulting in identities.

$$\hat{\xi} = \begin{bmatrix} 0 & -\omega & v_x \\ \omega & 0 & v_Y \\ 0 & 0 & 0 \end{bmatrix} \implies e^{\hat{\xi}} = \begin{bmatrix} \cos\omega & -\sin\omega & \frac{1}{\omega}[v_x \sin\omega - v_y(1-\cos\omega)] \\ \sin\omega & \cos\omega & \frac{1}{\omega}[v_x(1-\cos\omega) + v_y \sin\omega] \\ 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

$$e^{\hat{\xi}} = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \implies \hat{\xi} = \begin{bmatrix} 0 & -\theta & \frac{\theta}{2}[x\cot\frac{\theta}{2} + y] \\ \theta & 0 & \frac{\theta}{2}[-x + y\cot\frac{\theta}{2}] \\ 0 & 0 & 0 \end{bmatrix} \tag{3.6}$$

The advantage of these definitions is that they allow us to define a body velocity, which is the linear and angular velocity of the frame B, as measured by an observer at the frame W, but expressed in terms of the frame B. Mathematically, the body velocity twist $\hat{v}_b$, which can be easily converted to and from twist coordinates, is defined in (3.7), where Δt represents a time step and t is a general time parameter[2]. We can use this velocity to update the body's pose, or conversely we can find a velocity given a desired pose and a time interval, such as in (3.8) (where for convenience we have set $(t = 0)$). However, for these equations, it is important to stay in regions where $\hat{v}_b$ is piecewise constant.

$$g_{WB}(t + \Delta t) = g_{WB}(t) e^{\hat{v}_b \times \Delta t} \tag{3.7}$$

$$\hat{v}_b = \frac{1}{\Delta t} \log[g_{WB}^{-1}(0) g_{WB}(\Delta t)] \tag{3.8}$$

As for the absolute distance between to elements $q_1, q_2 \in SE(2)$, we will use the same function offered in [Cheng 2021b], expressed in (3.9), where w_r is a weight parameter.

$$dist(q_1, q_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + w_r \min(|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|) \tag{3.9}$$

[2]The reason why we didn't use the notation $\vec{v}_b$ for the body velocity in twist coordinates was to avoid notation overload when dealing with the twist $\hat{\vec{v}}_b$

The final tool from $SE(2)$ algebra needed to describe our problem is the adjoint matrix Ad_{pose}, expressed in (3.10), which transforms tangent spaces and will be used to build the so-called "grasp map" on the object.

$$g = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \implies Ad_g = \begin{bmatrix} \cos\theta & -\sin\theta & y \\ \sin\theta & \cos\theta & -x \\ 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

3.2.2 Grasp Map: Relating Contact Frames To Body Frame

The grasp map G is a matrix which allows us to convert forces acting on the contact points, expressed in the contact frames, to a wrench acting on the body frame B and expressed in that frame. The construction of this matrix follows the same method as in [Murray 2017]. First, we can find the transformation matrix g_{BC_i} from contact frame C_i to body frame B using $\vec{p}_i$ and $\vec{n}_i$, which are geometrically determined and known at each instant. This construction is expressed in (3.11).

$$g_{BC_i} = \begin{bmatrix} n_{iy} & n_{ix} & p_{ix} \\ -n_{ix} & n_{iy} & p_{iy} \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

Knowing the contact frame's transformation matrix, we can built the contact map G_i as in (3.12), which relates the force $\vec{\lambda}_i$ to the wrench $\vec{F}_i^B$ on the frame B via the equation $\vec{F}_i^B = G_i\vec{\lambda}_i$. It is worth noting that the matrix at the right is what converts the force on the contact to the wrench on the contact frame (which is assumed to lack a torque). To build the complete grasp map, we can consider that the total wrench on the body frame as a sum $\vec{F}^B = \sum_i \vec{F}_i^B$, which results in a grasp map G as defined in (3.13), where n is the total number of contact points and $\vec{F}^B = G\vec{\lambda}$, with $\vec{\lambda} = \begin{bmatrix} \vec{\lambda}_1 \\ \vec{\lambda}_2 \\ \dots \\ \vec{\lambda}_n \end{bmatrix}$

$$G_i = Ad^T_{g^{-1}_{BC_i}} \times \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{3.12}$$

$$G = [G_1, G_2, ..., G_n] \tag{3.13}$$

Conversely, we can also use the grasp map to relate the body velocity in twist coordinates v_b to the velocity of the contact points in the contact frame. This is done in (3.14),

where $\vec{v}_c = \begin{bmatrix} \vec{v}_{c1} \\ \vec{v}_{c2} \\ \dots \\ \vec{v}_{cn} \end{bmatrix}$.

$$\vec{v}_c = G^T v_b \tag{3.14}$$

The reason for these conversions is that we will apply the constraints at the contact points, but we want to see how they relate to the object's movement.

3.2.3 Drone Limitations and Description

As was said before, each drone d_i is subject to certain limitations. These are:

- Limitation on the maximum thrust T_i^{max}. Considering the thrust $\vec{T}_i$ exerted by the drone, we must have $\|\vec{T}_i\| \leq T_i^{max}$ at all times. This will be important when solving the problem of finding the forces on the object.
- Limitation on the maximum banking angle ϕ_i^{max}. We can find the banking angle ϕ_i of each drone w.r.t. the world vertical by knowing its thrust and applying $\phi_i = \arccos\left(\frac{-\vec{g}}{\|\vec{g}\|} \cdot \frac{\vec{T}_i}{\|\vec{T}_i\|}\right) = \arccos\left(\frac{T_{iy}}{\|\vec{T}_i\|}\right)$. At each instant, we must have $\phi_i \leq \phi_i^{max}$

If we can determine all the contact forces $\vec{\lambda}$ acting upon the body, we can find the value of $\vec{T}_i$ for each drone by applying simple transmission of the forces through the cable together with some frame rotations. Suppose drone j is connected to contact point i on the body, then the value of $\vec{T}_j$ is given by (3.15), where the R matrices represent rotation matrices $\in SO(2)$. It is worth noting that the value of $\vec{T}_j$ is written w.r.t. the world frame W.

$$\vec{T}_j = R_{WB} \times R_{BC_i} \times \vec{\lambda}_i - m_j \vec{g} \tag{3.15}$$

Moreover, we can also use the forces to find the position $\vec{d}_j$ of the drone w.r.t. the world frame. If we consider that the forces can only act in the direction of the cable and considering that cable of length l_{ij} connects contact point i to drone j, the position of drone j in the world frame is given by (3.16), whic assumes a cable force pulling on the object. It goes without saying these positions should be valid: for example, the drone can't be inside a part of the environment or inside the object.

$$\begin{bmatrix} \vec{d}_j \\ 1 \end{bmatrix} = g_{WB} \times \begin{bmatrix} l_{ij} \times R_{BC_i} \times \frac{\vec{\lambda}_i}{\|\vec{\lambda}_i\|} + \vec{p}_i \\ 1 \end{bmatrix} \tag{3.16}$$

3.2.4 Constrained motion

The choice of a contact mode m_i on a contact point c_i leads to the application of constraints on the contact force $\vec{\lambda}_i$ and on the contact velocity $\vec{v}_{ci}$. The original formulation of these constraints comes from [Cheng 2021b], but here they are broken down into a force problem and a velocity problem, for reasons that will become apparent when we deal with the path planning algorithm. Moreover, the solution of the force problem allows us to solve the inverse configuration problem in an efficient manner.

Force Problem

For contact points of type environment, the constraints acting on the contact forces $\vec{\lambda}_i$ are a function of the contact mode, as shown in (3.17). In these equations, we have the friction coefficient $\mu = \mu_{obj}\mu_{env}$, where μ_{env} is evaluated on the environment at the contact point.

$$\begin{cases} \lambda_{ix} = 0, \lambda_{iy} = 0, \text{if } m_i = \text{separate} \\ \lambda_{iy} > 0, -\mu\lambda_{iy} < \lambda_{ix} < \mu\lambda_{iy}, \text{if } m_i = \text{fixed} \\ \lambda_{iy} > 0, \mu\lambda_{iy} + \lambda_{ix} = 0, \text{if } m_i = \text{right-slide} \\ \lambda_{iy} > 0, \mu\lambda_{iy} - \lambda_{ix} = 0, \text{if } m_i = \text{left-slide} \end{cases} \tag{3.17}$$

The force constraints for contact points of type attachment can be seen in (3.18) as functions of the point's contact mode. The rationale behind these constraints is that the ideal cable can only pull, but not push, with "pulling" being defined as a force in the negative y direction of the contact frame C_i. It is important to note that satisfying the force constraints in the attachment contact points does not necessarily mean that the limits on the drones' thrust and banking angle will be respected.

$$\begin{cases} \lambda_{iy} < 0, \text{if } m_i = \text{strained} \\ \lambda_{ix} = 0, \lambda_{iy} = 0, \text{if } m_i = \text{loose} \end{cases} \tag{3.18}$$

Equations (3.17) and (3.18) specify the constraints on each contact point, but the constraints on all points must be satisfied at the same time. Therefore, we must consider not each point individually, but the total set of n contact points $c = [c_1, c_2, ..., c_n]^T$, contact modes $m = [m_1, m_2, ..., m_n]^T$ and contact forces $\vec{\lambda} = [\vec{\lambda}_1^T, \vec{\lambda}_2^T, ..., \vec{\lambda}_n^T]^T$. Moreover, we will solve the problem of the forces acting upon the object under a quasi-static assumption, meaning that at each instant we must satisfy (3.19), where $\vec{F}_e^B(q)$ is the external wrench applied on the object, which does not from contact forces, as viewed in the body frame B, for some configuration q of the object. This equation follows naturally from the construction of the grasp map if we wish that the object be at static equilibrium at all times. This

model, which assumes no accelerations, admits velocities on the object, but does not deal with the mechanism of how to imprint such velocities.

