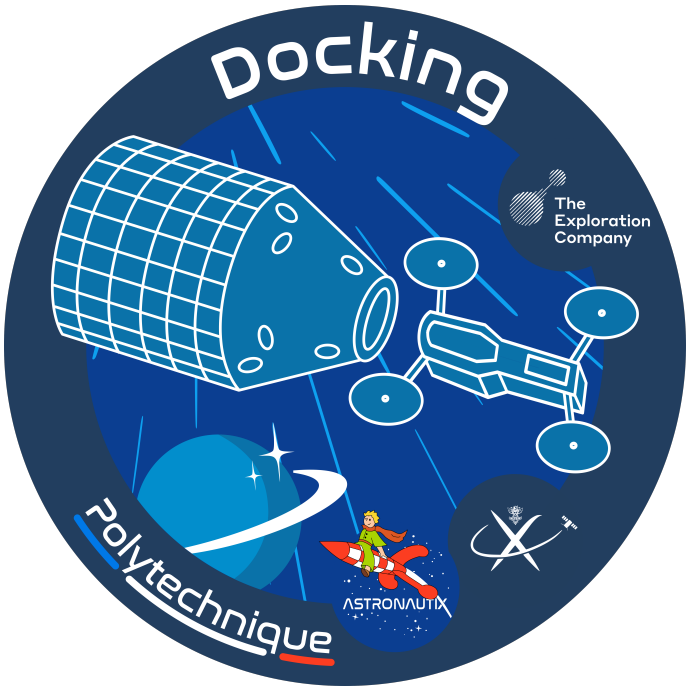
Collective Scientific Project : Final report

VISUAL-BASED NAVIGATION FOR DOCKING



Coordinator : Guilhem GALLOT & Luca BUCCIANTINI
Tutor : Antoine PALLOIS

April 2024

Matis ESCOTS, Florentin HEBRARD, Perrine PORCHER, Erwin POUSSI, Grégoire THOMAZEAU

ÉCOLE POLYTECHNIQUE | INSTITUT POLYTECHNIQUE DE PARIS

ACKNOWLEDGEMENTS

First and foremost, we would like to express our sincere gratitude to our project tutor, Antoine Pallois, for his continuous assistance and invaluable advice. His guidance and frequent supervision were instrumental throughout the duration of this project.

Secondly, we extend our heartfelt thanks to the Space Center of the Ecole Polytechnique (CSEP), particularly its president, Lucas Buccatiani, for their sponsorship and provision of the drone used in our experiments. We also wish to acknowledge the Fabrication Laboratory and its team for their training and generous access to 3D printing machines, which facilitated the simulation of docking and the approaching vehicle during the docking process.

Last but not least, we would like to express our gratitude to the coordinators of the Physics Department, including Alistair Rowe, Pascal Chabert, Guilhem Gallot, and François Ozanam, for their unwavering support and guidance since the inception of the project.

EXECUTIVE SUMMARY

Docking is a crucial procedure in space exploration, involving the precise alignment and connection of two vehicles. This maneuver, commonly performed between spacecraft and space stations, demands extreme precision to prevent damage as the vehicles interlock. To ensure this precision and automate the process, a variety of algorithmic methods are employed.

In our project, we focus on visual-based docking which consists to use a visual feedback, to get the relative position of the approaching vehicle with the respect to the target vehicle. For that matter, we propose a groundwork of the algorithms that would be applied to the Nyx spacecraft, for its docking on the international space station (ISS). Our prototype is applied on a DJI tello drone to facilitate the tests as in space tests are challenging. Thus, we started by understanding the drone commands, then we implemented navigation algorithms to have a precise relative position, and we ended up by implementing guidance algorithms.

To locate the target, we used aruco Qr codes. The drone's sensors, including an HD camera, visual positioning system (VPS), IMU sensor, collision detection system, and barometer, provide essential data for precise navigation. However, challenges such as detecting Aruco codes at varying distances and noise interference in IMU data necessitated problem-solving approaches. We had to limit the research distance to 7 meters to maintain focus on the ArUco code the beginning research distance to 7m to have a good focus on the aruco code. Additionally, we controlled the drone through its speed to limit noise interference.

Initiating the camera calibration process was imperative to ensure the accuracy of visual data. By capturing reference images of a known object, such as a chessboard, we determined the intrinsic parameters of the drone's camera like focal length and optical center coordinates to minimize distortions caused by the camera lens and to establish a precise correlation between real-world 3D points and their 2D projections in captured images.

The ArUco codes we used as target are detected thanks to OpenCv functions in captured images, allowing to obtain their corner coordinates and associated identifiers. This detection process enables precise localization of the docking targets, laying the foundation for navigation.

For our final guidance algorithm, we principally use a PID controller to enable precision attitude control during the docking, but we also combined with different command depending on the drone state. This algorithm encompasses distinct phases tailored to specific states of the drone and adaptively adjusts guidance commands accordingly.

Phase Lost : Designed for situations where the drone loses sight of the target, this phase systematically explores the surroundings to rediscover the target. Divided into stages, it includes waiting, rotation, and exploration maneuvers to locate the target.

Phase Getting Closer: Initiates when the drone is at a distance from the target, focusing on aligning the drone and approaching the target smoothly. It consists of three subphases each with specific control strategies to ensure accurate navigation towards the target.

Phase Docking : Activated when the drone is in close proximity to the target, this phase prioritizes precision and slow speed to facilitate successful docking. Utilizing PD control on multiple axes, it ensures stable alignment with the target.

Phase Landing : Executed when the drone is in the vicinity of the target with minimal positional error, this phase involves simple backward movement to avoid collisions with surrounding obstacles before initiating the landing process.

CONTENTS

LIST OF FIGURES

1
CONTEXT

Our project focuses on the visually guided docking of the Nyx orbital spacecraft, specifically engineered for orbital refueling operations. Created by The Exploration Company and chosen by Axiom to provision its upcoming space stations, Nyx not only boasts the capability to transport cargo, but also envisions the prospect of transporting astronauts in the future. To establish the groundwork for Nyx's docking navigation algorithms, we propose refining the navigation algorithms of a quadcopter drone and leveraging its onboard camera to precisely maneuver and approach a target painted on a fixed object.

Docking in space refers to a highly controlled operation involving the attachment and securing of two spacecraft in orbit or during spaceflight. This maneuver requires precise coordination of navigation and communication systems to ensure a safe and stable connection between the vehicles. NASA conducted the first orbital rendez-vous in 1966 with Gemini 8; however, it wasn't automated until 1977. Since then algorithms have been developed to enhance precision and autonomy.

Several technologies are used for space navigation. The first method is absolute navigation, typically achieved through a Kalman filter that merges data from various sensors such as an inertial measurement unit, a GNSS receiver, and a star tracker. However, these technologies are not precise enough for the final docking phase, where accuracy on the order of centimeters is required.

For this reason, relative navigation is widely favored for the final docking phase, while absolute navigation allows approaching the initial target. Relative navigation involves determining the spacecraft's position relative to a reference point, usually the docking target. To obtain

this positional information, image recognition is used, forming the basis of visual navigation.

All collected information will be integrated into the GNC (Guidance, Navigation, and Control) loop of the drone. This loop, essential to the operation, determines the trajectory, assesses the vehicle's position, and corrects its course. In the context of our project, implementing a GNC loop will promote a docking approach that is both precise and secure. In synergy with data from visual navigation, this loop will ensure an optimization of the process by considering dynamics and potential disturbances.

2
CONVENTION AND NOTATION

The code we write must respect the conventions and notations defined by The Exploration Company to be understandable by everyone and coherent, these rules are stated below.

2.1 The frame

The two frames presented below are the frame that will be used throughout the whole program.

2.2 Drone frame

The body frame will be used to represent the drone frame, its characteristics are as follows :

- His origin is at the drone's camera lens center
- (Ox_b) corresponds to the drone axis of roll and points towards the front
- (Oy_b) corresponds to the drone axis of pitch and points to the right
- (Oz_b) corresponds to the drone axis of yaw and points to the bottom

The drone frame is indeed a right-handed frame.

