POLITECNICO DI TORINO

**Corso di Laurea Magistrale**
**in Ingegneria Aerospaziale**

Tesi di Laurea Magistrale

# Guidance and Control strategies for UAS applications for Precision Agriculture


**Relatori**
Prof. Giorgio Guglieri
Dott.ssa Nicoletta Bloise

**Candidato**
Lorenzo Becce

Dicembre 2020

# Contents

# List of Algorithms

# List of Figures

# List of Tables

**Abstract**

The present work focuses on the development of a waypoint-based guidance algorithm to conduct a multirotor UAS over a number of plants (grapevines, but application to other crops is worth investigating) in order to precisely administer Plant Protection Products (PPP).
The position of each plant requesting intervention is fed into the algorithm, which will generate a feasible path by means of a Traveling Salesman Problem (TSP) solver and a suitable path planning routine (Theta*).
In the second part of the work, the UAS' autopilot has been implemented in Simulink and said path was fed to it for the execution of the task.
Since the development of the spray controller is still underway, the need for a certain flexibility in the planning effort arose and therefore this work is intended as a framework to be expanded by further research.

# 1: Introduction

## 1.1 Introduction

Precision Agriculture is making numerous important leaps forward recently, and one of its major advantages is to reduce and optimize the amount of resources employed while reducing and optimizing the man-hours needed for crop management.

In the typical vineyard scenario, plants often require the spraying of products to counter the insurgence of diseases and parasite attacks that requires highly trained personnell to walk the yard, row after row, with several kilograms of tank/sprayer system on their shoulders to assess each plant and, in case, treat it.

The present work is part of a larger research effort aiming at automating this specific task by means of Unmanned Aerial Systems (UAS). The project foresees the employment of a small, off-the-shelf UAS to scan the field with multispectral imaging sensors, the development of an AI algorithm to individuate, by canopy characterisation, the single plants requiring the administration of Plant Protection Products (PPP) and finally the use of a larger, dedicated UAS to precisely administer the PPPs on the marked plants.

In particular, this work focuses on the development of a waypoint-based guidance algorithm to conduct the latter multirotor UAS. The input to this system, together with a Digital Elevation Model (DEM) of the vineyard, will be the position of each plant to reach as individuated by the AI. [1] suggests an unsupervised algorithm for the evaluation of features in a vineyard which could be of great use during the upcoming work. Since for the moment the plant recognition algorithm is not available, random points will be generated for the testing and evaluation of performance, at first in a totally random way, then random plant positions from a realistically-arranged vineyard will be picked.

The algorithm will then generate a feasible path by means of a Traveling Salesman Problem (TSP) solver and a suitable path planning routine (Theta*, as we will see) where needed. The generated path is finally uploaded in the form of a series of waypoints into the sprayer UAS for the execution of the task.

The advantage of using a multirotor system for this mission is the availability of a large airstream from the rotor downwash, which is very effective in the nebulisation and channeling of fluids towards the environment below. Since the development of the spray controller is still underway, though, many open questions require a certain flexibility in the planning and control effort, making this work a pathfinder to be followed by further research.

Other applications for this system can be foreseen, such as the precise administration of fertilizers on hardly-accessible hillside areas.

## 1.2 Methodology and Organization

All the software code has been written and ran in MATLAB® R2018b.

Part I designs an algorithm which organizes the input waypoints in the shortest feasible sequence and, if required by obstacles in the way, searches for the shortest path to get through them. A first chapter, no. 2, introduces the Traveling Salesman Problem with a brief look at some of the solution methods and then transitions to the explanation of a picked solver.

Chapter 3 introduces then some path planning algorithms of interest for the problem, namely A* and its extension Theta*, going through the MATLAB implementation and comparison of the two methods on three dimensional grids generated *ad hoc*.

Part II goes through the implementation of the UAS dynamics model (chapter 4) and of the prototype of guidance and control logic in Simulink® (chapter 5) with the aid of the Control System Toolbox™ for the interactive tuning of controllers. After the validation of the model, some tests are performed in chapter 6 and results are presented and discussed.

The work is closed with chapter 7 which presents the conclusive remarks and outlines the future steps to take to improve on the developed model.

# Part I

# Planning

# 2: The optimal sequence

The algorithm starts with the position of the plants to be treated fed as input by the charachterisation routine. For the sake of energy and time saving, it makes sense to individuate the shortest (closed) path through each and every of them: this is a typical iteration of the so called Traveling Salesman Problem (TSP). The system will run through the path and administer the PPP where required, starting and returning to a precise location whose coordinates are appended to the list of waypoints.

## 2.1 The Traveling Salesman Problem[1]

The general formulation of the problem, to which it owns its name, could be phrased like this: *Given a number of cities that a salesman has to visit for business, what tour can he/she take so as to visit each and every one of them exactly once while minimizing the total distance traveled?* It goes without saying that the cities represent points in space of interest (like waypoints on an actual tour).
This is a well-known and debated open problem in the scientific community which has many practical implications, especially in the logistics of freights and passengers, routing of networks, microchip manufacture and many other applications that it's not worth noting here.
Since this problem is currently open, i.e. with no closed solution, some form of heuristic has to be employed. The choice of said heuristic depends not only on the goodness of the solution, but also to its computational requirements, which can increase tremendously with the number of cities. A brief description of some of the possibilities, on which the reader can find further information in [2], follows.

**Brute Force Search (BFS)** This method is based upon picking the best of all the possible combinations. While being somewhat simple in its formulation, the reader can easily see how prohibitive this evaluation becomes when the number of cities reaches immediately after what can be considered "a handful": the number of possible permutations goes with the factorial of the number of cities. For example, given 15 cities, the possibilities are

$$15! = 1.307.674.368.000$$

While this would be an exact algorithm (hence not a heuristic), it is obvious to exclude this possibility.

**Nearest Neighbor (NN)** One idea is to proceed by having the salesman move to the nearest city from where they currently stand. This does not necessarily generate the shortest possible path ([3] have shown that on certain specifically arranged scenarios it can even give the *worst*, that is longest, possible solution).

[1] The curious reader interested in more stimulating heuristics is encouraged to visit the comprehensive Wikipedia page. Albeit not very professional to cite, at the time of the composition of this work the English article contained an extensive overview of the historical and technical aspects of a problem much important to the last decades or even centuries of technical progress.

**k-opt Heuristic, a.k.a. Lin-Kernighan** k-optimization is an iterative algorithm which foresees taking an existing path (generated by some other means, like a random path) and exchanging $k$ edges (linkages between cities), and rearrange them so as to yield a shorter route. The number of edges is usually two or three, but it does not necessarily have to be constant: a variation of the method, called V-opt, calls for a variable number of edges to be rearranged at every iteration.

**Genetic Algorithms (GA)** These solvers create a population of possible paths, evaluate the *fitness* of each of them and *evolve* the fittest ones based on some logic, such as random permutations or k-opt heuristics. More details are in the following section.