$$G(c)\vec{\lambda} + \vec{F}_e^B(q) = 0 \tag{3.19}$$

Equations (3.17)-(3.19) are either linear equations or linear inequalities acting on the contact points, meaning it is possible to compile all information about the constraints in a matrix system of equations and inequalities. Such a system is written in (3.20), where the matrices involved are a function of the contact modes m (which also defines the contact types, since there are no two modes alike for different types). Moreover, matrix $A_{eq,f}$ is also a function of the contact points c, insofar as the information about the grasp map is embedded in it. The same logic applies when explaining that b_{eq_f} is a function of q, since this matrix contains $\vec{F}_e^B(q)$.

$$\begin{cases} A_{eq,f}(m,c)\vec{\lambda} = b_{eq,f}(m,q) \\ A_{ineq,f}(m)\vec{\lambda} > b_{ineq,f}(m) \end{cases} \tag{3.20}$$

The system written in (3.20) is not necessarily determined, which means these equations might not be enough to calculate the value of $\vec{\lambda}$. However, we can use this system as constraints to a minimization problem, such as the one shown in (3.21).

$$\begin{aligned} \min_{\vec{\lambda}} \quad & \|\vec{\lambda}\|^2 \\ \text{s.t.} \quad & A_{eq,f}\vec{\lambda} = b_{eq,f} \\ & A_{ineq,f}\vec{\lambda} > b_{ineq,f} \end{aligned} \tag{3.21}$$

By solving (3.21), which is a quadratic minimization problem subject to linear constraints, we find the set of contact forces $\vec{\lambda}$ with the smallest module that still respects the constraints imposed by the contact modes. It is important to remark that these operations take place before the object's movement, which means that there is no information whether it is possible to solve the force problem after applying a velocity to the object.

Finding $\vec{\lambda}$ by means of (3.21) allows us to solve the inverse configuration $\vec{d}(q)$ by inputting $\vec{\lambda}$ in (3.16). We can also use the contact forces to calculate the drones' thrusts in (3.15). The values of $\vec{d_i}$ and $\vec{T_i}$ can then be used to verify if no violation of the drones' limits ocurred and if their positions are valid.

Velocity Problem

The solution of the velocity problem follows a similar rationale to that of the force problem and was also first proposed in [Cheng 2021b]. We start by defining a direct body

velocity v_d between two object configurations q_k and q_{k+1}, to be obtained from (3.8). There are, however, constraints on the contact velocities as a function of the contact modes. It is worth remarking that these contact modes are the same that were chosen during the solution of the force problem. For contact points of environment type, these constraints are listed in (3.22). There are no constraints on the velocity of attachment contact points.

$$\begin{cases} v_{ciy} > 0, \text{if } m_i = \text{separate} \\ v_{ciy} = 0, v_{cix} = 0, \text{if } m_i = \text{fixed} \\ v_{ciy} = 0, v_{cix} > 0, \text{if } m_i = \text{right-slide} \\ vciy = 0, v_{cix} < 0, \text{if } m_i = \text{left-slide} \end{cases} \tag{3.22}$$

All these constraints are, again, linear equations and inequalities, meaning that they can be compiled into a single system by joining the contact velocities $\vec{v}_c = [\vec{v}_{c1}^T, \vec{v}_{c2}^T, ..., \vec{v}_{cn}^T]^T$. The system is shown in (3.23).

$$\begin{cases} A_{eq,vc}(m)\vec{v}_c = b_{eq,vc}(m) \\ A_{ineq,vc}(m)\vec{v}_c > b_{ineq,vc}(m) \end{cases} \tag{3.23}$$

We are interested in the velocity of the object, not of the contact points. If we consider v_o the object's body velocity in twist coordinates, we can use the grasp map transformation in (3.14). This transformation means we can write (3.23) in terms of v_o in (3.24), with $A_{eq,vo} = A_{eq,vc}G^T$ and $A_{ineq,vo} = A_{ineq,vc}G^T$.

$$\begin{cases} A_{eq,vo}(m,c)v_o = b_{eq,vo}(m) \\ A_{ineq,vo}(m,c)v_o > b_{ineq,vo}(m) \end{cases} \tag{3.24}$$

We can use the system in (3.24) as a constraint to a minimization problem that allows us to find v_o such that v_o is the closest possible velocity to v_d that still respects the contact mode constraints (3.25).

$$\begin{aligned} \min_{v_o} \quad & \|v_o - v_d\|^2 \\ \text{s.t.} \quad & A_{eq,vo}v_o = b_{eq,vo} \\ & A_{ineq,vo}v_o > b_{ineq,vo} \end{aligned} \tag{3.25}$$

Solving this velocity problem yields v_o, which can be used to update the body's position to q'_{k+1} via the implicit relation in (3.26), which is the same as (3.7), but with an explicit reference to the object's transformation matrix as a function of its configuration. This equation uses discrete steps (separated by time Δt) when implemented in a computer program.

$$g_{WB}(q'_{k+1}) = g_{WB}(q_k) \times e^{\Delta t \hat{v}_o} \tag{3.26}$$

Movement Manifolds

Given a set of contact points, the choice of contact modes restricts the movement of the object to certain configuration spaces, which can intersect among each other. Consider, for example, the object in Fig. 3.3, where the environment contact point is set either to right-slide or fixed, and where all restrictions on the drones' movement have been ignored. When we evaluate the boundaries of the configuration regions defined by each environment contact mode, we can see that they intersect, as shown in Fig. 3.4. These intersections are what makes a solution to the path planning problem possible, for they enable the transition between two different configuration spaces.

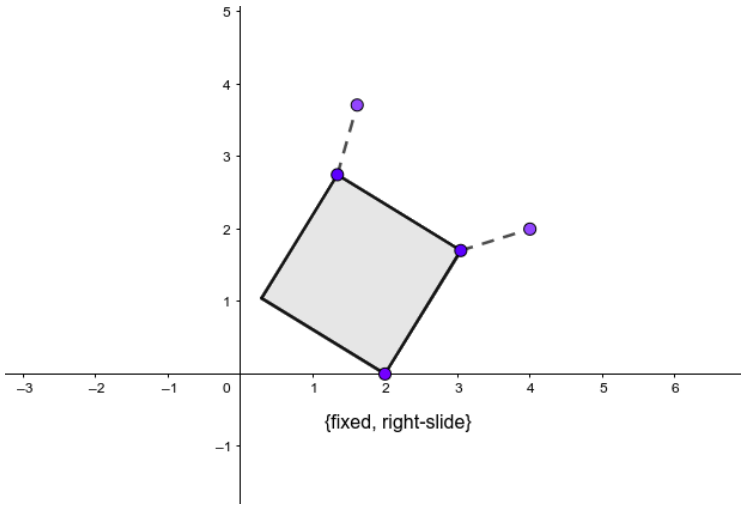


Figure 3.3: This object has two attachment points and a single environment contact point. The environment contact point can be set to "fixed" or "right-slide"

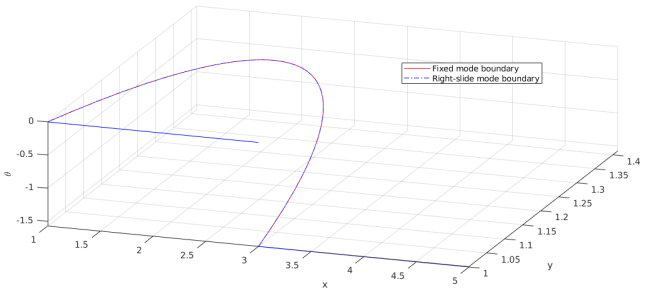


Figure 3.4: Boundaries of the configuration regions for two different contact modes

Chapter 4

Path Planning Algorithm

4.1 Classes and Program Diagram

In order to explain the algorithm used, we will describe the classes implemented and their relations.

4.1.1 Algebra Module

The module *lie.py* implements the tools discussed in section 3.2.1 (and some tools for rotation matrices $\in SO(2)$). This module was built around the numpy matrix structure, which is similar to the traditional array but optimized for 2D arrays. This module is just a collection of static functions and has no constructors, therefore its "objects" cannot be used directly as attributes by other modules, but *lie.py* serves as a background tool for most of the operations performed. The reduced UML diagram (which includes no specification on variable typing or dependency relations) for the module can be seen in Fig. 4.1 and gives an overall sense of the module.

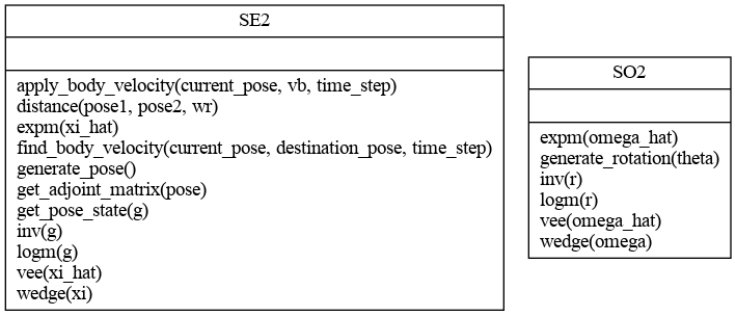


Figure 4.1: Reduced UML diagram for *lie.py* module

4.1.2 Environment

In order to define the environments, which is done in the module *environments.py*, we need two classes: EnvironmentShape and Environment, whose UML diagrams are shown in Fig. 4.2. Concerning the functionality of this module, each EnvironmentShape object is composed of a geometry, which is a list of (x, y) points defining vertices for a convex geometry, a state defining where the shape's frame of reference is w.r.t. the world, and a series of properties (friction coefficient μ_{env}, elasticity and color). An Environment object is composed of a list of EnvironmentShape objects, plus some class constants to define all environments. Notice that it is possible to create non-convex shapes in the environment by overlapping convex shapes. The function that should be called is *generate_ environment*, which takes a name as an argument and returns an Environment object. Four environments are currently supported, which are illustrated in Fig. 4.3: "plain_surface", "rugged_surface", "ground_and_central_hill", "ground_and_central_bump".