2.3 The target frame

The target is intrinsically oriented which allows us to define a frame as such :

- Its origin is at the target top left corner
- (Ox_t) points from the target plane towards its back
- (Oy_t) points to the right when looking at the target
- (Oz_t) points towards the bottom of the target

The target frame is indeed a right-handed frame, and for our purpose we will consider the target motionless. The first figure below represents the target frame, while the second one illustrates its orientation on the ArUco QR code that we use.

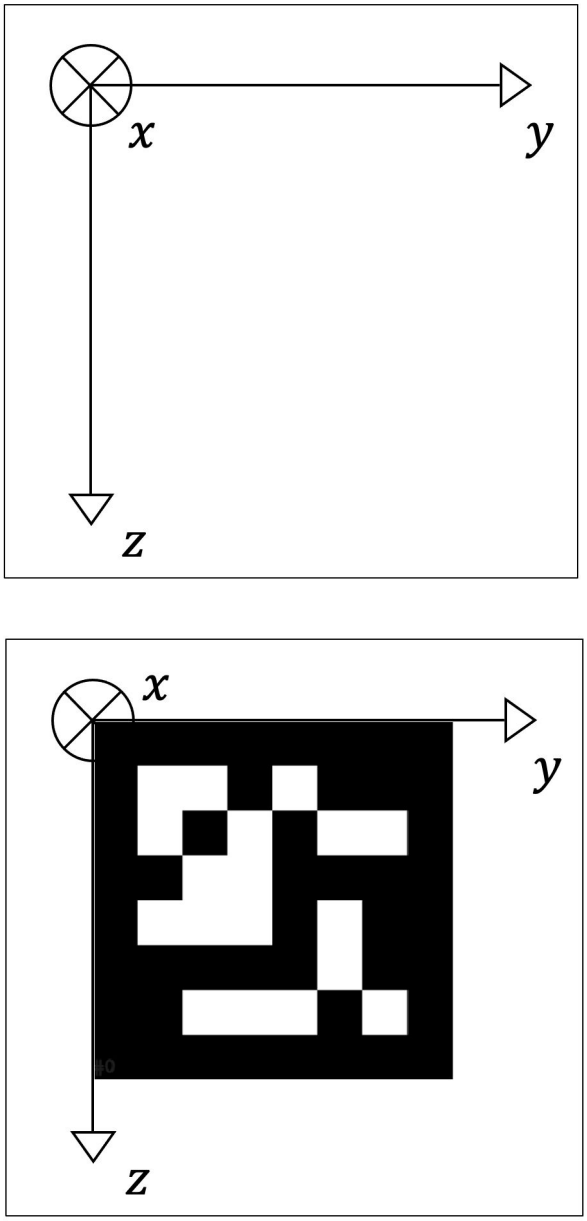


Figure 1: Target frame

3 THE DRONE

3.1 Sensors

First of all, the Tello drone has a HD camera which will allow us to state where the drone is compared to an Aruco code. The Tello drone uses a visual positioning system. This VPS is formed of infrared sensors and a camera pointing down to allow a static flight and avoid drift (this can be a problem over a homogenous floor or in a dark environment, we take into account to solve our problem).

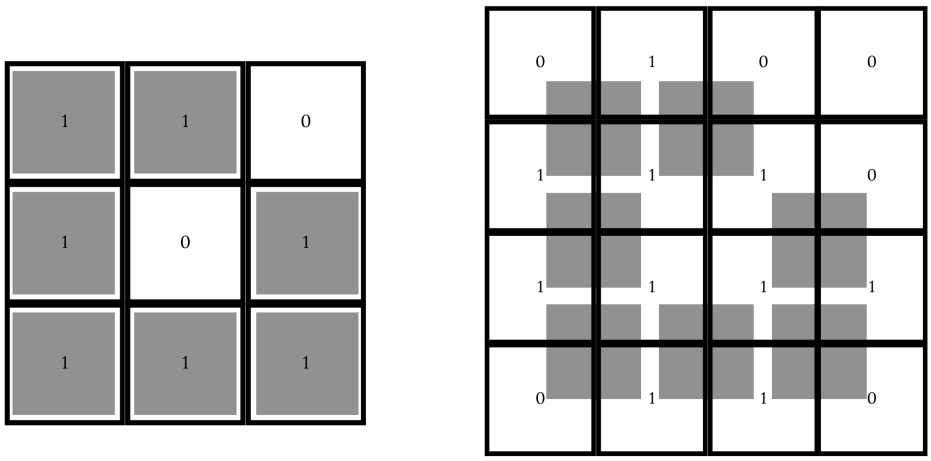
It also has an antenna for communication (works in wifi)

There is also an IMU sensor (inertial measurement unit) that we can use.

We can also quote a collision detection system which is no more than a torque sensor, and a barometer.

Thoughts and conclusions about sensors :

The camera has a field of view of approximately 82° and captures 1280 pixels on it. A linearisation makes us conclude that to detect an Aruco code of 7 centimeters (it's a 7 by 7 binary grid) we must be at a maximum of 7.5 meters. But, a bit like for Nyquist-Shannon, our resolution must be finer than the pattern we search :



(a) code can be read (b) Code isn't correctly readable anymore

Figure 2: Binary grids generated by the drone's sensors for Aruco codes detection

This makes us conclude that this distance of 7 meters is overestimated and is in fact verified in reality. It also clearly showed that we wouldn't be able to navigate only by focusing on the Aruco code. Our navigation algorithm had to take into account this phase where the code is not detected. We discuss later the idea of having different sizes of Aruco code and switch to the smallest during the navigation to be more precise.

We first thought that the IMU sensor could help us to know our position. But the position is the second primitive of the acceleration (the data we have), and the acceleration has a noise over it; the IMU present on the drone does not offer sufficient accuracy to propagate the drone's state inertially. Other sources of measurements would theoretically enable to correct the drift generated by the IMU's poor performance. However, there were no useful measurements sources

(no GNSS since indoors, no LiDAR, ...). It would only be the sensor against the theory and wouldn't have solved the problem of drift (because it can't predict exterior parameters like the movement of the environment air). Nevertheless it can give us an instantaneous speed:
Let x be the real quantity and $x*$ the quantity detected.
Let ϵ be a noise and let us consider an initial null velocity

$$\begin{aligned} v(t) &= \int_{t_0}^{t} a(s)ds + v(t_0) \\ &\approx \int_{t_0}^{t} a(s)ds \\ &\approx \int_{t_0}^{t} a^*(s)ds + \int_{t_0}^{t} \epsilon(s)ds \\ &\approx \int_{t_0}^{t} a^*(s)ds \quad \texttt{if the acceleration is enough} \end{aligned}$$

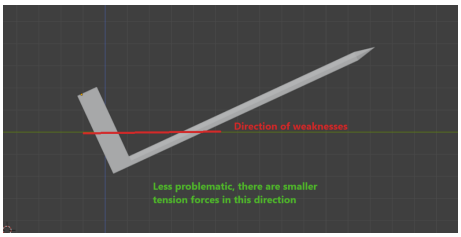
We will be focusing on the visual positioning system and the IMU sensor.

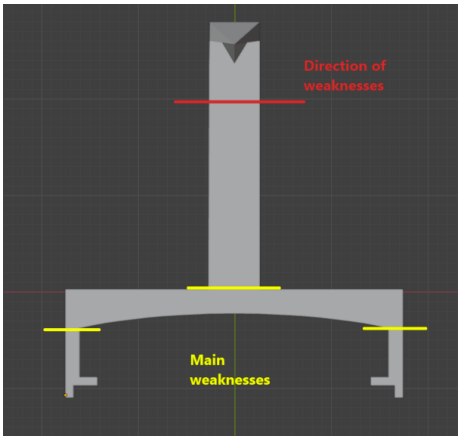
3.2 3D Printing

To mimic the docking in reality, we wanted to create docking parts that could lock in as it would on the real spacecraft. We modeled in fusion 360 both of the parts, by measuring and scanning the drone to create a piece that could clip to it.