Of course, the list would be much longer, but for a basic presentation and the scope of current application the presented methods are sufficient.

## 2.2 Solver selection

In an effort to best exploit the available time and to channel the efforts to the wider guidance problem, it has been decided to pick a solver from the abundant choice available at the MATLAB Central portal. The chosen solver is a very complete package spanning several cases like open TSPs, multiple salesmen problems or vehicle routing problems[2] developed by Joseph Kirk: the `MATLAB Traveling Salesman Problem (TSP) Genetic Algorithm Toolbox`[3]. The content of this toolbox that are of interest for the problem are the NN-TSP and the GA-TSP (Genetic Algorithm) solvers which, as the author suggests in the examples, prove very fast when executed in series: the output to the NN algorithm are fed into the GA for further refining.

### 2.2.1 Mechanics

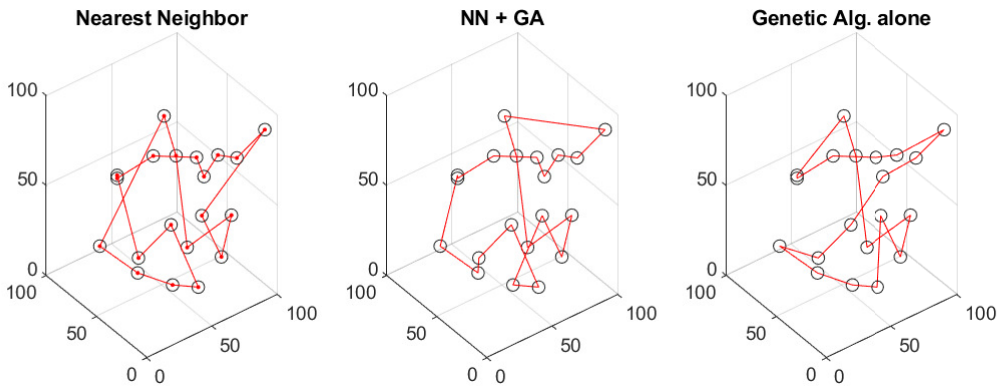
The GA starts with the generation of a number of random permutations of the original path (which can be an existing path, fed as input). Those permutations represent the initial population, which is then evolved by chosing the best ones (four in this case) and mutating them so as to rebuild a new population of the original size (hence, the remaining individuals from the choice are discarded, or better *extinct*, mantaining the biological terminology from which the GAs take inspiration). The possible permutations are flip, swap, and slide. If the cities are contained in the rows of a matrix, a path is represented by a vector containing the indices of said rows in some order.

- Flip: a portion of the path, chosen by a random pair of indexes (start and end of the segment) is reversed.
- Swap: the positions of two randomly chosen cities in the path are swapped.
- Slide: the last city of a random segment is pulled out and reinserted at the beginning of the segment, which has now slid forward one position.

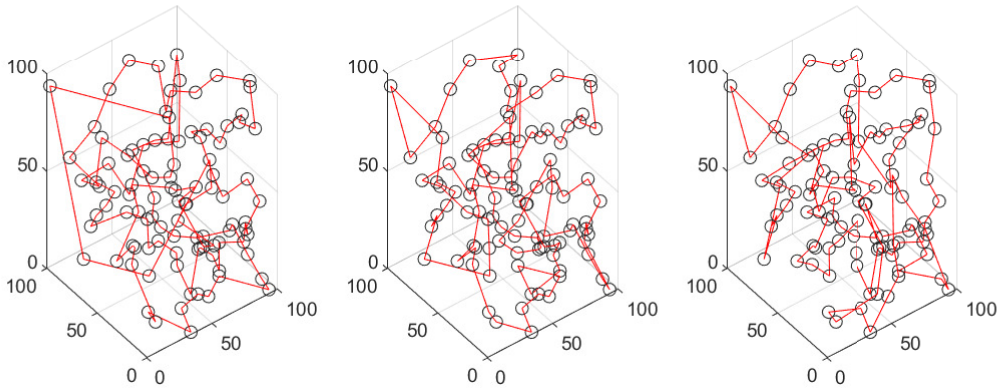
The process comes to an end after a predetermined number of iterations is reached.

[2] A popular variation of the TSP concerning the optimisation of the paths of one or more freight vehicles among a series of deposits. The vehicles can even have a limited capacity or even a determined loading/unloading sequence: it is easy to imagine how such kind of problems is relevant in the modern logistics of goods and passengers.

[3] Joseph Kirk (2020). Traveling Salesman Problem (TSP) Genetic Algorithm Toolbox *https://www.github.com/rubikscubeguy/matlab-tsp-ga,* GitHub. Retrieved June 16, 2020.




(a) 20 cities




(b) 100 cities

Figure 2.1: The evolution of the test paths in the first two cases

### 2.2.2 Comparison

A brief comparison between two methods has been put in place: the GA alone and, as suggested by the author, a two-stage computation executing the NN algorithm and feeding its result to the same GA as the first case.
Sets of 20, 100 and 500 cities have been randomly generated in 3D space, spanning 100 metres in each direction. The running times have been recorded using the well-known `tic/toc` MATLAB commands and are compared in table 2.1 together with the best distances found. Figure 2.1 shows the paths calculated at each step, except for the 500-cities case which would add no further clarity than actual confusion.
The results from table 2.1 show the clear avantage of the combined approach over the direct run of the GA: as described previously, the NN produces a widely suboptimal solution and the second run provides a substantial improvement of about 2.2%, 8.9% and 2% (over the NN solution) in the 20, 100 and 500-city scenarios respectively. The direct run is still 1.4%, 6.7% and 158.8% worse than the combined run in terms of total length, even though having taken 92.9% and 13.8% longer and 62.3% shorter, respectively. We can see that the computational times of the direct run with respect to the two-stage execution descend rapidly with the increase of the population, at the expense of an ever-worsening solution.
In conclusion, the combined approach has been selected.