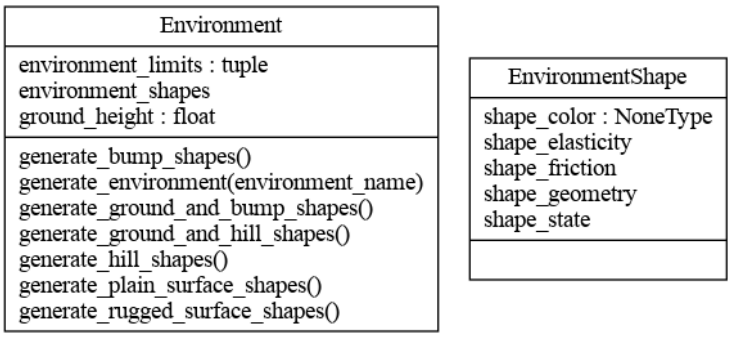


Figure 4.2: Reduced UML diagram for *environments.py* module

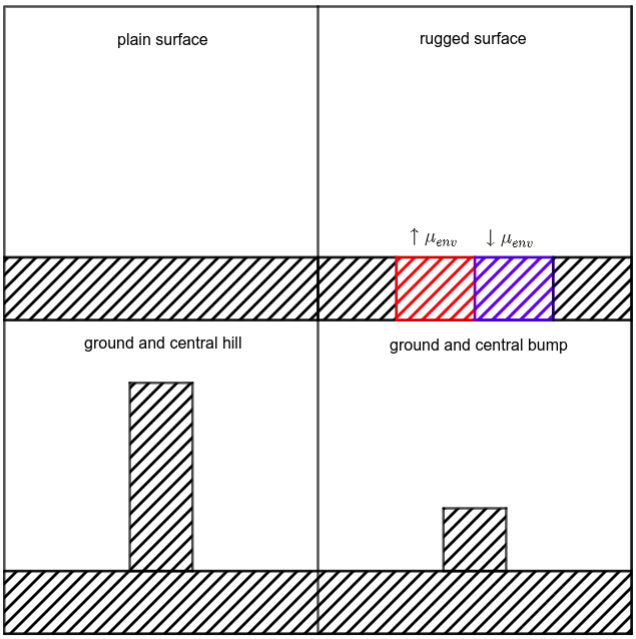


Figure 4.3: Schemes of the current supported environments

4.1.3 Drones

Module *drones.py* is quite straightforward and its UML is show in Fig. 4.4: it is composed of a single GenericDrone class, which contains the parameters needed to define the maximum thrust, maximum banking angle, and the mass of some generic drone. Objects of this class also have an attribute defining their position w.r.t. the world frame.

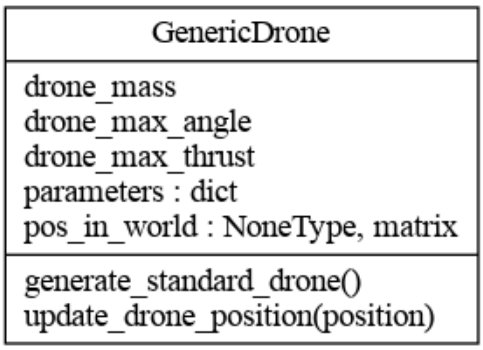


Figure 4.4: Reduced UML diagram for *drones.py* module

4.1.4 Manipulable Object

The ManipulableObject class, whose simplified UML is in Fig. 4.5, works similarly to the Environment class insofar as it is also constituted of several convex shapes (ObjectShape objects) which bundle together to form a single object $\mathcal{O}$. The difference is that ManipulableObject objects also contain information about drone connections, such as the drones d_i in question, the attachment points and normals, and the length of the cables. An object from the ManipulableObject class is the element that will be manipulated by the drones. Finally, since we will be dealing with squares by default, a function just to create square objects has been implemented.

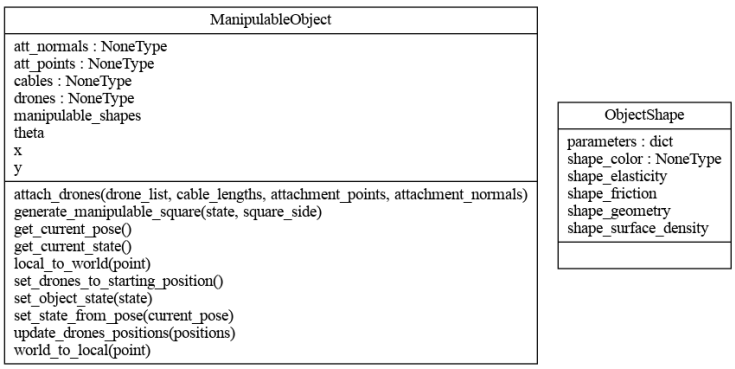


Figure 4.5: Reduced UML diagram for *objects.py* module

4.1.5 Simulator

The *simulation.py* module contains the Simulator class, which is mainly used to determine contact points between the object and its environment. This class uses the Pymunk package [Blomqvist 2022], which is a simple, python-based, 2d physics library, as a backend engine. The environment, manipulable object and drones are added to the simulation space, which can alter some of their properties, such as the current configuration. Moreover, the Simulator object also detects shape overlapping, which is useful during the movement phase of the program, since overlaps between the environment and the object can occur. The UML for the Simulator class is shown in Fig. 4.6.

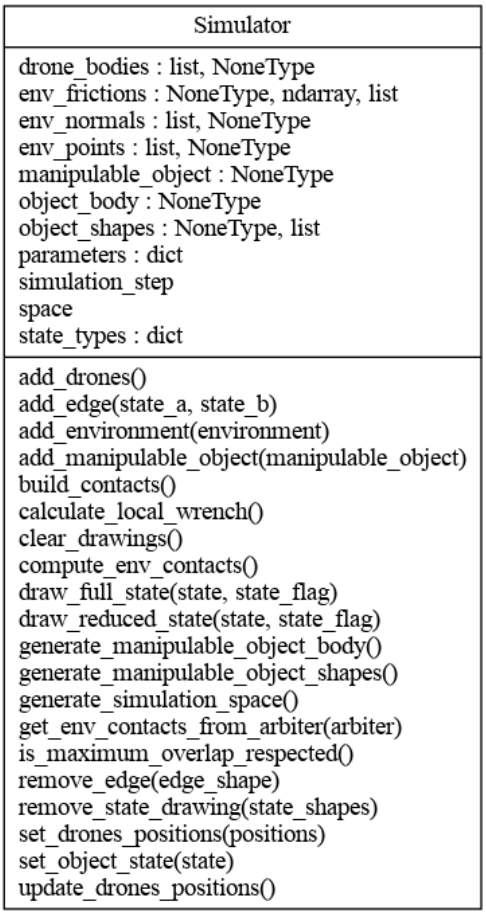


Figure 4.6: Reduced UML diagram for *simulation.py* module

4.1.6 Constraint Calculator

In the *constraints.py* module, we implement the equations defined in sections 3.2.3 - 3.2.4. The UML for class ConstraintCalculator is show in Fig. 4.7. It is worth noting that this class works in close proximity with the Simulator class when obtaining contact information. Furthermore, this class uses the library qpsolvers [Caron 2022] to solve the minimization problems defined in 3.21 and 3.25. One important aspect of the computational implementation of the minimization problem is that the package transforms the strict inequalities presented into non-strict inequalities, meaning that solutions for two different contact modes can be numerically equal.

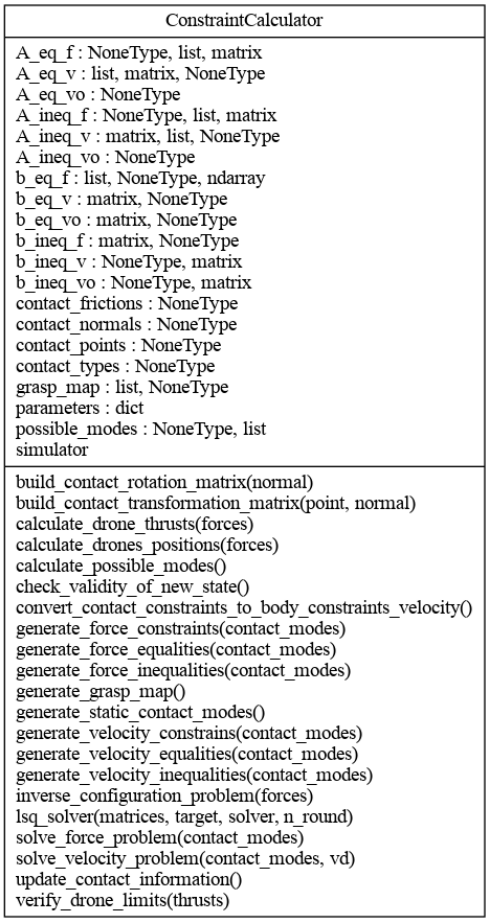


Figure 4.7: Reduced UML diagram for *contraints.py* module

4.1.7 Scenario

Finally, the Scenario class, whose UML is in Fig. 4.8, is comprised of parts from all the previous classes, plus a few other attributes. In terms of the classes previously presented, a scenario is comprised of:

- 1 Environment object
- 1 ManipulableObject object, to which are linked N_{uav} GenericDrone objects
- 1 Simulator object, which is linked to the Enironment object and to the ManipulableObject object
- 1 ConstraintCalculator object, which is liked to the Simulator object

As for the attributes of this class which were not previously defined, their explanation is as follows:

- *int_time*: Value of Δt to be used in (3.7)
- *vb_time*: Value of Δt to be used in (3.8)
- *init_state*: Initial configuration q_{init} of the manipulable object
- *goal_state*: Goal configuration q_{goal} of the manipulable object
- *parameters*: Static dictionary containing some default values for the class