The many advantages of 3D printing allowing us to create lightweight and precise pieces was unfortunately counterbalanced by the relative fragility of the latter. To create as solid pieces as possible, we did not print the pieces with the basis parallel to the support, because then the weakest direction was also parallel to it, while being the stress direction.

We then decided to print it at an angle, allowing us to put the weakest direction in one that is not really affected by stress, as shown below.





We also printed a physical target, consisting of an inverted cone, as shown below.

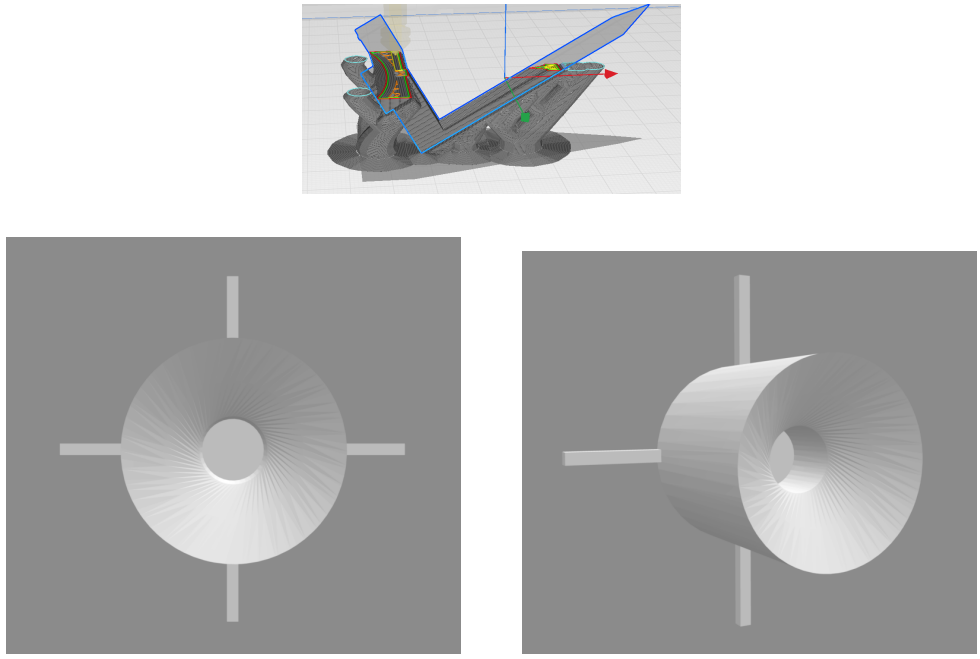


Figure 3: 3D printed pieces simulating docking conditions

However, even with all these precautions, the male part, attached to the drone, was never solid enough to resist to the drone inaccuracies when docking, resulting in a frequent loss of the piece. We subsequently decided to abandon physical docking, and limit ourselves to an immaterial docking, defined after.

3.3 Drone control

We selected the Tello Drone manufactured by DJI and Ryze Robotics because it comes equipped with a Python library called the Tello library, which grants us full control over the drone's functionality.

3.3.1 • Connection to the drone

The program we have written runs on a computer, and data is transmitted via WiFi. This approach allows us to leverage the processing power of a real computer, as opposed to relying on the drone's relatively limited internal processor. This is particularly relevant as the Nyx capsule is not constrained by the size of its internal computer (not at this scale at least).

To establish a connection with the drone, we simply need to create an instance of the drone class and then inform Python that this instance is linked to the actual drone using the `.connect()` method:

```
tello = Tello()
tello.connect()
```

The main problem with this type of connection is that we are not able to execute different programs at the same time. Unable to bypass this restriction, we built our code as one program that did every sequences of the GNC loop. While sufficient for our project, it implies a relatively low reliably as there is no redundancy, which cannot be the case when in space.

3.3.2 • Real time control of the drone

Once the connection is established, we have the ability to send commands to the drone and receive data from it. The library provides a variety of functions, including:

Commands to control the drone's movement:

```
tello.takeoff()
tello.land()
tello.forward(100)
tello.flip("l")
```

To retrieve sensor data:

```
tello.get_frame_read()
tello.get_acceleration_x()
tello.get_battery()
tello.get_distance_tof() #get the distance from the takeoff point
tello.get_height() #get the height relative to the floor
```

This data may help us but we will mainly gather data from the camera to execute a visual navigation. The height is obtained thanks to the infrared sensors under the drone.

To obtain video feed, we utilize the `get_frame_read()` function, which returns the current frame captured by the camera.

This library presents two methods of controlling the drone:

Positional Control Using the `move_direction` functions within the library, we can command the drone to move in specific directions (up, down, left, right, forward, backward) for distances ranging from 0.2m to 5m. However, we opt not to use this method due to its lack of precision (no movement less than 20 centimeters) and its disruption of other concurrent activities, including camera queries, throughout the maneuver. Additionally, these commands cannot be aborted once initiated.

Speed Control We prefer to use the `send_rc_control` function provided by the Tello library for controlling the drone. This method allows us to control the drone in all three spatial directions, with additional control over the yaw angle, enabling us to orient the drone as desired. This function can be invoked every millisecond, and each argument can vary from -100 to 100, providing precise control over the drone's movements. Furthermore, we have developed a program to record all the data we receive and the commands we send. This ensures that we can review and analyze any issues that may arise.

4 NAVIGATION

4.1 Camera Calibration

Initiating the camera calibration process is crucial for leveraging a camera as an accurate visual sensor. To mitigate distortions stemming from the camera lens and imperfections, we capture reference images of an object, usually a chessboard. These reference images aid in computing intrinsic parameters such as focal depth, optical center coordinates, and skew.. The camera calibration procedure encompasses acquiring internal parameters (e.g., focal length and optical center) and external parameters (rotation and translation), establishing an exact correlation between real-world 3D points and their 2D projections in captured images.

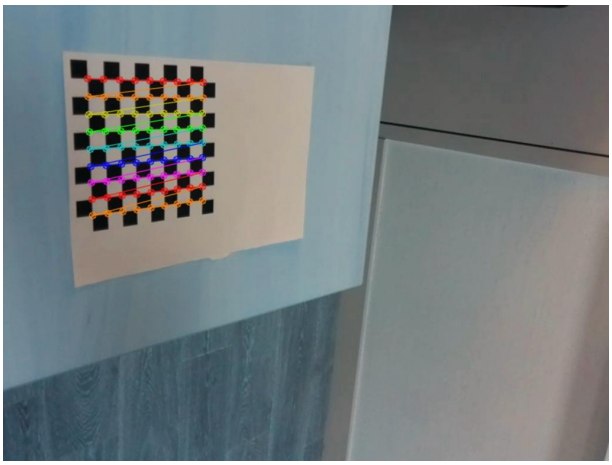


Figure 4: Example of a chessboard during the camera calibration

The objective of the calibration process is to ascertain the 3×3 intrinsic matrix K:

$$\begin{bmatrix} f_x & \gamma & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

- f_x, f_y are the x and y focal lengths, they are usually the same, or very close
- c_x, c_y are the x and y coordinates of the optical center in the image plane, they are usually near the center of the image
- γ is the skew between the axes

In OpenCV, the camera intrinsic matrix lacks the skew parameter, considered null, so the matrix is of the form :

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Once intrinsic and extrinsic parameters are obtained, the camera is considered calibrated.