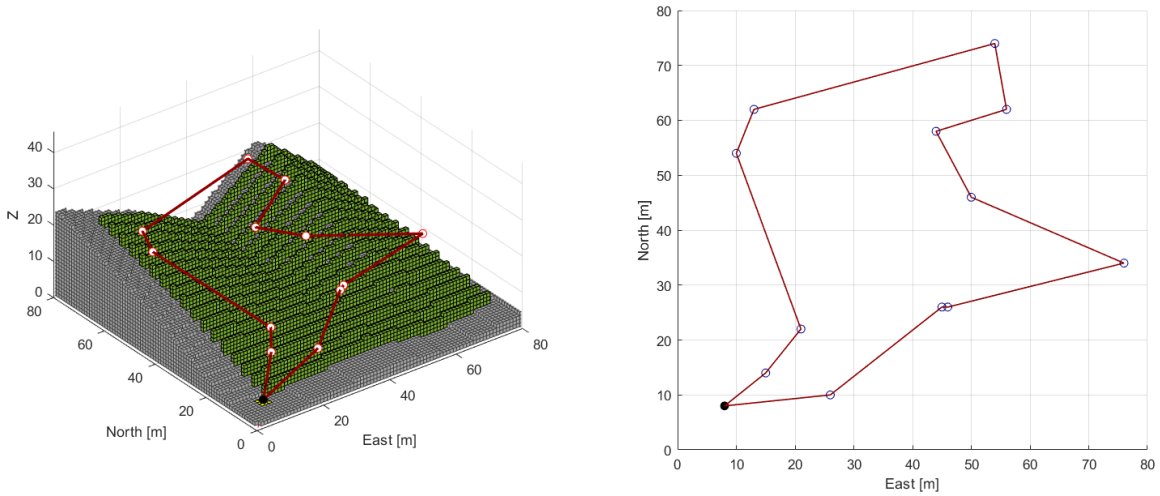
| | 20 cities | | 100 cities | | 500 cities | |
|---|---|---|---|---|---|---|
| | $t$ | length | $t$ | length | $t$ | length |
| NN | 1.38 | 646.36 | 5.139 | 1949.9 | 17.5 | 5263.5 |
| GA | 2.91 | 632.05 | 10.429 | 1775.6 | 143.6 | 5157.0 |
| **Total** | **4.296** | **632.05** | **15.568** | **1775.6** | **161.1** | **5157.0** |
| **Direct GA** | **8.287** | **640.86** | **17.726** | **1883.3** | **60.7** | **8191.2** |

Table 2.1: Comparison between combined approach and direct run

## 2.3 Application

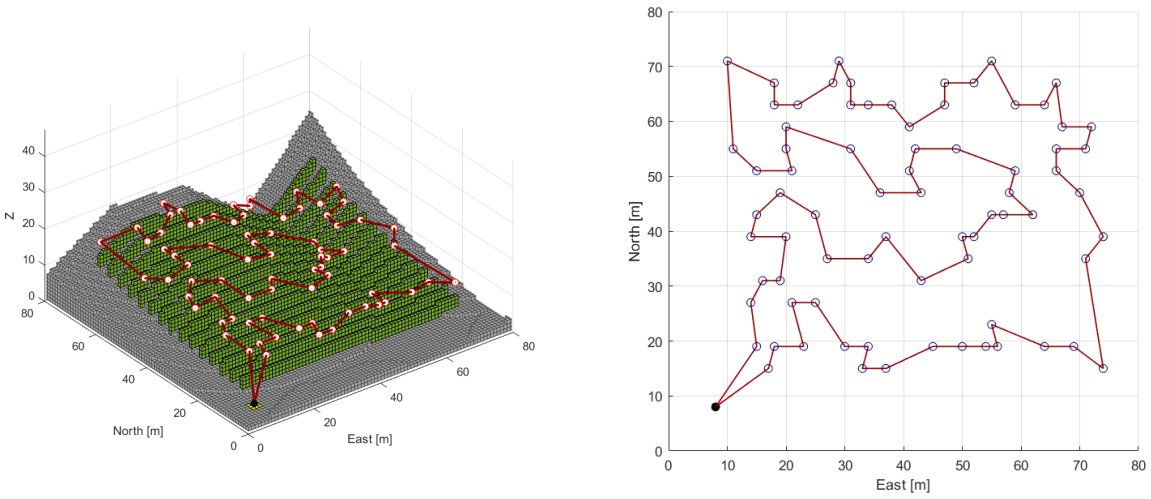
The selected algorithm has been applied to a simulated vineyard built and visualised in MATLAB. The rows are uniformly distributed, as are the plants in each row. Each plant's height matches the local map height and spans three metres above it. The $x, y, z$ coordinates of the plants are registered in a list, with their height increased by an arbitrary height (4 metres in the present case) from their tops. This constitutes the list of possible waypoints, some of which the system will randomly pick, in a number specified by input, to become actual waypoints.

A pair of test scenarios has been put in place, with 12 and 80 waypoints each, to show the performance of the algorithm. Figure 2.2 shows the generated path for the 12-waypoint case, while in 2.3 presents the results for the 80-waypoint scene.




(a) 12 cities (b) 12 cities

Figure 2.2: The 12 waypoint case




(a) 80 cities

(b) 80 cities

Figure 2.3: The 80 waypoint case

# 3: Path Planning Algorithms

## 3.1 Introduction

The calculated optimal sequence of waypoints from the previous chapter is not sufficient, by itself, as a feasible path for the UAS to fly through: this would be close to true if the yard were a flat (even though sloped), perfectly obstacle-free surface where line of sight between each and every waypoint were guaranteed. Since this is almost never the case, especially in the hillside vineyards of Italy, a path planning approach to refine the generated tour was deemed essential. An initial proposal was made to employ Dubins paths to smooth the corners of the trajectory, but after a short literature review it was rejected due to the suboptimality of the solution when compared to other strategies involving path planning on grids, which also enabled obstacle avoidance. The next subsection illustrates the reason behind this choice.

### 3.1.1 Dubins paths

Dubins curves are minimum-lenght paths that connect two points in space with prescribed tangents to the path at said points, when the curvature is contrained to a certain turning radius. It was proved by mathematician L. Dubins in [4] that said shortest path is always composed of two minimum radius circular segments and a straight segment or three curved segments, typically shortened as CSC (Curve-Straight-Curve) or CCC. The 2D case is often used for the planning of paths of (wheeled) robots and cars with constant forward speed, but 3D adaptaions for flying systems have been devised as well, placing a further constraint on the maximum climb/descent angle such as [5]. Some papers evaluated involved interesting approaches, such as the solution of a Dubins TSP (DTSP) where each city has to be visited with a predetermined heading ([6]). The reason to not include this system is that another philosophy, based on the A* search algorithm, can easily include obstacles and is known to produce shorter trajectories.

### 3.1.2 On Path Planning on grids

In a most general way possible, path planning is a task consisting of the generation of a sequence of actions that will take the user (a UAS in this scenario) from a starting point (or configuration) to a goal point (or configuration). The surrounding environment is discretized in some kind of compatible notation: in the example of the navigation in a road network that a stanav may perform, the intersections are represented by nodes connected by edges (the roads themselves) in what is commonly referred to as a *graph*. The present case, however, does not feature distinct paths that can be taken and instead models the surrounding environment as a *grid*: a set of cells, regularly distributed and characterised by the possibility of being traversed of not, connected by a series of allowed movements. Often, the cells, which are portions of 2D or 3D space and therefore not practical to handle, are themselves represented as nodes, hence 0-dimensional points in space described by coordinates and connected by edges representing the possible movements: this makes the two examples virtually identical from the mathematical standpoint, and hence able to be treated with the same instruments.
As introduced above, the path is a sequence of actions to be undertaken by the system, possibly with the lowest effort which is generally minimised by the planning algorithms. This is done by

means of a **cost** system: some form of cost function is implemented in order to quantify how convenient is the use of an edge or movement with respect to the other possibilities. In the example of the satnav system, the cost associated to a road segment can be proportional to the time required to drive through it, but it can also take into account traffic jams, tolls, etc.
On the other hand, for a generic system moving in a tridimensional space, the cost of the maneuvres from a node to the other can be quantified with the distance between the start and end points. An extensive presentation on planning algorithms can be found in [7][1]. The one employed here and its extension are quite popular and will be introduced in the next section.