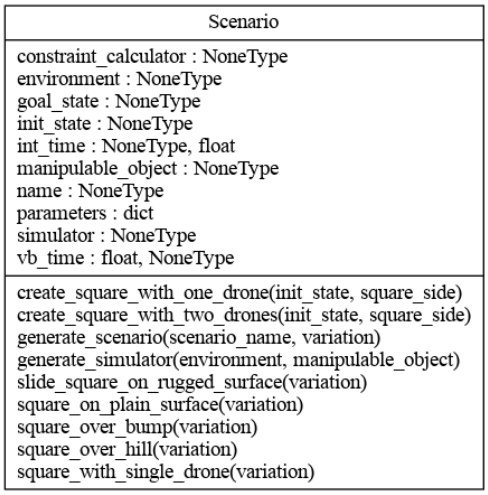
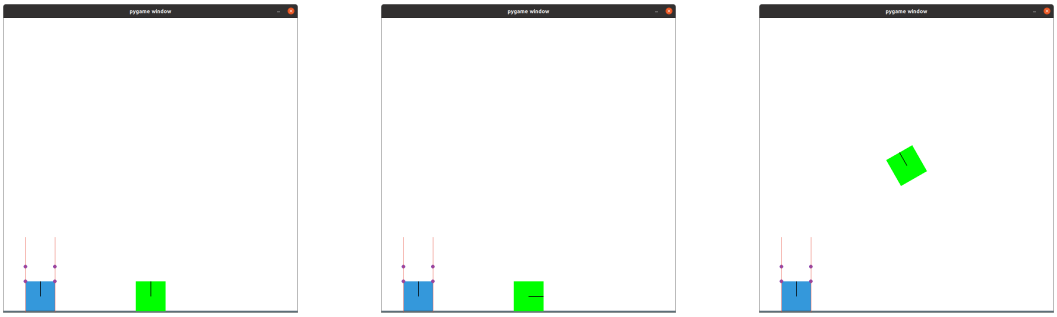


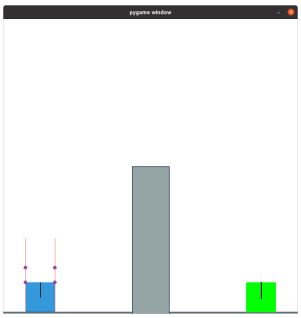
Figure 4.8: Reduced UML diagram for *scenarios.py* module

As for the attribute *name*, this is what defines the scenario and is used as one of the arguments of static function *generate_scenario*. The other argument for the function is the variable *variant*, which defines subtle changes upon the scenario, namely the initial and goal configurations. The scenario names and their descriptions, together with possible variants, are shown in Figs. 4.9 - 4.13, where the initial configuration is in blue and the goal configuration is in green. When calling the main function, the name of the scenario and the number of the variant desired are inputs supplied by the user. The python library PyGame [Shinners 2022] was used to visualize these scenarios.



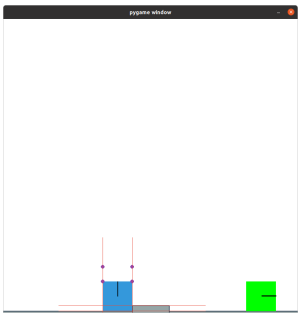
(a) Variant 1 (b) Variant 2 (c) Variant 3

Figure 4.9: Scenario: “square_on_surface”



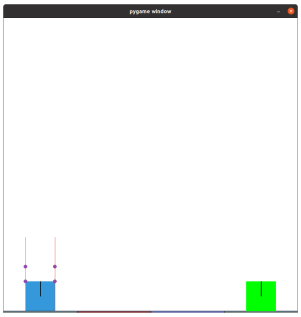
(a) Variant 1

Figure 4.10: Scenario: “square_over_hill”



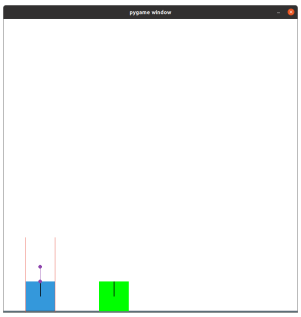
(a) Variant 1

Figure 4.11: Scenario: “square_over_bump”

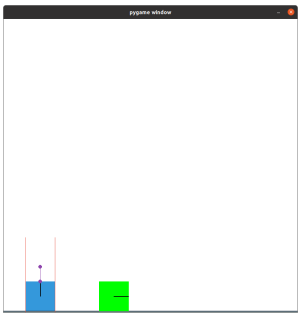


(a) Variant 1

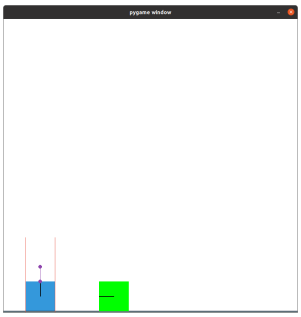
Figure 4.12: Scenario: “slide_square_on_rugged_surface”



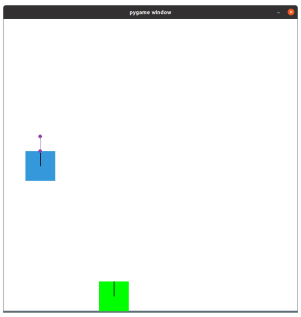
(a) Variant 1



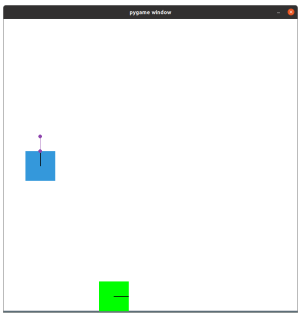
(b) Variant 2



(c) Variant 3



(d) Variant 4



(e) Variant 5

Figure 4.13: Scenario: “square_with_single_drone”

4.2 RRT Path Planning Algorithm

The main sequence for executing our version of the rrt algorithm can be seen in algorithm 1, which will now be explained part by part. First of all, we should define the input and output of this algorithm. To run this algorithm, the inputs are the scenario name and the variation desired, as well as if the user desires the algorithm to be animated or not.

As for the other parameters that could be adjusted, such as the times Δt involved and the weight w_r in the distance function, these are variables which are saved in the *parameter* attributes of the respective classes.

```
$ python main.py <scenario_name> <variation> <animate>
```

- *scenario_name*: one of the names defined in section 4.1.7.
- *variation*: integer number.
- *animate*: "false", "partial" or "full". Details the level of animation desired. Affects time performance of the algorithm.

Three files should be seen as the output of the program: one "tree_info.txt", which contains information about the program execution, and two json files, with one representing the full tree and the other representing the path to the goal. As for the tree, it was modeled as a dictionary of the form $\{q_i : [(q_d, \vec{d}, \text{contact-modes})]\}$, which means that the tree is a dictionary where each key is an object configuration q_i which points to a list of edges, where edges are defined by the destination object configuration q_d obtained from 3.26, the position of the drones for the movement $\vec{d}$, and the contact modes involved in the movement.

We shall now describe the functions involved in algorithm 1. Right off, the function "*generate-random-node*" returns a random object configuration q_{rand} which respects the limits of the environment. The argument of the function is a probability which specifies the chance of $q_{rand} = q_{goal}$. This argument can be varied to account for more aggressive pursuits of the goal. The function "*find-closest-node*" searches all the keys in the tree and returns the object configuration which is the closest to q_{rand}, based on (3.9). The next two functions "*snap-simulator-to-node*" and "*update-simulator-information*" simply recompute the contacts $\vec{p}$ and $\vec{n}$, as well as the grasp map G, on the object after changing its configuration q_{near}.

After updating the object's contact information, we calculate the direct body velocity between the current configuration and the desired configuration using (3.8), implemented in function "*direct-velocity*", since this does not depend on the contact mode chosen. Also, we can now enumerate the possible contact modes, which in total are $n_{modes} = 2^{n_{att}} \cdot 4^{n_{env}}$ where n_{att} and n_{env} are the number of attachment and environment contact points respectively. Of course, not every one of these modes will always be feasible, and [Mason 2001] even describes an algorithm for the purpose of determining which are, but we'll ignore this problem in this work, for there is little gain to be obtained from this method in 2D scenarios. To save programming time, it is easier to just iterate through all possible contact modes, regardless of *a priori* feasibility, which is what the algorithm does next.

Algorithm 1: Expansion of the random tree $\mathcal{T}$

```
1  Function expand(T: Tree)
2      while not T.found_goal and T.nodes < T.max_nodes do
3          q_rand = generate-random-node(T, prob)
4          q_near = find-closest-node(T, q_rand)
5
6          T.snap-simulator-to-node(q_near)
7          T.update-simulator-information()
8
9          v_d = direct-velocity(q_near, q_rand)
10         possible_modes = T.simulator-enumerate-modes()
11         for contact_modes ∈ possible_modes do
12             forces = solve-force-problem(contact_modes)
13             drones_pos, are_drones_valid = inverse-configuration(forces)
14
15             v_o = solve-velocity-problem(contact_modes, v_d)
16
17             if (are_drones_valid) and (v_o is not None) and (forces is not
                None) then
18                 q_new, is_node_valid = move-object(q_near, v_o)
19                 if is_node_valid then
20                     T.found_goal = T.is-goal(q_new)
21                     T.add-node(q_new)
22                     T.node(q_near).add-edge(q_new, drones_pos, contact_modes)
23                 end
24             end
25         end
26     end
27 end
```

Considering the constraints in (3.17), (3.18) and (3.22), we see that the value of the forces on the object and the feasible body velocity depend on the contact modes chosen. Function "*solve-force-problem*" compiles the matrices for the system in (3.20) and solves the minimization in (3.21) The value of these forces is then used in function "*inverse-configuration*" to calculate equations (3.16) and (3.15) and to verify if the drone limits are respected. Finally, the same contact modes are used to find the velocity constraint matrices in (3.24) and solve the problem in (3.25) in function "*solve-velocity-problem*", which yields v_o, the velocity which is the closest to v_d while still respecting the constraints imposed by the contact modes.