The matrix K is then used is the equation relating 3D point (X_w, Y_w, Z_w) in world coordinates to its projection (u, v) in the image coordinates :

$$\begin{bmatrix} u' \\ v' \\ z' \end{bmatrix} = \mathbf{P} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \texttt{with} \begin{cases} u &= \frac{u'}{w'} \\ v &= \frac{v'}{w'} \end{cases}$$

Where, $\mathbf{P}$ is the 3×4 Projection matrix :

$$\mathbf{P} = \overbrace{\mathbf{K}}^{\text{Intrinsic Matrix}} \times \overbrace{[\mathbf{R} \mid \mathbf{t}]}^{\text{Extrinsic Matrix}}$$

And $([\mathbf{R} \mid \mathbf{t}])$ is the combination of the 3×3 rotation matrix $\mathbf{R}$ and the 3×1 translation $\mathbf{t}$ vector.

4.2 Choice of docking target

As indicated in the detailed proposal, we use the relative navigation method for the final phase of docking (as opposed to absolute navigation, which is used to approach the target during an initial phase). Relative navigation involves determining the drone's position relative to a reference point, which in our case is the docking target.

Visual navigation is made thanks to image recognition via the drone's camera. The drone's position information can be acquired using this technique. We have settled on an ArUco code as the type of docking target read by the drone's camera via the OpenCV library. For the camera to recognize the target, the recognition algorithm must first specify which ArUco dictionary it comes from. The ArUco dictionary is a predefined collection of specific visual patterns designed to be easily detected and identified by computer vision algorithms.

Several ArUco dictionaries are available in OpenCV, and each is characterized by the number of bits per side of the marker and the total number of markers in the dictionary. For example, the `cv2.aruco.DICT_4X4_250` dictionary we use indicates that it is a 4x4 dictionary (16 bits per marker) containing up to 250 different markers. We chose this dictionary because it meets our specific requirements in terms of resolution, number of markers and detection distance.

4.3 Detection of docking target

We then detect the ArUco code using this native OpenCV function:
`cv2.aruco.detectMarkers(img, arucoDict)`
This function takes as parameters the image img obtained by the drone and the ArUco arucoDict dictionary to which our target belongs. The outputs are:

- `corners`: a list of NumPy arrays where each array corresponds to the detected corners of an ArUco marker, represented by a set of coordinates (x, y) in the image
- `ids`: a list of identifiers associated with each detected ArUco marker
- `rejectedImgPoints`: a list of NumPy arrays where each array corresponds to the corners that were rejected during the detection process (false positives or failure to meet certain detection conditions)

To check the quality of detection, we can plot the targets detected on the image using the function: `cv2.aruco.drawDetectedMarkers(img, corners, ids)`
This function takes the image img, corners and ids as parameters and draws rectangles around the detected markers, adding the corresponding identifiers.

At this point, we know which ArUco markers are seen by the drone's camera, and for each of them we know the coordinates of their four corners on the image of the camera.

4.4 Determining the relative position of the drone

Finally, having determined the coordinates of the docking target on the image we have acquired, we need to determine the drone's position relative to it in the real world. To do this, we use the function:
`cv2.aruco.estimatePoseSingleMarkers(corners, aruco_square_size, camera_matrix, distortion_coefficients)`
This function takes as input:

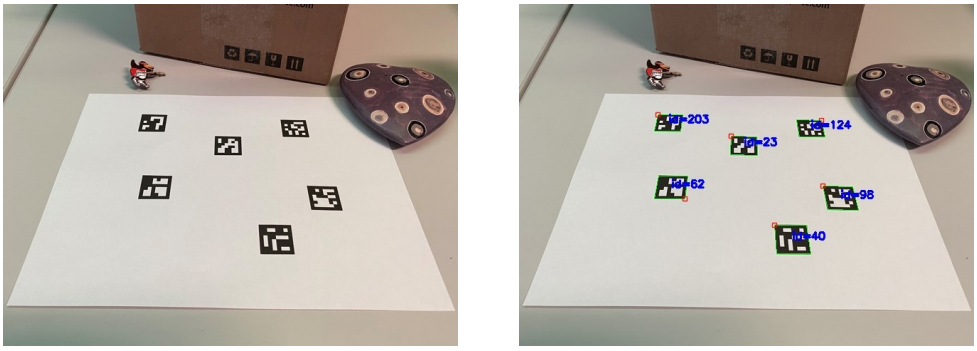
- `corners`: target coordinates (see sub-section "Detection of docking target")
- `aruco_square_size`: the side length of an ArUco marker as printed in reality (in meters)
- `camera_matrix`: intrinsic camera matrix (obtained by calibrating the camera)
- `distortion_coefficients`): camera distortion coefficients (obtained during camera calibration). This function returns:
- `rvecs`: the output rotational vector: this represents the orientation of the marker in terms of rotation around each axis (yaw, pitch, roll)
- `tvecs`: the output translational vector: represents the marker's 3D position relative to the camera (x, y, z)

This function uses the formula :

$$\begin{bmatrix} u' \\ v' \\ z' \end{bmatrix} = \mathbf{P} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \texttt{with} \begin{cases} u &= \frac{u'}{w'} \\ v &= \frac{v'}{w'} \end{cases}$$

P is known thanks to the camera calibration that has been done, and u and v are the two coordinates of each corner of the ArUco marker that have been obtained in the previous step.

`cv2.aruco.estimatePoseSingleMarkers` computes the real world position of each corner of the target, relative to the camera, and return the position of the upper left corner. It also computes the normal of the target in the form of a three dimension vector using the Rodrigues formula. Thereby we obtain the position of the target relative to the drone and the orientation of the target relative to the drone, which enables us to have the position and rotation of the drone relative to the target.



(a) Examples of ArUco codes

(b) Examples of ArUco codes correctly identified

Figure 5: ArUco codes

4.5 Dealing with outlier values from the target orientation

Using the drone, however, we noticed that the orientation angle of the target varied significantly and was relatively imprecise. The issue might stem from the method used to calculate the orientation of the target. Indeed, it relies on the position of the four corners of the target, each of which has uncertainty associated with the quality of the camera image and the detection of the corner positions using the `cv2.aruco.detectMarker` function. The precision of each corner's position is sufficient to obtain good accuracy for the drone's position, but comparing the positions of each corner does not yield a very precise orientation angle for the target. Without any other means of obtaining the target angle with the drone's camera, we opted to average the target's inclination over several images.

After numerous tests, we noticed that we obtained an aberrant value approximately every 10 images. Therefore, at each moment, we take the last 10 angles from the last 10 images. We discard the 2 extreme values, leaving us with 8 out of the 10 values, and then take their average to obtain the angle at that moment. With 25 images per second, this method introduces inertia of less than half a second, which does not pose a problem for subsequent guidance as the objective is to achieve precise docking. In space, precision is prioritized over speed.

4.6 Using two ArUco markers

The size of the ArUco marker must allow the drone to detect the marker from a distance and be able to fully visualize the marker up close, at 20 cm from the target, which is the distance we want to consider the drone as docked. These two conditions cannot be met by a single ArUco marker. Therefore, we decided to use 2 markers, one large enough for the drone to detect from "far away," meaning at least 4 meters, and one smaller. The position of the large target relative to the smaller one is known. If the drone detects only the larger marker, we calculate the position of the top-left corner of the small marker relative to the large marker, and then we calculate the position of the drone. If only the small ArUco marker is detected, we use the position of the top-left corner of this marker. If both are detected, we only base our calculations on the smaller ArUco marker.

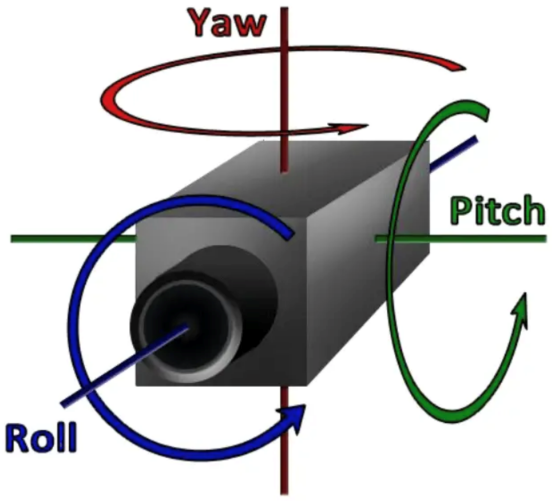
5 GUIDANCE

5.1 Drone controls

We implemented the guidance algorithm, starting from the most basic commands, to a preliminary algorithm achieved in January, and finally we developed our final guidance algorithm, whose characteristics will be described later.