## 3.2 A* on grids

The A* search algorithm was introduced in [8]. Many expansions have been thought of to adapt the logic to different situations and notations, and some of them will be seen in the following. The domain is discretised through a grid made up of nodes, or vertices, for each of which three values are retained:

**G-value** $g(s)$ is the cost-to-come of the vertex $s$, that is the smallest cost found to go from the starting node $s_I$ to $s$;

**H-value** is an estimate of the cost-to-go, or an underestimation of the cost to go from $s$ to the goal node $s_G$. It is identified by $h(s)$ and usually the best estimate, easy to calculate and sufficient to ensure a satisfactory performance, is the distance between $s$ and goal:

$$h(s) = \|s_G - s\| \tag{3.1}$$

Of practical usefulness is the F-value, namely the sum of G and H, which indicates the estimated cost of a shortest start-end path passing through $s$:

$$f(s) = g(s) + h(s)$$

**parent identification** is needed to trace back the path from the goal to the start once A* has succesfully reached the end of the search. In the following, it will be identified by $p(s)$.

Two data structures are the building blocks of the procedure:

**open list** contains the nodes that are still available for expansion;

**closed list** gathers all the nodes which have been expanded and for which no further examination is possible.

The idea is to move from node to node by choosing the one with the lowest F-value. Algorithm 1 is extracted from [9] and contains the pseudocode for the procedure.

In the basic initialization of lines 2 to 5, the cost-to-go for the starting node is set to zero, its parent is set to itself and the open list is initialized with only $s_{start}$ inside. Afterwards, a `while` loop is executed as long as the open list contains at least one node. The function referred to as `list.getBest()` in line 8 operates on the list `list` by extracting from it the best node in terms of F-value. After a brief check for success in line 9, the node is registered in the closed list and every possible neighbour is evaluated for the next move: each candidate is initialised with infinite G-cost and $s$ as parent, then `UpdateVertex()` is run on the two nodes.

The inner working of this function is detailed in algorithm 2, which receives the two nodes as input and compares the current G-value of $s'$ (which after initialisation is $\infty$) with the value calculated in 1 as the sum of the parent's G-value and the length of the segment connecting the two nodes according to (3.1). Line 5 checks if the node has already been registered (which could happen in case the node $s'$ has been reached on a previous occasion by a different path). Finally, line 8 recalculates the F-value of $s'$ and (re)inserts it in the open list before returning control to 1. At this point, the cycle is repeated with the choice of the best neighbour $s'$ of $s$, which now becomes $s$ itself. Figure 3.1 visualises the choice mechanism with two neighbours.

[1] Available for download at http://planning.cs.uiuc.edu/

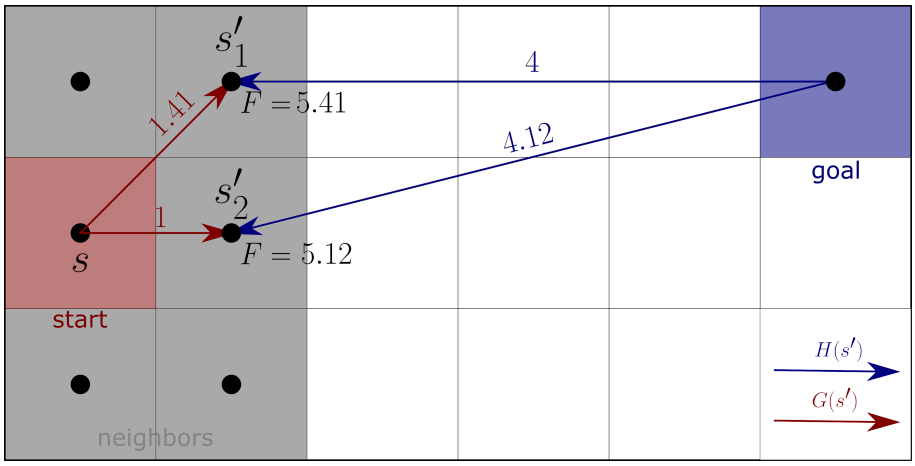



Figure 3.1: A* choice mechanism: node $s'_2$ is selected because of the lower F-cost

**A note on F-value update** The F-value is updated as follows, in this case:

$$f(s') = g(s') + \alpha h(s')$$

In this case, an $\alpha$ value which multiplies the heuristic H-value, called a "heuristic weight", is inserted. This weight influences the relevancy of the heuristic with respect to the cost-to-go. This technique does not appear in the original work cited, but was inserted by the author of the selected script (see next subsection). Another source, [10], uses an even more advanced method where two gains are employed:

$$f(s') = \alpha g(s') + \beta h(s')$$

The application of this latter method has not been deemed prioritary, but is worthy of further consideration.

**Obstacles** The obstacles are simply represented as nodes on the map which have a fixed, infinite G-value: even when evaluated as candidates for movement, their G-cost cannot be changed and therefore will remain the highest possible value, excluding them from consideration.

### 3.2.1 Selected code

As done in the previous chapter with the TSP solver, a MATLAB function implementing the A* was retrieved and adapted for the occasion.

The selected script was developed by A. Chrabieh and is available at MATLAB Central File Exchange[2]. This is a 2D solver that was choosen based on the clarity and readability of the code, due to the necessity for an intense modification. Figure 3.2 demonstrates its functionality on a simple map with $\alpha$ being set to 1.5. It must be noted that the cells are marked as an obstacle (white) if one of the neighboring nodes is forbidden, but this does not mean that stepping on a white cell is also forbidden. The obstacles here are one-dimensional objects (lines), spanning the cells' borders and not actual 2D, rectangular elements as might appear. The reader unaware of this plotting logic of MATLAB which is based, in the present case of map plotting, on cells and not points or lines, might think that the path treads on some of the obstacles, therefore failing in the pursuit of feasible planning. This little graphic mishap was not important (it only affects the 2D view) and hence not been corrected, since it would have required quite a large reprogramming effort that was prioritary elsewhere. From figures 3.2 the algorithm's main drawback is clearly