If there is a valid set of forces that solve the problem, and these forces result in valid drone positions, and there is a feasible velocity which respect the constraints, then we try to apply the velocity to the body using (3.26). The function "*move-object*" is a bit complex and is expanded upon in algorithm 2. First and foremost, at least one of the components of velocity v_o must be higher than a certain threshold in order for the movement to be computed. If that is the case, a new object configuration is obtained via function "*apply-body-velocity*", which implements (3.26). The tricky part is function "check-validity". This does two things: 1) it sees if the new object configuration is not overlapping any environment shape and 2) it checks if there is at least one possible set of forces that can keep the new configuration in static equilibrium. The function checks this set of forces by setting all attachment contact modes to "strained" and all environment contact modes to "fixed", which is the set of contact modes that exerts the least force upon the object. The reason for this check is to avoid tree nodes from which no expansion can possibly occur, for it is impossible to keep the object static. More on the consequences of this post-movement check will be shown in chapter 5.

Algorithm 2: Function "*move-object*"

```
1 Function move-object(q_current, v_o)
2     if abs(v_o) < threshold then
3         return q_current, False
4     else
5         q_new = apply-body-velocity(q_current, v_o)
6         is_state_valid = check-validity(q_new)
7         return q_new, is_state_valid
8 end
```

Finally, if the new configuration if valid, functions "*add-node*" and "add-edge" just add the corresponding node and edge, which are based upon the tree formulation which were previously shown.

Chapter 5

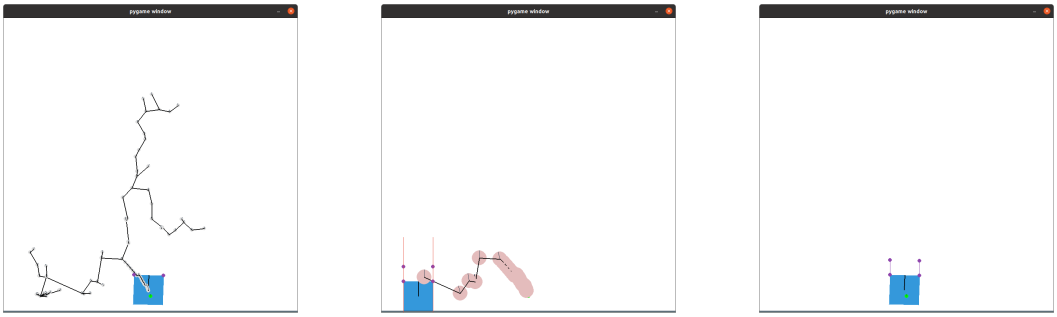
Results

The scenarios described in section 4.1.7 were run by means of the previous algorithm, with their results described below. Unless otherwise stated, the program was run with a limit of 500 nodes on the tree, a probability of 20% of sampling the goal configuration, and a goal radius of 0.2 (for $w_r = 1$). Considering the specific implementation of the simulation, it is important to state that the objects shown in the pictures were also actuated, with their positions controlled directly together with the position of the drones. The reason for this is that Pymunk does not deal well with moving constraints. Moreover, as will be discussed in Chapter 6, translating the movement of the drones into movement of the object is not always simple in practice. This is why we chose to also control the position of the object in these animations.

5.1 "Square_on_surface" Scenario

5.1.1 Variant 1

This simple scenario serves to test some basic properties of the algorithm. There is a remarkable property to this scenario in that its result, shown in Fig. 5.1 is counter-intuitive: instead of simply sliding the object on the surface, the drone lifts it and reaches the goal from above. This is not the most efficient way to solve the problem (meaning that the algorithm is not optimal), and it is clear that the object does not explore the manifold, but it arrives at the goal nevertheless. Some data for this scenario is available in Table 5.1.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.1: "Square_on_surface" variant 1

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	6.572 s	97	25	77	0.191

Table 5.1: Average values for scenario 1 - variant 1

5.1.2 Variant 2

If the block is allowed to leave the surface, the problem is not solved. Fig. 5.2 shows this situation: even though the object gets to a minimal distance of 0.275 to the goal configuration, it is still unable to reach this configuration. If we ensure that the block stays on the surface, the algorithm is successful in dealing with this scenario. Comparison with scenario 5, variant 2 (which was actually tested first), hints that keeping the object attached the surface is key to solving the rotation problem. Indeed, by restraining the object's move so that it stays attached to the surface, we get solutions such as the one in Fig. 5.3 and the results registered in Table 5.2.

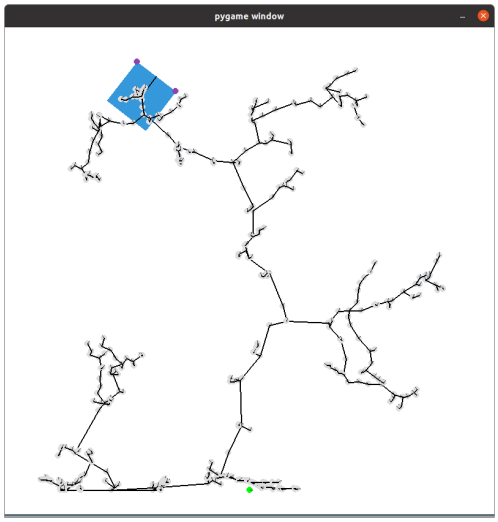
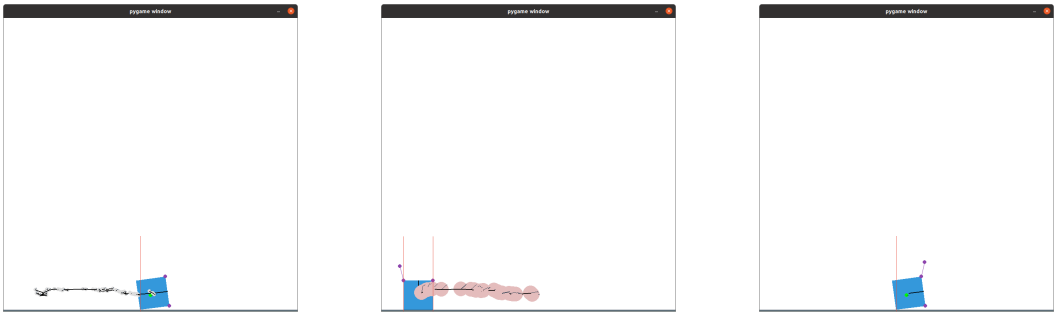


Figure 5.2: Extended tree - Scenario 1 variant 2



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.3: “Square_on_surface” variant 2 - alternate version with movement restricted to surface

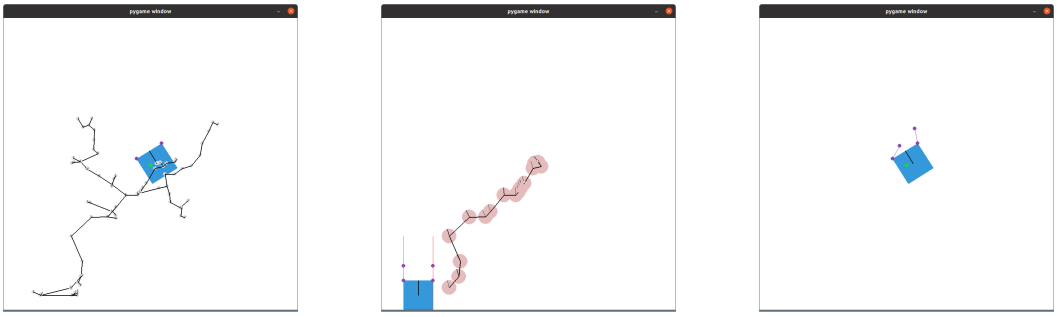
Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
4/5	8.879 s	95	29	64	0.151

Table 5.2: Average values for scenario 1 - variant 2

5.1.3 Variant 3

This scenario variant explores the capability of the drone to deal with goals that include rotation. By setting the goal in the air, we free the object from any constraint requirements except that it be in equilibrium at all times. We see in Fig. 5.4 that the object gets to the

goal, albeit through a kind of crooked path. Information about the results of this scenario are shown in Table 5.3.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.4: “Square_on_surface” variant 3

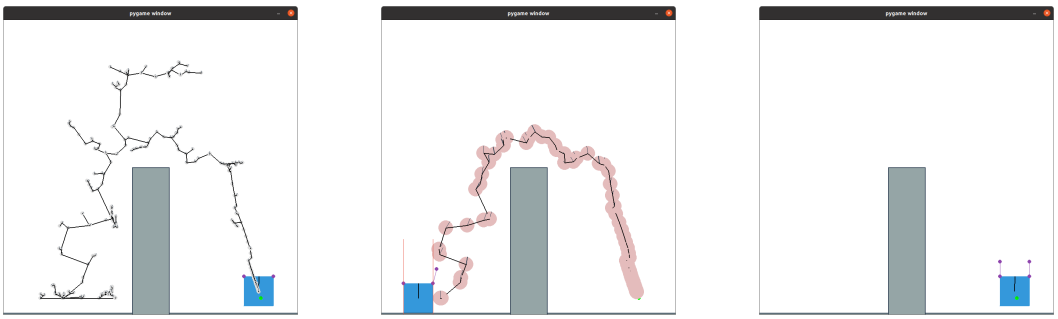
Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	5.415 s	88	27	80	0.188

Table 5.3: Average values for scenario 1 - variant 3

5.2 “Square_over_hill” Scenario

5.2.1 Variant 1

This scenario was designed to check if the drone could avoid an obstacle in its path. In this scenario, it would be impossible to arrive at the goal if we pursued it directly, without any random sampling. The results shown in Fig. 5.5 show that the drones can indeed contour an obstacle. Results for this scenario are show in Table 5.4.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.5: “Square_over_hill” variant 1