The basics of relative guidance is relatively straightforward and can be divided into four primary axes: the forward and backward movement of the drone, the left and right movement, the up and down movement and the yaw movement. The pith and roll axis are not parameters that we can tune, as the drone must remain horizontal or else it would fly uncontrollably. In reality, the drone has an inclination along these two axes, but only during its movements.

It is important to note that the drone's final target is the top left corner of the smallest ArUco marker. This algorithm is executed each time an image is acquired and the relative position is determined by the drone's sensors.



Forward/Backward commands We need this command to control the drone position so that its distance to the target is in an interval centered around the target value, in our case 20 centimeters. Above this interval, we need to apply a roughly proportional forward velocity, to minimize any recoil, and so that the drone comes to a stop within the acceptable range. Symmetrically, when the drone is too close to the target, we apply a slight backward velocity.

Up/Down and Left/right commands Just as with the forward/backward command, when the drone is below the target, a positive up/down velocity is applied to make it ascend, and symmetrically, when it is above, a negative velocity is applied. The same needs to be done to the left/right command, but it will imply synchronicity with the yaw control, as explained below.

Yaw command We observed that applying a constant yaw velocity to the drone results in oscillations, hindering precise docking. To stabilize the drone's movement, we needed to implement a PID loop, which will be explained later. The figures below illustrate how the yaw error is evaluated and showcase the control loop code.

5.2 Preliminary algorithm

As mentioned, the initial guidance algorithm is relatively basic.

In this algorithm, we determine the distance (x, y, z) between the drone and the target. We then calculate the angles on the vertical and horizontal planes using a straightforward arctangent function. By applying a constant vertical or yaw velocity when the angles exceed a specific threshold, we can center the target within the drone's field of view. We only move the drone forward—proportional to the distance to its target position—once the target is centered (i.e., when both angles are below the threshold).

The code for this preliminary guidance algorithm can be found in the annex.

While this method allows the drone to approach and maintain the target centered, it does not consider the angular alignment between the target frame and the drone frame. This omission makes the algorithm unsuitable for our docking purposes because it fails to align the drone precisely with its target. To address this issue, we must utilize the drone's left-right command to ensure the normal angle approaches zero, as depicted in the figure below.

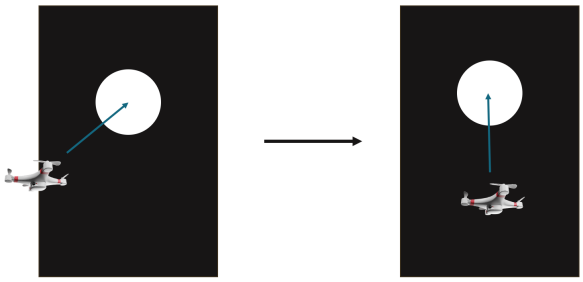


Figure 6: Objective of Normal Alignment

For effective realignment, we need to finely tune the left-right to work in concert with the yaw command. This adjustment is crucial to avoid losing sight of the target, but is impossible with the simple laws used by the preliminary algorithm.

Please refer to the following link for a video of the first version of our docking algorithm: **https://youtu.be/XzU6y0Kbspk**.

As demonstrated, the drone was effectively able to reach the target, but, in a docking purpose, would not be able to dock with it as the angle between the two frames was non zero.

Moreover, the drone is quite fast, in order to to minimize the effect of any disturbances that could exist. But for sensitive operations such as space docking, a slower speed is required to ensure that the components come together without any damage. Therefore, we need a more precise approach, that wan get rid of as much disturbances as possible, as it is permitted by the drone.

Furthermore, while the yaw control effectively centers the target in the drone's field of view, the realignment process is too abrupt. This maneuvering during the docking phase

could potentially damage the docking vehicles. To minimize this risk, we have then conducted various tests to determine the optimal method for a smoother and more efficient control. Simply reducing the coefficients was ineffective as did the drone would then simply not realign with the target.

5.3 PID Controller

5.3.1 • Definition

For our final algorithm, we mainly used a PID controller (Proportional-Integral-Derivative controller), its description and configuration is depicted below. As the name implies, this controller produces a response $r(t)$ to a signal $s(t)$, which we aim to minimize, based on the following equation:

$$r(t) = P \cdot s(t) + I \cdot \int_0^t s(\tau)\, d\tau + D \cdot \frac{ds(t)}{dt} \tag{1}$$

We quickly determined that the integral term was not conducive to our objectives, as our intent was to slowly approach the target. The integral component accumulated excessively, thus failing to provide a meaningful response to the current signal.

Moreover, to achieve a non-zero set point, such as for the x value—where, in our scenario, reaching $x = 0$ would entail collision with a wall and the loss of target visibility due to proximity—we adapted the algorithm to utilize the error $e(t) = s(t) - X$:

$$r(t) = P \cdot e(t) + D \cdot \frac{de(t)}{dt} \tag{2}$$

Ultimately, the output values were constrained to a range of -20 to 20.
Henceforth, we refer to this algorithm as a PD controller or PD algorithm.
In python, this controller can be implemented as simply as follows :

```
PD_coeffs = [100, 10]

proportional_z = PD_coeffs[0] * self.z
derivative_z = PD_coeffs[1] *
            (self.z - self.previous_z) / (time - previous_time)

command = int(np.clip(proportional_z + derivative_z, -20, 20))

return command
```

5.3.2 • Configuration of the PD Controller

To configure the PD controller, we employed the Ziegler-Nichols tuning method, which prescribes a systematic approach to determining the coefficients based on the following steps:

1. Set $P = 0$ and $D = 0$, then increment P until the measured error shows consistent oscillations.

2. Record the oscillation period T and the corresponding P_{osc} value at which the oscillations stabilize.
3. Determine the final parameters as follows:

$$P = 0.8 \times P_{\text{osc}} \tag{3}$$
$$D = 0.1 \times P_{\text{osc}} \times T \tag{4}$$

For instance, controller tuning for the z axis, intended to reach zero, initiated oscillations at $P = 200$. Stability was achieved at $P = 500$.