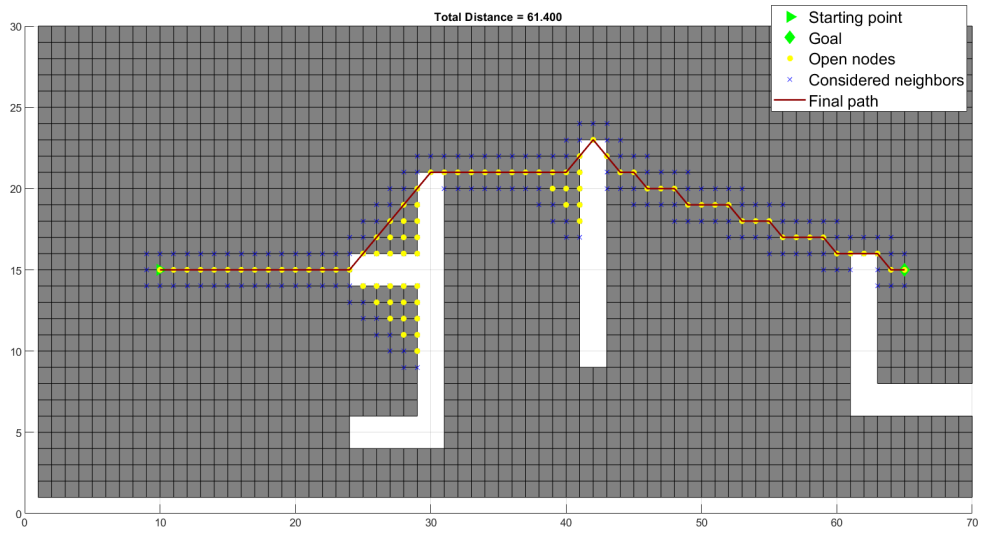
[2]Anthony Chrabieh (2020). A star search algorithm
*https://www.mathworks.com/matlabcentral/fileexchange/64978-a-star-search-algorithm*, MATLAB Central File Exchange.
Retrieved July 13, 2020.

**Algorithm 1** Basic A* path finding

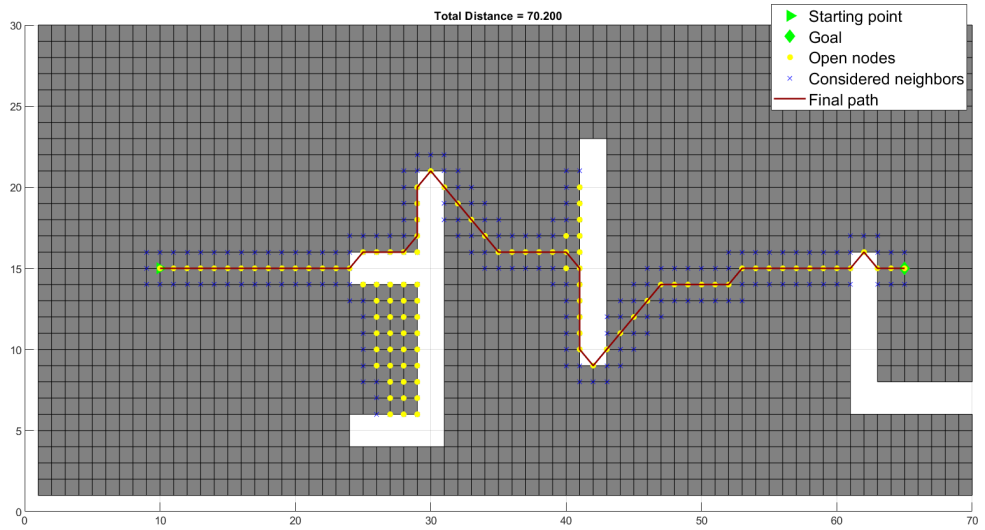
```
1.  % Initialization start
2.  g(s_start) = 0
3.  p(s_start) = s_start
4.  open = [empty]
5.  open.Insert(s_start, g(s_start))
6.  % Initialization end
7.  while !isEmpty(open) do
8.      s = open.getBest()
9.      if s = s_G then
10.         return "path found"      % Success!
11.     end if
12.     closed.Insert(s)
13.     for all s' ∈ N(s) do
14.         if s' ∉ closed then
15.             if s' ∉ open then
16.                 g(s') = ∞
17.                 p(s') = s
18.             end if
19.             UpdateVertex(s,s')
20.         end if
21.     end for
22. end while
23. return "no path found"      % Failure
```

**Algorithm 2** The `UpdateVertex(s,s')` subroutine for A*

```
1.  G_new = g(s) + c(s, s')
2.  if G_new < g(s') then
3.      g(s') = G_new
4.      p(s') = s
5.      if s' ∈ open then
6.          open.Remove(s')
7.      end if
8.      updateF(s')
9.      open.Insert(s')
10. end if
```




(a) Weight $\alpha = 1.5$




(b) Weight $\alpha = 10$

Figure 3.2: A* script functionality demonstration

evident: the course is changed by multiples of 45°, due to the geometrical constraint of working on a grid. It is clear how the true shortest path would cut those corners, requiring smaller turning angles. The next section introduces a smarter extension of the procedure to account for this.

Also, the effect of a greater heuristic weight is that of pulling more strongly the solver towards the goal. The solution takes shorter, but the result is not as accurate as with a smaller weight.

**Tridimensionalisation** As mentioned, the script works on two-dimensional grids. A tridimensionalisation effort was necessary and, luckily, not particularly challenging: usually just the repetition and slight modification of lines of code, or the shifting of vector indexes, were necessary.

The presentation of the result is put off to a later section, when the performance of the original A* will be compared to those of its extension.

## 3.3 Any-angle path planning with Theta*

Theta*, or $\Theta^*$, was introduced by Nash et al. [9] in 2010 to overcome A*'s main drawback, outlined at the end of the previous section. It is an extension of A* based on the concept of line of sight (LOS), that is the existence of an unobstructed path between two points, used as explained below. The modification is sourced directly by [9].

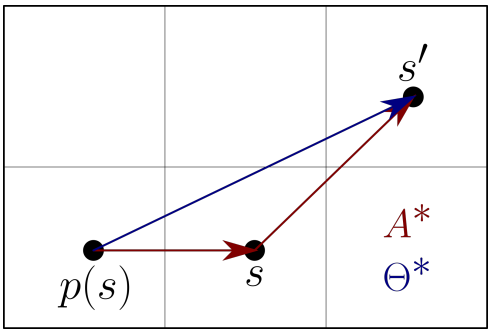



Figure 3.3: $\Theta^*$ parent choice mechanism with line of sight

As anticipated, the mechanism is the same as algorithm 1; what changes is `UpdateVertex()`, which is explained in algorithm 3. Lines 1 to 6 represent the novel approach, based on some form of LOS checking routine that will be introduced in 3.3.1. In short, if there is visibility between the candidate $s'$ and the parent of $s$, the same vertex updating routine of A* is run with this last node as parent. This happens, as an example, in figure 3.3, where LOS is clearly present between nodes $p(s)$ and $s'$. Therefore, $s$ by itself is by-passed and the corner it represents is cut. If there is no LOS, instead, line 7 and following are executed as before.