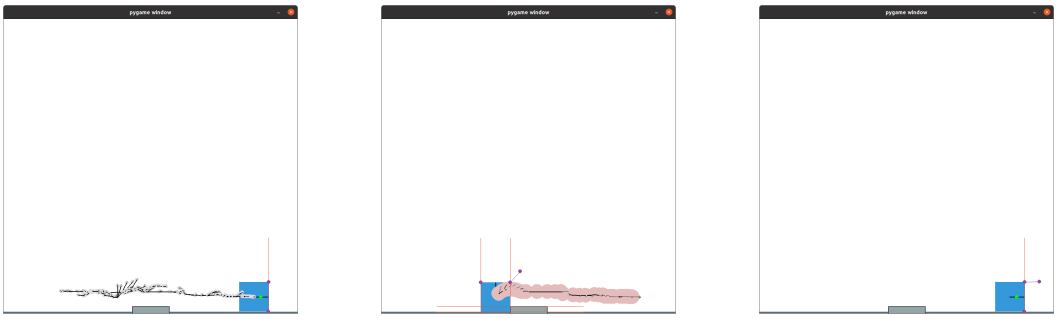
Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	88.53 s	306	62	403	0.190

Table 5.4: Average values for scenario 2 - variant 1

5.3 “Square_over_bump” Scenario

5.3.1 Variant 1

Fig. 5.6 shows a successful case when this scenario was executed. It is worth noting that the objective here was to try to flip the object using the bump, without lifting, therefore a constraint was created where at least one of the points of the object must be touching a surface. Table 5.5 shows this scenario is often unsuccessful, but it did manage to achieve its objective more than once.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.6: “Square_over_bump” variant 1

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
2/5	140.21 s	490	40	415	0.315

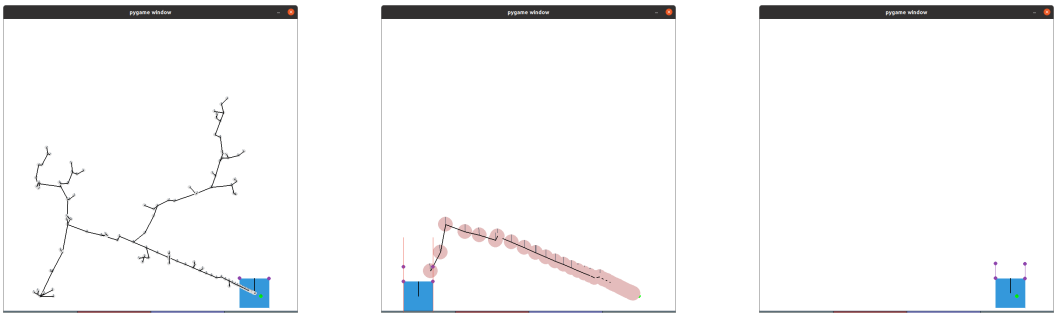
Table 5.5: Average values for scenario 3 - variant 1

5.4 “Slide_square_on_rugged_surface” Scenario

5.4.1 Variant 1

The objective to this scenario is to see how the model behaves when faced with different friction coefficients. If we allow unrestricted movement of the object, we have seen before

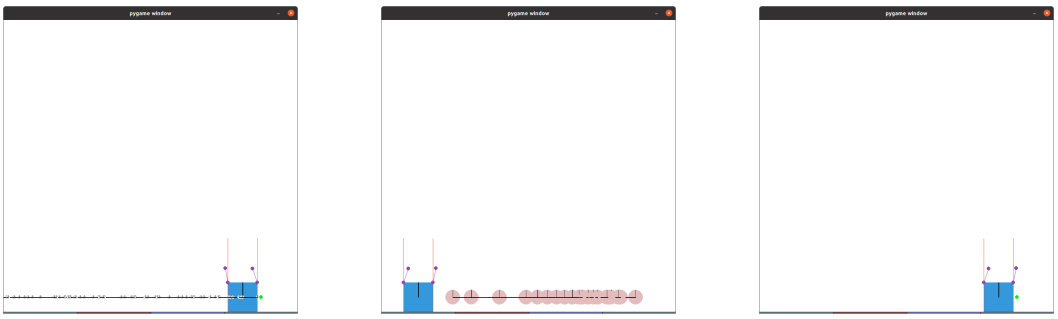
that the object will typically be lifted from the surface, which kind of defeats the purpose of this scenario, such as in Fig. 5.7.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.7: "Slide_square_on_rugged_surface" variant 1

Therefore, in order to study this scenario, we have placed the condition that the contact mode "separate" is not valid for environment contact points, which means the object will stay attached to the surface and cannot rotate, such as in Fig. 5.8. The accumulated data for this scenario is seen in Table 5.6.



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.8: "Slide_square_on_rugged_surface" variant 1 - alternate version with movement restricted to surface

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	57.33 s	57	21	96	0,135

Table 5.6: Average values for scenario 4 - variant 1

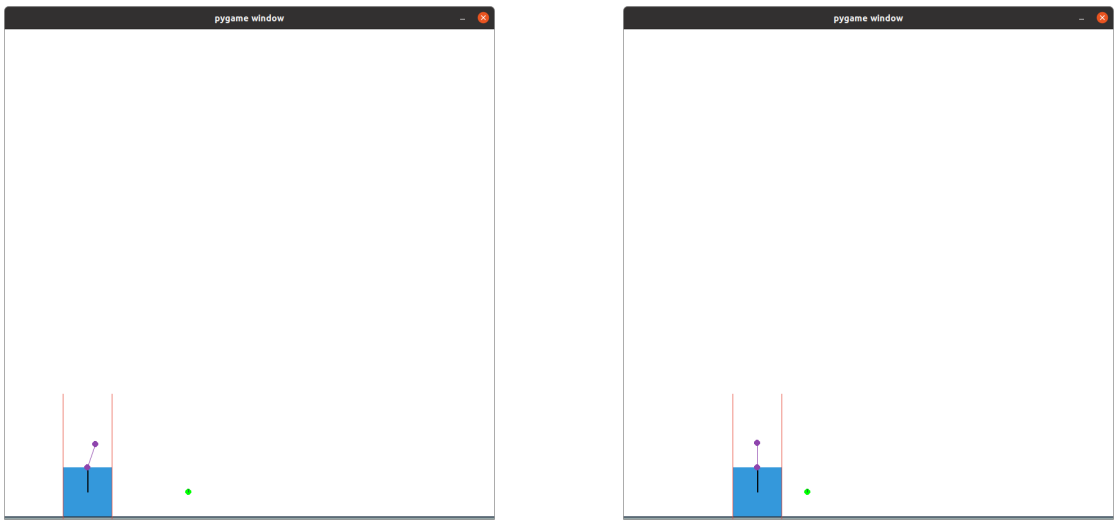
As for the behavior of the object on the surface, it was observed that the change in frictions was reflected in the drones. Taking the rightmost drone as representative, the angle of its cable with the vertical direction increases by around 3% when it is in the red

zone as opposed to its value in the blue zone. Drone thrust increased by 46% in the same case (for a massless drone). Both these values are in agreement with the expectation that the drones will have to offer more push in the horizontal direction to compensate the higher friction of the red zone as opposed to the lower friction of the blue zone.

5.5 “Square_with_single_drone” Scenario

5.5.1 Variant 1

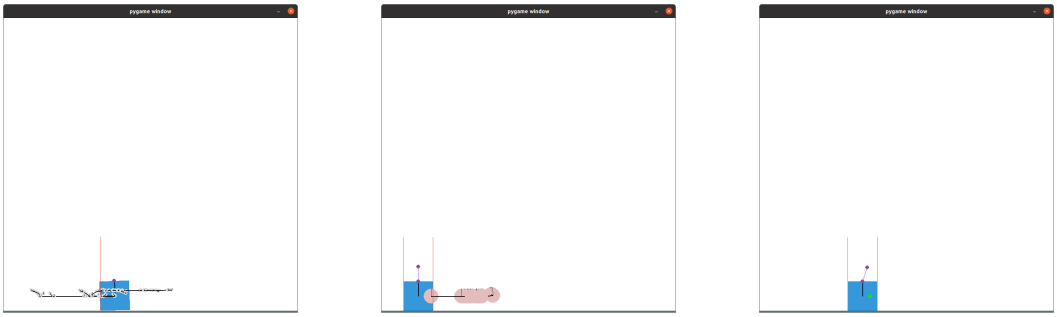
Variant 1 of this scenario offers some interesting insight into the movement of the drone, which is along the same lines as scenario 1, variant 1: namely that the algorithm does not optimize for minimal drone effort. In effect, the contact modes on the object alternate from “separate”, which entails more effort on the drone’s part, and “right-slide”, which alleviates contact on the drone by allowing the surface to counter some of the object’s weight. Fig. 5.9 shows a moment where the drone is lifting the object and a moment when it is dragging the object. The full scenario can be seen in Fig. 5.10 and its details are summarized in Table 5.7.



(a) Drone dragging the object on the table. Its angle compensates the table’s friction

(b) Drone lifting the object, which only slightly touches the table

Figure 5.9: Comparison between two different behaviors for the drone-object system, which alternate among each other during transport



(a) Extended tree (b) Goal path (c) Final object position

Figure 5.10: "Square_with_single_drone' variant 1

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	14,54 s	163	14	77	0.159

Table 5.7: Average values for scenario 5 - variant 1

5.5.2 Variant 2

Contrary to the analogous variation in the first scenario - where two drones had to rotate the object in the direction of the movement - this time the algorithm is successful, mainly because it is extremely unlikely that a single drone could lift the object from the surface in our sampling method because of the quasi-static condition (since the object has to stay static at each moment, the desired pose would need to have an angle equal to zero so that the object could be lifted). Data about this scenario is summarized in Fig. 5.11 and Table 5.8.

(a) Extended tree (b) Goal path (c) Final object position

Figure 5.11: "Square_with_single_drone' variant 2

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
4/5	6.853 s	113	22	57	0.106

Table 5.8: Average values for scenario 5 - variant 2

5.5.3 Variant 3

As shown in Table 5.9, this variant is unsuccessful. Fig. 5.12 shows that the object explores the surface, but doesn't get close enough in rotation to match the goal.