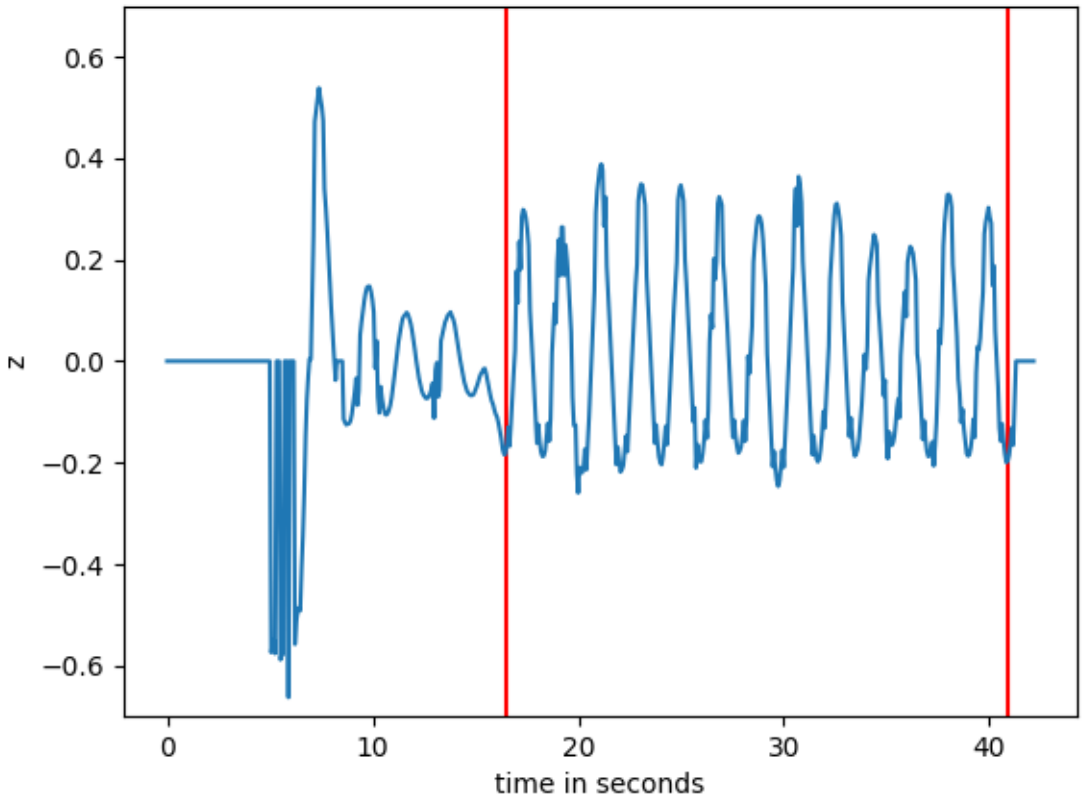


Figure 7: Oscillations of z (position along the vertical axis) measured for P = 500, D = 0

The oscillation period was determined to be $T = 1.87$ s (13 oscillations in 24.4 seconds), resulting in the final coefficients:

$$P = 400 \tag{5}$$

$$D = 94 \tag{6}$$

Although the PD controller, as tuned by this method, swiftly met our objective, it proved too rapid for our docking procedures. Consequently, we divided both coefficients by four, a modification that moderated the approach velocity toward the target.

This methodology was subsequently applied to the x and y axes, furnishing a dependable mechanism for target alignment when the drone was approximately positioned.

Nonetheless, the PD controller proved infeasible for yaw control as the camera feedback lacked the requisite precision; the derivative term thus offered no advantage. We opted for a Proportional controller, which was manually tuned.

5.4 Final guidance algorithm

We quickly notice that the initial guidance algorithm only works in a very specific starting configuration. The commands to be given to the drone must depend on its current position. It's essential not to proceed in the same direction and at the same speed if the drone is far from the target or very close, if it's facing the target or positioned sideways. Therefore, we decided to construct phases that correspond to specific states of the drone and adapt the guidance commands accordingly. In Python, this process was implemented using classes. Each class includes a method that calculates the commands to be given to the drone, as well as a method that calculates the phase transition, meaning which phase the drone should be in based on its previous phase, whether it sees the target or not, and, in the case where it does see it, its position relative to the target.

5.4.1 • Phase Lost

The phase "Lost" corresponds to the state where the drone does not detect the target. Since navigation is relative, when it doesn't see the target, it has no idea where it is. Hence, it is "lost". Nonetheless, the drone has a sensor that provides access to its altitude, a data point we'll exploit during the "Lost" phase. The objective of this phase is to look in all directions to find the target and then move on to the next phase for docking. The drone is in the "Lost" phase at the beginning of docking just after takeoff and may again enter the "Lost" phase if it loses sight of the target at any phase of docking. Searching for the target is conducted in a 3-dimensional space, yet we know that the target is vertical. Unlike a space shuttle, the drone cannot rotate along the pitch or roll axis. The "Lost" phase is divided into 4 stages that follow one another.

- The first stage is where it waits without doing anything. If an error occurs in the navigation algorithm resulting in no target detection in 1 or 2 images, the drone should not move away from the target or lose its direction. This step is very short; after 2 seconds, it moves on to the next one.

- The second stage involves making the drone rotate around itself to see if the target might be at its altitude but not necessarily facing it. When the drone loses sight of the target in the middle of docking, for example due to a deviation too far to the right or left, it is during this rotation that it finds it again. Since the drone's camera has a relatively wide angle, during this stage, the drone often detects the target, unless it is not at the same level at all. The next two stages aim to explore the bottom and the top by performing ellipses.
- The drone starts by slowly descending while rotating until it reaches a height of 50 cm, at which it gets very close to the ground. This algorithm was designed for a drone flying on the ground with ground underneath; the target has a minimum height.
- During the fourth and final stage, the drone ascends, still rotating, until it reaches 2 meters. This height was chosen for flying a drone in a room with a ceiling 2.5 to 3 meters high, but one could also choose not to impose a limit and let it ascend as high as desired outdoors.

When the drone reaches its maximum height without detecting the target, the algorithm stops, and the drone lands.

This algorithm is specific to our drone and could not be applied as is in space, as the height of the target has no lower bound, but a space shuttle can orient and rotate in all directions and the absolute position is of the target in known, allowing the spacecraft to roughly known in which direction to look at.

5.4.2 • Phase Getting Closer

This phase consist to getting close to the target and is applied outside the security zone ie at long distance. It contains three subphases which are:

- **Phase A**: Here, we only align the drone with the target using yaw control and vertical position (z) control. Since the drone does not get closer horizontally (along the x-axis), this phase allows for the target to be locked in the center of the drone's view before moving forward.

  Thus, in practice, this phase consists of a PD control on z and yaw. Under absolute threshold values of Z and yaw, the transition to phaseB is assured.
- **Phase B**: Once the target is locked in the drone's view, we apply two laws :
  - PD loop on z and yaw : As before, we assure that the target is locked in the center of the camera's field of view by applying a PD control loop on the z-axis and the yaw parameter.
  - The law guidance_far_away : This law only provide output to the forward/backward and the left/right commands, assuring no unwanted competition with the PD loop stated before. We respectively note V_x and V_y the forward/backward velocity and the left/right velocity. This law depends on four parameters : a, b, α, β, as follows :

$$\text{Note : } \theta = \arctan\left(\frac{y}{x}\right) \tag{7}$$

$$V_y = a \cdot \frac{\theta}{1 + \alpha \cdot |x|} \tag{8}$$

$$V_x = b \cdot \frac{x}{1 + \beta \cdot |\theta|} \tag{9}$$

In this law, we can easily note that :

* $|V_x|$ increases when $|x|$ increases, allowing for a smooth and rapid meeting with the target
* $|V_x|$ decreases when $|\theta|$ increases, allowing the drone to first realign with the target before getting closer to it.
* $|V_y|$ increases when $|\theta|$ increases, allowing for a rapid realignment with the target.
* $|V_y|$ decreases when $|x|$ increases, allowing the drone, when it is far away from the target, to first get closer via a strong value of $|V_x|$ relative to $|V_y|$, before moving sideways.

We manually tuned the four parameters a, b, α, β through trial and error, resulting in the following values :

$$a = 180 \tag{10}$$
$$b = 40 \tag{11}$$
$$\alpha = 2.5 \tag{12}$$
$$\beta = 50 \tag{13}$$

From these coefficients, we can see that the lateral velocity is much bigger, and the angle θ has as powerful effect on V_x, which ensures that the drone only starts going forward when roughly aligned with the target.

You will find 3 graphs of the drone trajectory, in the 3 situations, the drone starts at the bottom left, pointing to the right. As the yaw control is disabled here, its orientation stays the same throughout the simulation. The target is the red dot, pointing to the left. :

* The first is with the parameters we tuned. You will remark that the drone is not perfectly aligned with the target at the end, but this small misalignment will be corrected in the third phase, the docking phase, thanks to a PD control on y.
* The second is when the parameter β is too big, so that the drone first realign itself and then proceed the move forward. While it would theoretically be a valid law, we would risk losing sight of the target as we do not get close enough to it rapidly.
* The third graph is when the parameter β is not big enough, so that the drone does not realign itself before moving forward. In that situation, the docking fails as we do not approach the target on its left side.