The function call terminates with the update of $s'$ in `open` and the return to the main routine.

**Algorithm 3** The `UpdateVertex(s,s')` subroutine for $\Theta^*$

```
1.  if isThisLOS(p(s),s') then
2.      G_new = g(p(s)) + c(p(s), s')
3.      if G_new < g(s') then
4.          g(s') = G_new
5.          p(s') = p(s)
6.      end if
7.  else        % In common with A*: it's the same as alg. 2
8.      G_new = g(s) + c(s, s')
9.      if G_new < g(s') then
10.         g(s') = G_new
11.         p(s') = s
12.     end if
13. end if
14.     % Update of the vertex
15. if s' ∈ open then
16.     open.Remove(s')
17. end if
18. updateF(s')
19. open.Insert(s')
```

### 3.3.1 Checking line of sight

The line of sight (LOS) checking routine makes use, as suggested by [9], of the Bresenham line-drawing algorithm. Published in 1965 to draw rectilinear segments on rasters, it can easily be modified to check if there is a straight, unobstructed rectilinear path between to points on a map. This algorithm is included in the firmware of virtually all devices needing to draw lines (plotters, monitors, graphic cards) because of one of its great advantages: its simplicity. The procedure is

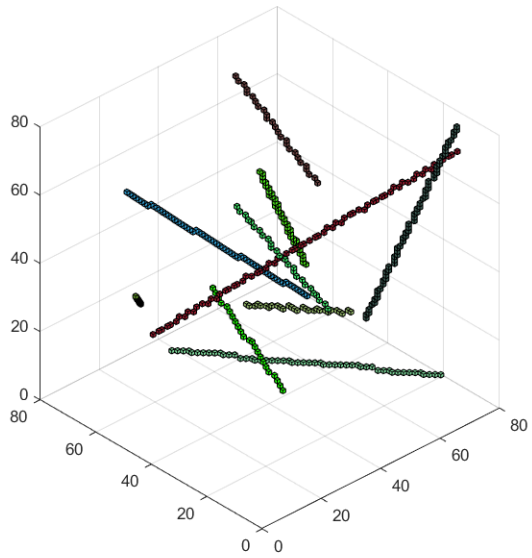
based on the parametrisation of the line segment connecting the two points to verify which cells are crossed by the line. Algorithm 4 illustrates the procedure and the modification put in place to switch from line-drawing to LOS check and to perform the check in three dimensions.

The 3D extension is just made of two instances of the 2D algorithm, working on the XY and XZ projections of the line and merging the coordinates of the blocks. Since the original, 2D algorithm can only run correctly for lines in the first octant (East to South-East), a series of preliminary checks has to be performed to determine which among $\Delta x, \Delta y, \Delta z$ is the largest with respect to the others (and therefore, in which octant the XY and XZ projections lie) and swap those coordinates so as to reorganise the problem in order for the algorithm to work. At the end, the coordinates are swapped back to the original. This operation is performed by the `swap()` subroutine, which also raises a flag identifiyng the swapped coordinates so as to restore the original order before proceeding with plotting/checking (otherwise, with the coordinates mixed up, the plotted/checked point would be another on the map).

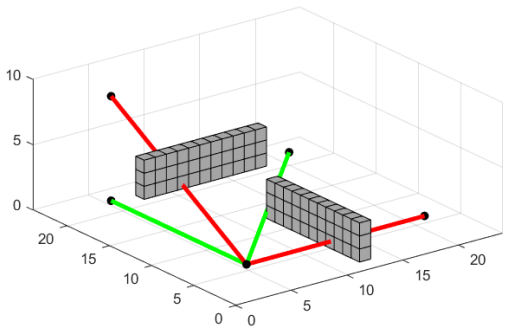
The first 11 lines perform exactly this operation, which is propedeutcal to the actual scanning cycle.

The coordinates are duplicated in line 16 and following to perform the unswapping and access the raster without altering the originals, needed to proceed with the scan. Line 23 is to be ignored in the case plotting is not the necessary operation, and instead the following cycle has to be run. If an object is encountered, line 25 stops the check with negative result: LOS is corrupted and it makes no sense to keep searching if the first object has been found.

This particular script was retrieved on the Internet[3] and adapted as needed.




(a) Some random lines generated with alg. 4




(b) Line of sight verification test

Figures 3.4a and 3.4b show the functionality of algorithm 4: in the former it has been used to generate straight lines between random pairs of points on a 3D raster (which implies using line 23), while in the latter the visibility between a starting point (coordinates 5,5,0) and an array of others has been put to test. Green lines represent a favourable outcome of the procedure, while red ones imply that one of the walls got in the way. It will be noticed that a green line actually crosses the corner of a wall, and this is due to an intrinsic limitation of the algorithm which is based on integer logic and operates on a discretised domain. A suggested solution, to keep the formulation as simple as possible, is to tighten the size of the grid (i.e. increase the resolution) so as to reduce this error as much as wanted, as is done on modern monitors and screens. A

[3] https://web.archive.org/web/20110708171823/http://www.cobrabytes.com/index.php?topic=1150.0

**Algorithm 4** 3D Bresenham algorithm

**Input:** 3D raster
**Output:** isThisLOS

1. $\Delta x = |X_1 - X_0|$
2. $\Delta y = |Y_1 - Y_0|$
3. $\Delta z = |Z_1 - Z_0|$
4. **if** $\Delta y > \Delta x$ **then**
5. $[X_0, X_1, Y_0, Y_1] = swap(X_0, X_1, Y_0, Y_1)$
6. swap_xy = true
7. **end if**
8. **if** $\Delta z > \Delta x$ **then**
9. $[X_0, X_1, Z_0, Z_1] = swap(X_0, X_1, Z_0, Z_1)$
10. swap_xz = true
11. **end if**
12. $y \leftarrow Y_0$
13. $z \leftarrow Z_0$
14. % begin of line scanning/drawing
15. **for all** $x \in [X_0 : step_x : X_1]$ **do**
16. cx = x; cy = y; cz = z; % Copy of the variables
17. **if** swap_xy **then**
18. $[cx, cy] = swap(cx, cy)$
19. **end if**
20. **if** swap_xz **then**
21. $[cx, cz] = swap(cx, cz)$
22. **end if**
23. plot(cx,cy,cz)
24. **if** map(cx,cy,cz) is an obstacle **then**
25. **return** false % there is no line of sight, execution stops
26. **end if**
27. step in $y$ plane
28. step in $z$ plane
29. **end for**
30. **return** true % no obstacle encountered, LOS confirmed

more complicated option is, instead, to use multiple iterations of Bresenham's algorithm to draw a voxel tube instead of a voxel line.