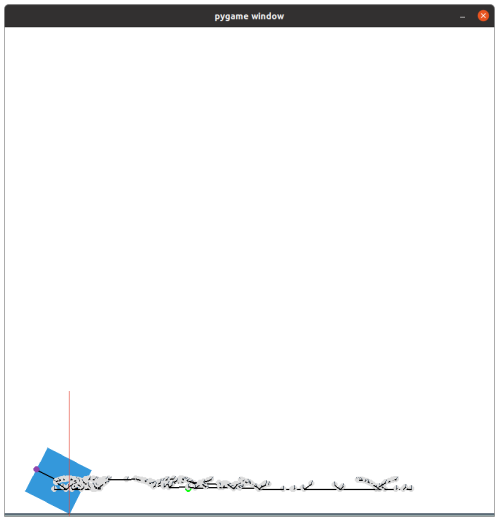


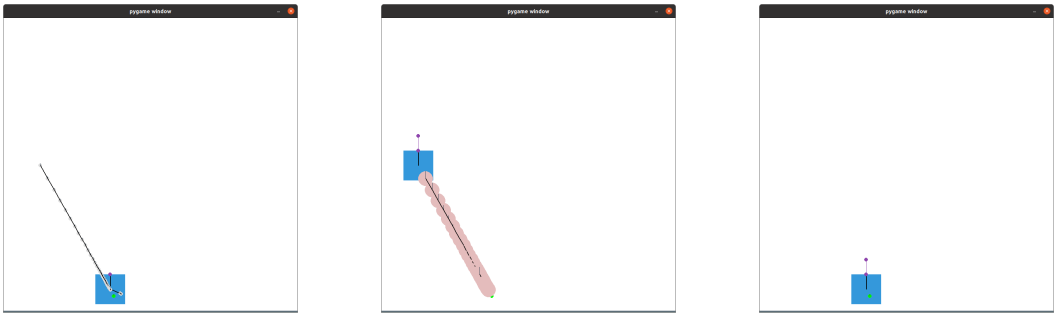
Figure 5.12: Extended tree - Scenario 5 variant 1

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
0/5	37.39 s	501	-	318	1.669

Table 5.9: Average values for scenario 5 - variant 3

5.5.4 Variant 4

The results for this variant are registered in Fig. 5.13 and Table 5.10. For reasons that will become clear in the following variant, the reason why the number of nodes in the tree is equal in average to the number of nodes in the path is because the object only moves when the goal is sampled. This is because the goal's angle is exactly zero, which enables quasi-static movement.



(a) Extended tree (b) Goal path (c) Final object position

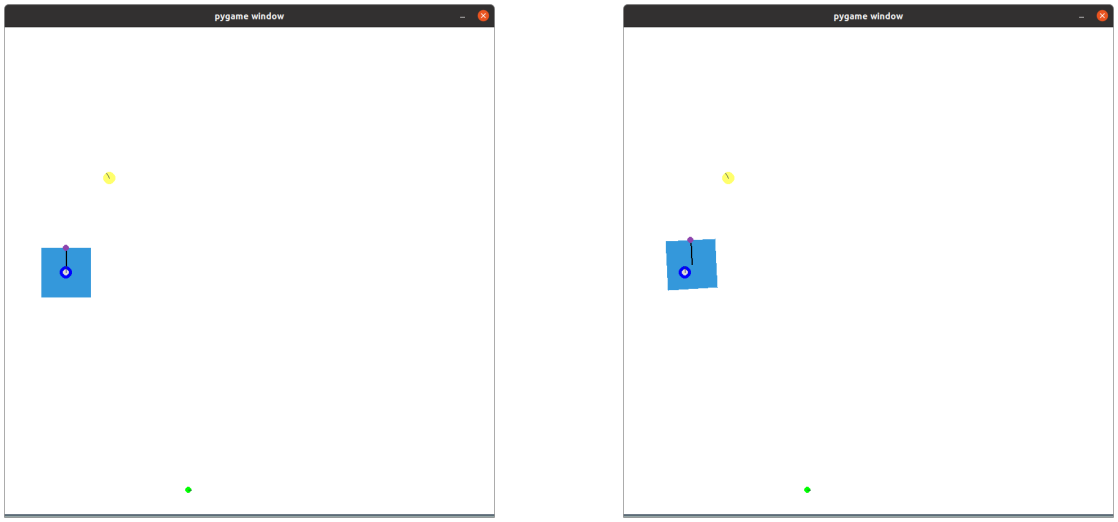
Figure 5.13: "Square_with_single_drone' variant 4

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
5/5	4.443 s	30	30	167	0.192

Table 5.10: Average values for scenario 5 - variant 4

5.5.5 Variant 5

Variant 5 is basically a combination of variant 4 and variant 2, which are both successful. We see, however, that variant 5 is unsuccessful: not only it doesn't reach the goal, it does not even add nodes to the tree, as shown in Table 5.11. The explanation for why no nodes are added can be seen in Fig. 5.14: from the starting position, the object gets a random sampled point (yellow) as destination; the static condition is checked before the movement and there are no constraints on the velocity, so it moves in translation and rotation towards the next goal. After its movement, the object now has an angle, meaning it cannot be kept statically by a single drone, therefore the new point is not added to the tree. This process is then repeated. The only way for the object to move in this case is if the sampled point has exactly zero angle, but the chance of that happening is quite low (but non-zero, since the sampling is discrete).



(a) Before movement (b) After movement

Figure 5.14: Movement in scenario 5 variant 5

Successes	Runtime	Nodes in tree	Nodes in path	Sampled Nodes	Min distance to goal
0/5	Inf s	1	-	Inf	5.565

Table 5.11: Average values for scenario 5 - variant 5

Chapter 6

Possible Improvements and Next Steps

Considering the results previously presented, some improvements to the model used are suggested so that better performance can be achieved.

6.1 Extending RRT to the maximum

The formulation presented in algorithm 1 uses small, fixed increments when calculating the object's movement. As we have seen, this results in the movement manifolds not being explored. Recall scenario 1 variation 1, for example, where the simplest solution would just be to slide the object on the table.

Based upon these considerations, one possible improvement for the model is to expand the tree in a manifold until $v_o = 0$, as exemplified in Fig. 6.1, which illustrates this idea for the case where all environment contacts are of mode "right-slide". As shown by [Cheng 2021b], applying this behavior means that we project configurations upon the movement manifold.

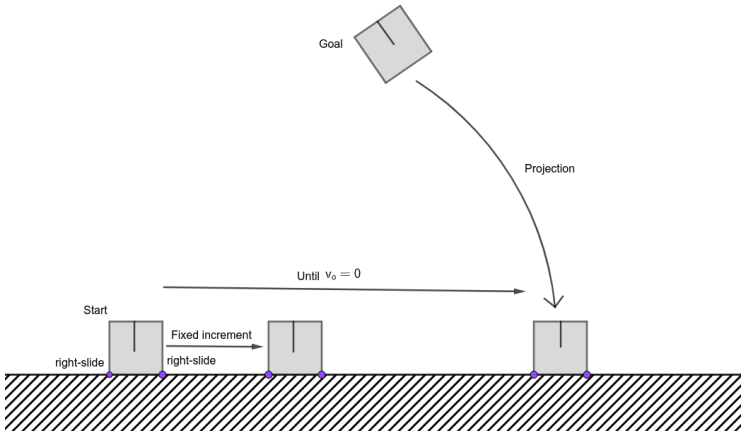
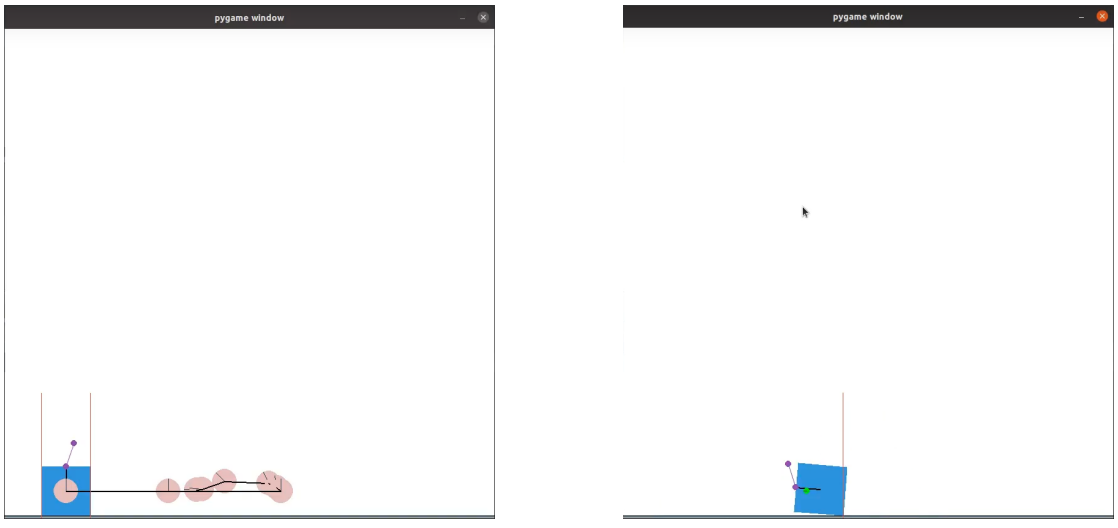


Figure 6.1: Comparison of fixed distance increment versus moving object until $v_o = 0$

Preliminary applications of this idea show that adapting the algorithm for maximum extension allows us to solve scenarios which were previously unsolved, such as scenario 5, variation 3, as shown in Fig. 6.2. The method also results in fewer nodes in the solution path. On the other hand, implementation of this modification should take into account how to verify collision for these long distance, so as to ensure that the object does not trespass obstacles.



(a) Goal path (b) Final object position

Figure 6.2: "Square_with_single_drone' variant 3 - Alternate version with RRT extending until $v_o = 0$

6.2 Simulations in the ROS/Gazebo environment

As mentioned before, during these simulations the object was also actuated, for the reason that Pymunk doesn't work properly when the constraints are moving. A more realistic simulation environment can be found in the Gazebo software. In it, we can simulate a drone controlled by a PX4 flight stack, which can be used to manipulate an object. The drone's trajectory can be commanded via ROS [Joseph 2018]. Such an environment is shown in Fig. 6.3. We expect to have this simulation implemented for the single drone scenario by the end of the internship.