- **Phase C**: This phase is activated when the drone is relatively close to the target (in what we defined as the security sphere), but either the horizontal angle or the vertical angle is not low enough to go to the docking phase. Therefore, applying forward velocity could cause it to lose sight of the target or even collide with the wall. This is why we suppress

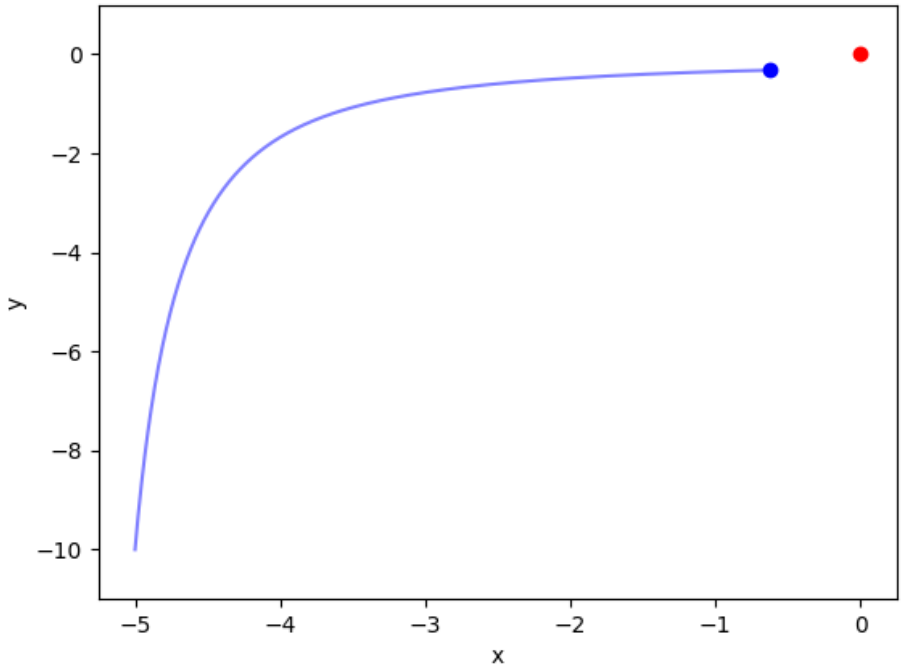


Figure 8: Correct guidance, achieved with our tuned parameters

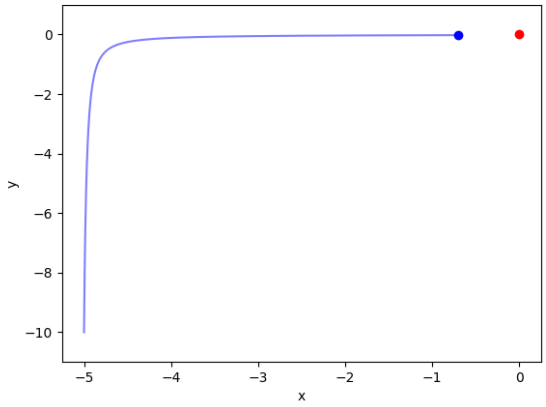


Figure 9: Incorrect guidance, the drone does not move forward quickly enough

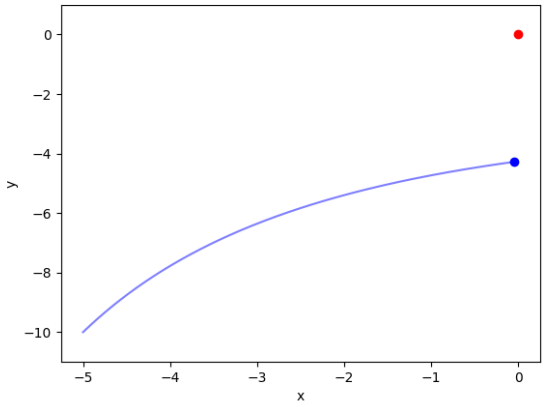


Figure 10: Incorrect guidance, the drone did not realign itself

the forward speed and apply only lateral speed computed by `guidance\_far\_away` to return it to the security cone.

As you may have noticed, the z and yaw PD loops are constantly active to ensure that the target remains in view.

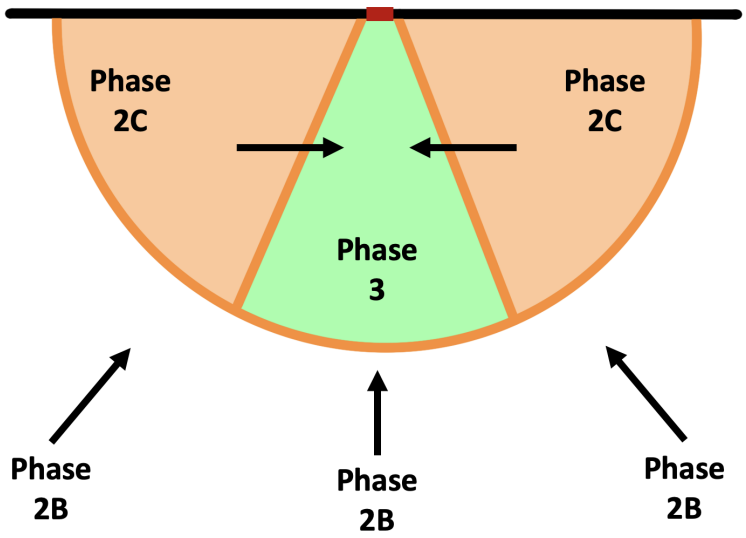


Figure 11: Overall view of different phases. Phase 2: Getting closer, Phase 3: Docking.

5.4.3 • Phase Docking

We activate this phase when the drone is is the security sphere and the horizontal and vertical angles are both below the threshold. This phase requires high precision and slow speed since we are very close to the target, and this is the moment when the different vehicles get docked. It is applied when the drone is in the security corner. The command is calculated using a PD control on x, y, z, and yaw because the PD approach allows for fast convergence with no oscillations. The drone remains quite stable in front of the target.

5.4.4 • Phase Landing

This phase would not exist in a real docking maneuver. It is quite a simple phase, as we activate this phase only if the absolute values of x, y, z and yaw are below a certain threshold. We simply proceed to move backward to not hit any wall, and then land the drone.

5.4.5 • Docking with the final algorithm videos

Here after can be found videos of the docking of the drone and a recapitulation of the final docking algorithm :

- Lost phase : **https://youtu.be/GCgSEGs4mZc**.
- Docking with the final algorithm : **https://youtu.be/zJnFji9IIs0**
- Docking algorithm:

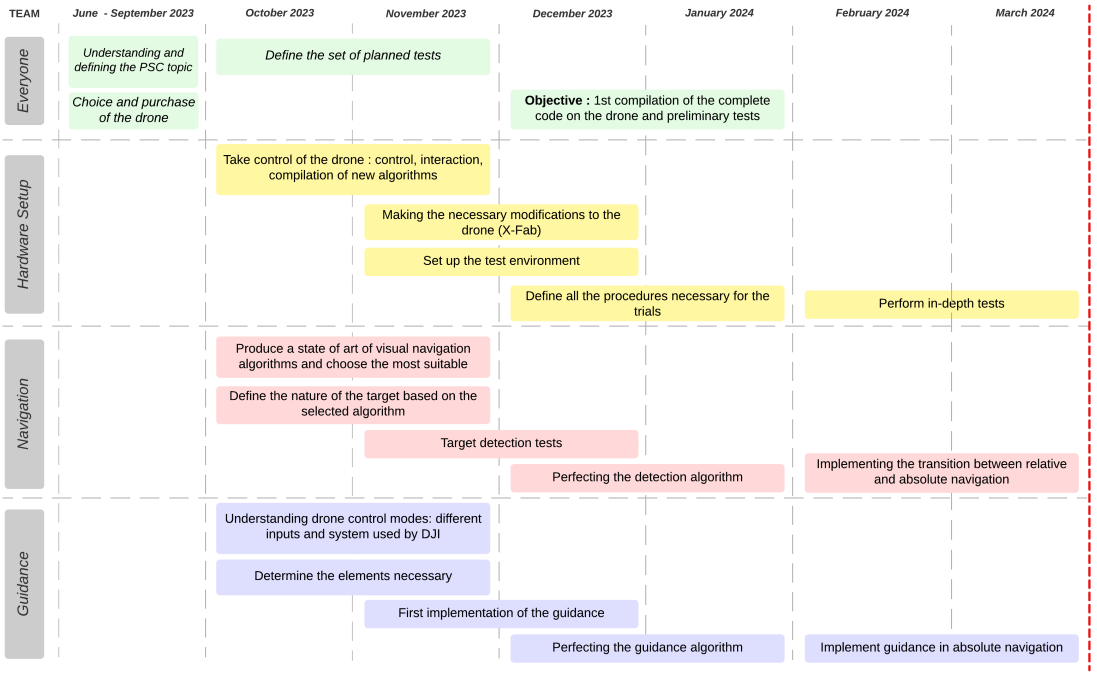
```
def compute_drone_command(self):
    if not self.is_very_close_to_target:
        self.begin_docking_time=datetime.now()
    position = [self.x, self.y, self.z, self.yaw]
    if not self.is_close_to_target():
        print("phase2a")
        a = self.guidance_pd_yaw_z() + self.far_away()
        return a
    else:
        command = self.guidance_pd_x_y() + self.guidance_pd_yaw_z()
        for d in range(4):
            command[d] = np.clip(command[d], -20, 20)

            if abs(command[d]) <= 5 and abs(position[d]) >= 0.08 and
                if command[d] > 0:
                    command[d] = 5
                else:
                    command[d] = -5
        if abs(np.arctan2(self.y, self.x)) > np.pi / 8 or abs(self.z)
        # left or right side ,we are in phase2 C
            command[0] = 0
            # We cancel the forward-backward velocity and keep the la
            print("phase2b")
        else:
        # We are inside the safety cone so in Phase 3 with a safety c
            print("phase3")
    return command
```

6
LEVEL OF ACCOMPLISHMENT OF GOALS

6.1 THE GOALS ACHIEVED

The first main goal was the **hardware setup**. We had to familiarise ourselves with the drone and its functionalities. We rapidly had control over it and understood how to operate it in a well-defined test environment. At first, we learnt to control it with our phone, and we quickly switched to computer control on Python.

The goal that followed was related to **relative navigation**. Before flying towards the target, the drone had to locate itself in space relative to it. Thus, we successfully developed an algorithm that recognised a target from the drone's video feed. The outputs were the drone's coordinates with the target as its origin.



Once we achieved locating the drone relative to the target, we coded the **guidance algorithm**. The initial basic guidance algorithm, developed before the hardware setup and navigation had conclusive results, was deeply modified so the drone could move towards the target in the most efficient way. It took us many trials and errors to further improve the algorithm for the very effective one we managed to create.

6.2 The goals abandoned

As we've seen, spatial docking is relative, meaning that the drone moves in relation to the target. In our case, the drone found its bearings using the ArUco codes that served as targets. During our project, we had the idea of implementing the phase preceding docking, which uses absolute navigation. This was one of our objectives at the end of the project.

Indeed, before the space shuttle gets close to the space module where it's going to dock, it's too far from the latter to move relative to it. In our case, this corresponds to the situation where the drone is too far away for the onboard camera to see the target. The spacecraft must therefore move according to its own position, i.e. absolutely, towards the space module, whose position is also supposed to be known. This is less precise than relative navigation, but initially enables the spacecraft to be brought closer effectively before the docking phase.

To implement an absolute navigation system on the drone, it is necessary to have access to its GPS coordinates. However, the DJI Tello drone does not have a native GPS system. It would therefore have been necessary to purchase a new, more powerful drone, or a GPS module that would have had to be attached to the drone. The drone also managed to locate

the target very well during our tests in a classroom, even at a distance from it. As the absolute navigation phase was not the main focus of our project, these reasons led us to abandon the implementation of this phase. We preferred to value our time perfecting the guidance algorithm which was at our project's core.

6.3 The problems that arose

At the end of our project, we'll be delivering a presentation of our work to our tutor's company, The Exploration Company. This will be an opportunity to share with them the various practical problems that arose when using the drone.

Firstly, the batteries heated up quite quickly, causing the drone to overheat. Beyond a certain temperature limit, the drone was unable to take off until it cooled down. The location of heat sources in relation to the space shuttle's computer components must therefore be carefully considered to avoid overheating.

The drone's precision was also less than perfect, as it didn't remain entirely still in the air in front of the target. These small disturbances made it impossible to move the drone at low speed. Even though we know that the electronics on board the space shuttle are much more sophisticated than those on our drone, these small residual movements will still have to be taken into account. Since the shuttle has to be move very accurately during the final docking phase, its precision at very low speed is crucial for the it to dock successfully.

Lastly, the camera sometimes returned frames with a lot of noise, and while for some of theses frames the navigation simply did not return a position value, which was an anticipated case in our program (the lost phase), it would also sometimes gives aberrant values. To limit this effect, we implemented averaged the last 10 values and well as eliminating the two most extreme values, as explained before.

REFERENCES

[1] Vongbunyong, S., Thamrongaphichartkul, K., Worrasittichai, N., & Takutruea, A. (2021). Automatic precision docking for autonomous mobile robot in hospital logistics - case-study : battery charging. IOP conference series

[2] Kolja Poreski. Autonomous Docking with Optical Positioning. Bachelor Thesis, Hamburg Universität

[3] Guglieri, G., Maroglio, F., Pellegrino, P., & Torre, L. (2014). Design and development of guidance navigation and control algorithms for spacecraft rendezvous and docking experimentation. Acta Astronautica

[4] J. G. Aviña Cervantes (15 Février 2005). Navigation visuelle d'un robot mobile dans un environnement d'extérieur semi-structuré. Institut National Polytechnique de Toulouse

[5] Li, S., Ozo, M., De Wagter, C., & De Croon, G. C. H. E. (2020). Autonomous Drone Race : a computationally efficient vision-based navigation and control strategy. Robotics and Autonomous Systems

[6] Pinard, D., Reynaud, S., Delpy, P., & Strandmoe, S. (2007). Accurate and autonomous navigation for the ATV. Aerospace Science and Technology

[7] Uri Kartoun (July 24-26 2006). Vision-Based Autonomous Robot Self-Docking and Recharging. World Automation Congress, Budapest, Hungary

[8] Optimum settings for automatic controllers, J. B. Ziegler and N. B. Nichols, ASME Transactions, v64 (1942), pp. 759-768.

[9] Rule-Based Autotuning Based on Frequency Domain Identification, Anthony S. McCormack and Keith R. Godfrey, IEEE Transactions on Control Systems Technology, vol 6 no 1, January 1998.

APPENDIX

.1 Guidance algorithm of the preliminary algorithm

```
threshold_angle = 0.15
goal_distance = 0.2

def guidance(x, y, z):
    command = [0, 0, 0, 0]
    # left_right, forward_backward, up_down, yaw velocities
    forward = True
    horizontal_angle = math.atan(y / x)
    vertical_angle = math.atan(z / x)

    if abs(horizontal_angle) > threshold_angle:
        command[3] = 20 * sign(y)
        forward = False

    if abs(vertical_angle) > threshold_angle:
        command[2] = -20 * sign(z)
        forward = False

    if forward:
        if x > 1:
            command[1] = 30
        elif x < threshold_angle * 0.7:
            command[1] = -10
        elif x < threshold_angle:
            command[1] = 0
        else:
            command[1] = int(x * 30)
    else:
        command[1] = 0
    return command
```

.2 Algorithm of the final algorithm

The full code is provided in a **separate file** attached to the report email, because it would be too long to include it here. 3 main files can be highlighted :

- The main.py file : it is the brain of our code, it connects to the drone and calls the other programs.

- The navigation.py : All the navigation is done here, the program takes an image taken by the drone, and returns the position of the target, if is has been detected.
- The phase.py : As explained before, all the phases are defined in this file, as classes. The program takes the position of the target and returns the velocity command.