### 3.3.2 Results

#### Comparison with A* (2 dimensions)

A small comparison between Theta* and A* has been put in place on the same map as of figure 3.2 and is shown in figure 3.5. For both functions, $\alpha = 1.5$.

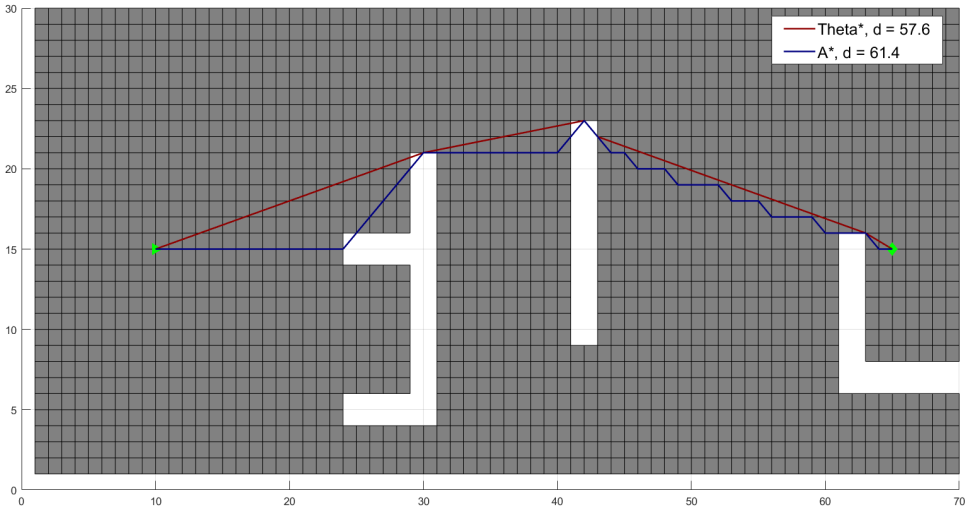
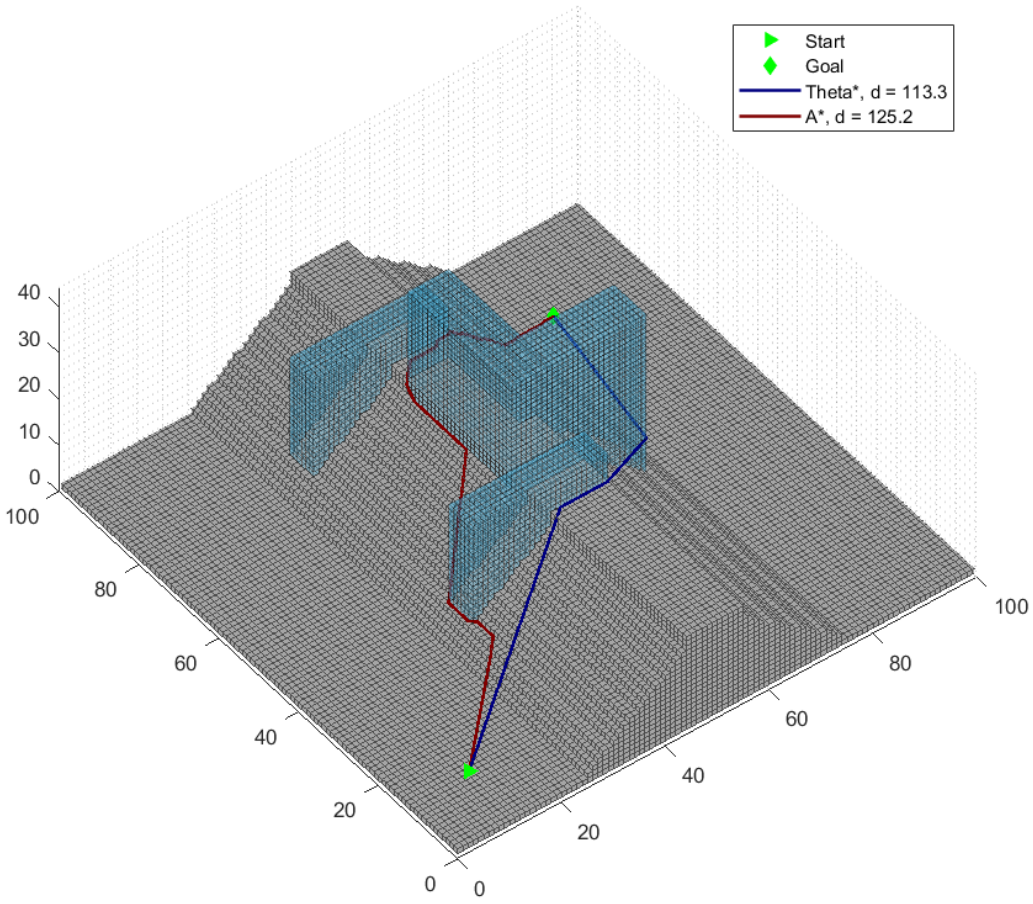


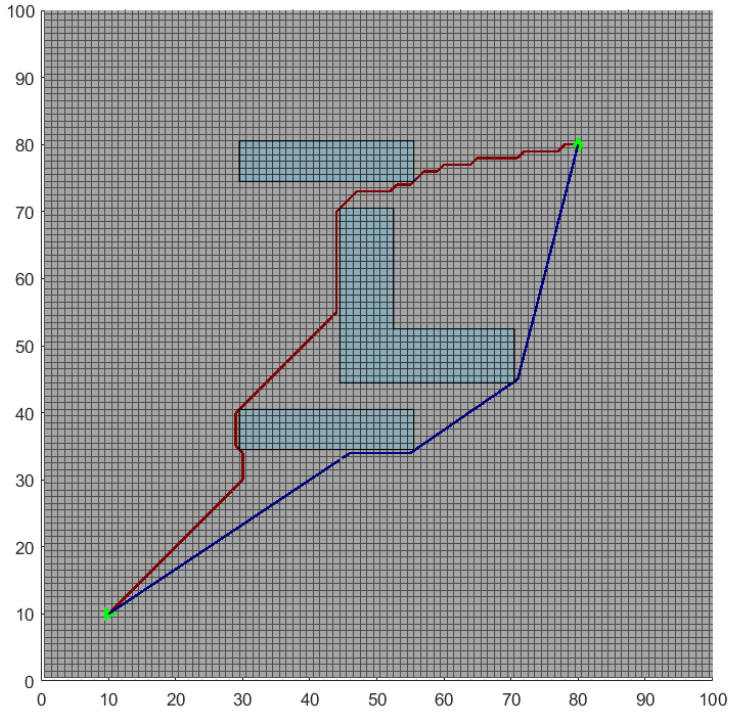

Figure 3.5: 2D comparison between A* and Θ*

#### Comparison with A* (3 dimensions)

Another comparison has been put in place after tridimensionalisation of both algorithms. The simple map was generated so as create a small difficulty for the algorithms, without being visually overbearing. Figure 3.6, where $\alpha$ is set to 2, showcases the staggering difference in the resulting paths. Not as staggering, yet still evident, is the difference in figure 3.7 in which $\alpha = 10$. In both cases, A* finds a 10% longer solutions than the competitor, with a path that looks much more fragmented than necessary, with a large number of useless direction and altitude changes. Between the two iterations, instead, Theta* show a very subtle and irrelevant difference in path lenght and geometry.