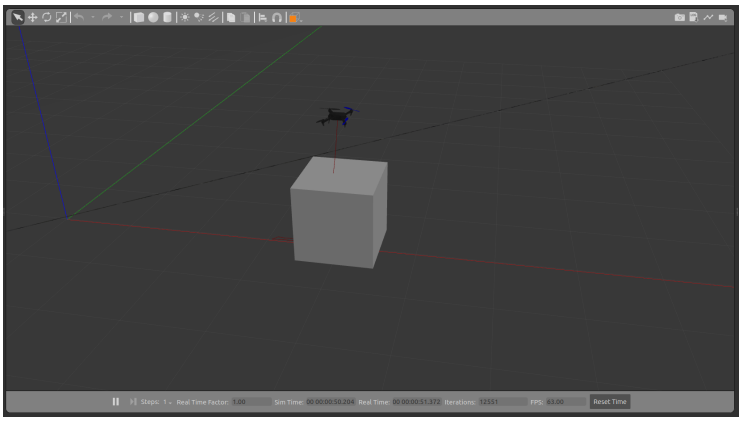


Figure 6.3: Gazebo simulation of scenario 5

6.3 Weighted planning

We remarked in chapter 5 that the path chosen does not always minimize the effort required by the drones. One possible way to do this is to create a cost function $C(\vec{T})$ that is a function on the drones' thrust, which could be attached to the edges. Since the path to the goal is calculated via Dijkstra's algorithm on top of the expanded tree, these costs would enable us to choose the path thar requires the least effort on the drones' part.

6.4 Smooth drone movement

The discretization of the drones' paths give rise to two problems: the first is how to make sure the movement of the drones from one position to another results in the desired object configuration. Although we used the notation $q(\vec{d})$ previously, this is an abuse: as illustrated in Fig. 6.4, the same drone configuration can result in different object configurations, which means the path to get to a destination is also important, and was

largely ignored during discretization.

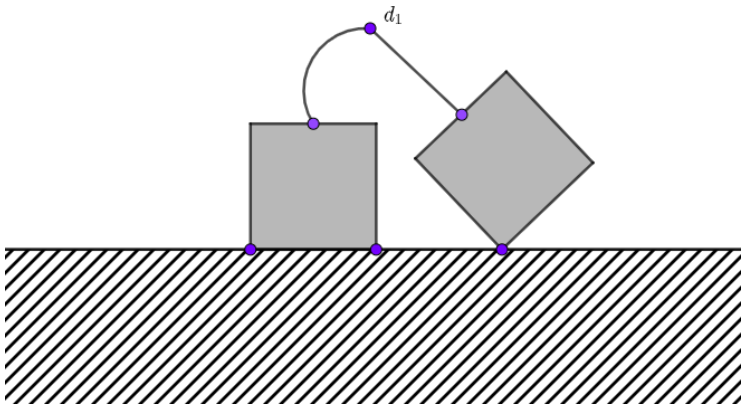


Figure 6.4: Two different object configurations that can arise from the same drone configuration

The second problem that needs to be dealt with is that the path found is not always "smooth": for example, in Fig. 6.5, the two drones alternate which one is "strained" and which one is "loose" between one path point and the other. This situation is evidently not realistic.

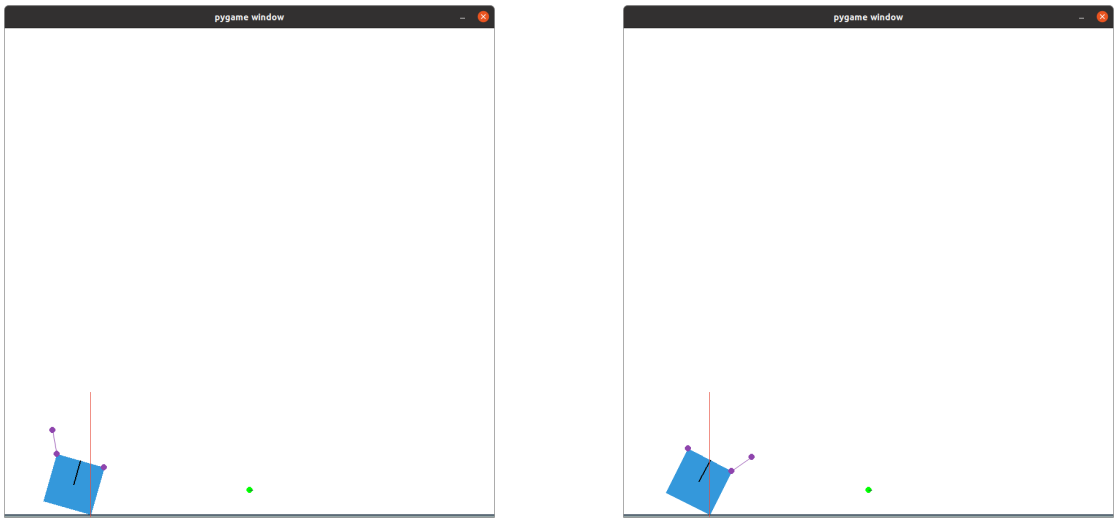


Figure 6.5: Drones alternating contact mode between one step and the other

6.5 Quasi-dynamic formulation

Scenario 5 (especially variant 5) showed that the quasi-static condition might be too restrictive to solve certain situations. Moreover, this restriction is unrealistic insofar as there is no link between velocity and force except that both should be valid for the movement to happen. One possible modification that should be studied is to substitute (3.19) for a quasi-dynamic formulation [Cheng 2021a]:

$$M\dot{v}_o = G(c)\vec{\lambda} + \vec{F}_e^B(q) \tag{6.1}$$

In this formulation, we should use a small time increment so that accelerations are not too significant. Under this relaxed assumption, we could get solutions for some of the scenarios which were previously unsolved and also have an explicit link between force and velocity.

Chapter 7

Conclusion

All in all, this internship was a unique opportunity for me to know firsthand how research is conducted at a top tier laboratory. I am honored to have been part of LAAS's quest to make life more comfortable through technology, and I hope to have contributed, even if only a bit, to the lab's projects on UAV development.

As for the method exposed in this work, it is noticeable that it has potential, even taking into account its shortcomings, for it offers some new insight into how to deal with cooperative manipulation and environment interaction in the context of UAV object manipulation. This method surely deserves further investigation so that it can be exploited to its full potential.

Bibliography

[Berenson 2009] Dmitry Berenson, Siddhartha S Srinivasa, Dave Ferguson et James J Kuffner. *Manipulation planning on constraint manifolds*. In 2009 IEEE international conference on robotics and automation, pages 625–632. IEEE, 2009.

[Blomqvist 2022] Victor Blomqvist. *Pymunk repository*. `https://github.com/viblo/pymunk`, 2022. Accessed: 2022-08.

[Byrne 2001] Richard W Byrne, Jennifer M Byrne*et al.* *Manual dexterity in the gorilla: bimanual and digit role differentiation in a natural task*. Animal Cognition, vol. 4, no. 3, pages 347–361, 2001.

[Caron 2022] Stephane Caron. *Qpsolvers repository*. `https://github.com/stephane-caron/qpsolvers`, 2022. Accessed: 2022-08.

[Cheng 2021a] Xianyi Cheng, Eric Huang, Yifan Hou et Matthew T Mason. *Contact Mode Guided Motion Planning for Quasidynamic Dexterous Manipulation in 3D*. arXiv preprint arXiv:2105.14431, 2021.

[Cheng 2021b] Xianyi Cheng, Eric Huang, Yifan Hou et Matthew T Mason. *Contact mode guided sampling-based planning for quasistatic dexterous manipulation in 2d*. In 2021 IEEE International Conference on Robotics and Automation (ICRA), pages 6520–6526. IEEE, 2021.

[Eade 2013] Ethan Eade. *Lie groups for 2d and 3d transformations*. URL http://ethaneade. com/lie. pdf, revised Dec, vol. 117, page 118, 2013.

[Fink 2011] Jonathan Fink, Nathan Michael, Soonkyum Kim et Vijay Kumar. *Planning and control for cooperative manipulation and transportation with aerial robots*. The International Journal of Robotics Research, vol. 30, no. 3, pages 324–334, 2011.

[Joseph 2018] Lentin Joseph et Jonathan Cacace. Mastering ros for robotics programming: Design, build, and simulate complex robots using the robot operating system. Packt Publishing Ltd, 2018.

[LAAS 2022] LAAS. *LAAS website*. `https://www.laas.fr/public/en`, 2022. Accessed: 2022-08.

[Mason 2001] Matthew T Mason. Mechanics of robotic manipulation. MIT press, 2001.

[Murray 2017] Richard M Murray, Zexiang Li et S Shankar Sastry. A mathematical introduction to robotic manipulation. CRC press, 2017.

[Ollero 2021] Anibal Ollero, Marco Tognon, Alejandro Suarez, Dongjun Lee et Antonio Franchi. *Past, present, and future of aerial robotic manipulators*. IEEE Transactions on Robotics, vol. 38, no. 1, pages 626–645, 2021.

[Sanalitro 2020] Dario Sanalitro, Heitor J Savino, Marco Tognon, Juan Cortés et Antonio Franchi. *Full-pose manipulation control of a cable-suspended load with multiple uavs under uncertainties*. IEEE Robotics and Automation Letters, vol. 5, no. 2, pages 2185–2191, 2020.

[Shinners 2022] Pete Shinners*et al.* *PyGame*. `http://pygame.org/`, 2022. Accessed: 2022-08.

[Sreenath 2013] Koushil Sreenath, Nathan Michael et Vijay Kumar. *Trajectory generation and control of a quadrotor with a cable-suspended load-a differentially-flat hybrid system*. In 2013 IEEE International Conference on Robotics and Automation, pages 4888–4895. IEEE, 2013.