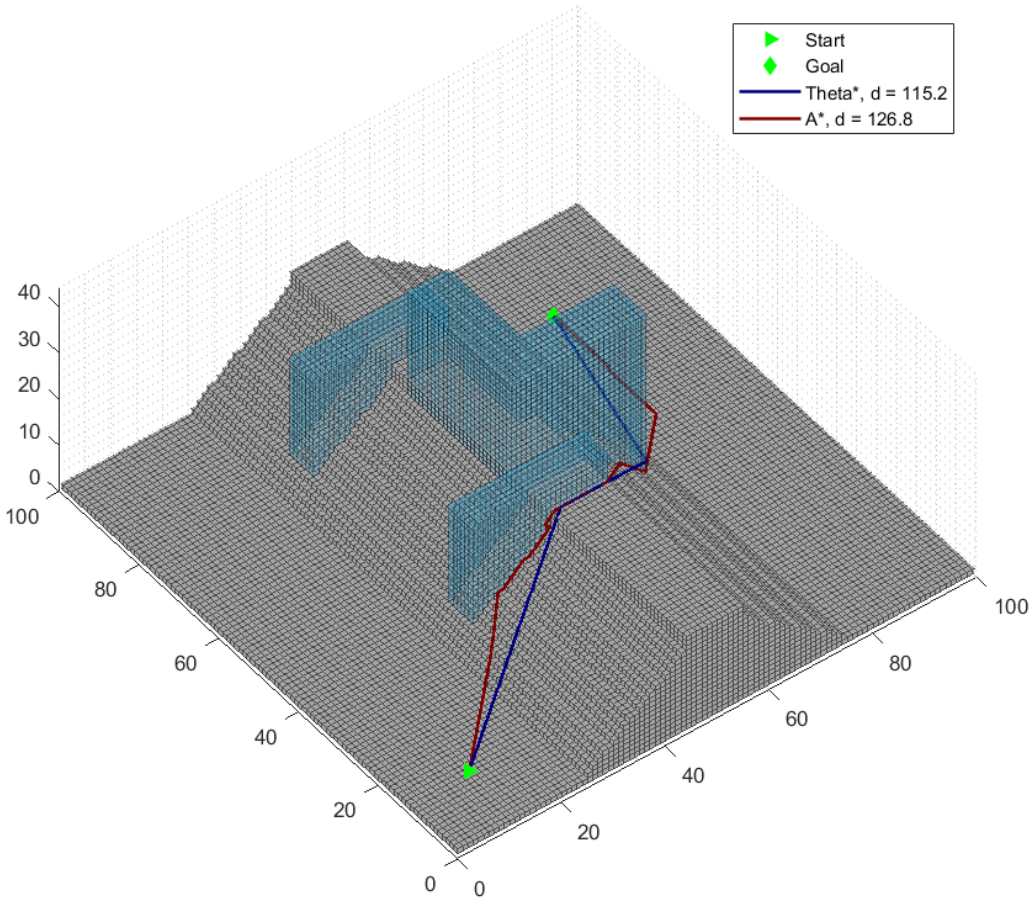



(a) 3D view

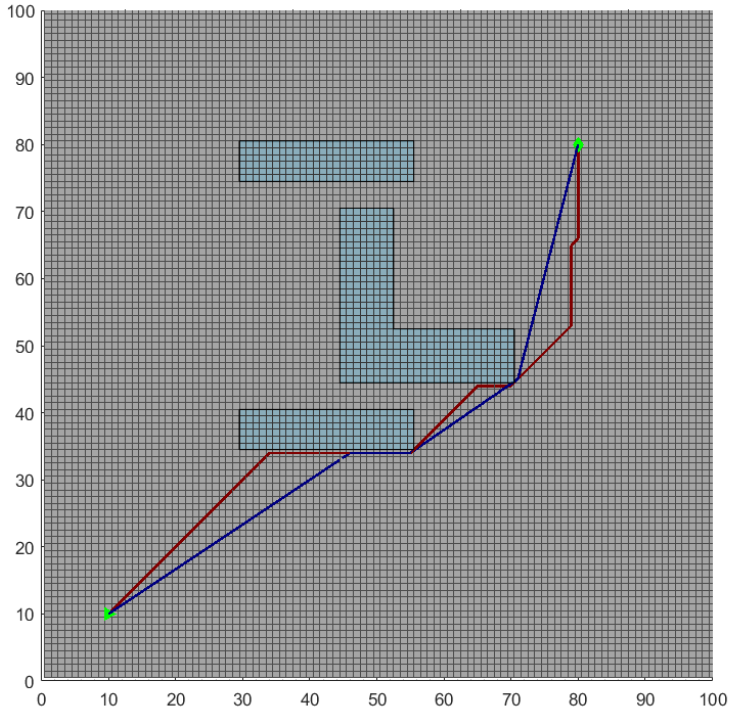



(b) Top view

Figure 3.6: 3D comparison between A* and $\Theta^*$, $\alpha = 2$




(a) 3D view




(b) Top view

Figure 3.7: 3D comparison between A* and $\Theta^*$, $\alpha = 10$

# Part II

# Guidance & Control

# 4: UAS dynamics and implementation

## 4.1 Dynamics

Considering the classical dynamics of an aircraft, whatever its size, six states are considered: three translations and three rotations. However, a quadrotor can influence its own dynamics by means of four control inputs, that is, its rotors' thrusts and torques. Due to the miniaturization involved, the rotors have a fixed geometry, hence the only way to modulate torques and forces is varying each rotor's angular velocity. At this point it is enough to quantify the relation between the i$^{\text{th}}$ rotor speed and its actions on the quadrotor frame as

$$T_i = K_T\ \omega_i^2 \tag{4.1}$$

$$\tau_i = (-1)^{i+1} K_q\ \omega_i^2 \tag{4.2}$$

This relations exclude more complex effects such as blade flapping and aerodynamics, but are sufficient to guarantee a satisfactory model.

### 4.1.1 Flying configuration

The general structure of a quadcopter is made of four arms spanning from the centre and holding one rotor each. They are placed at 90° increments. There are at least two ways to fly a quadcopter: either in a "cross" or in a "plus" configuration, depending on the orientation of the structure and, consequently, on the effect that each rotor has on the dynamics.

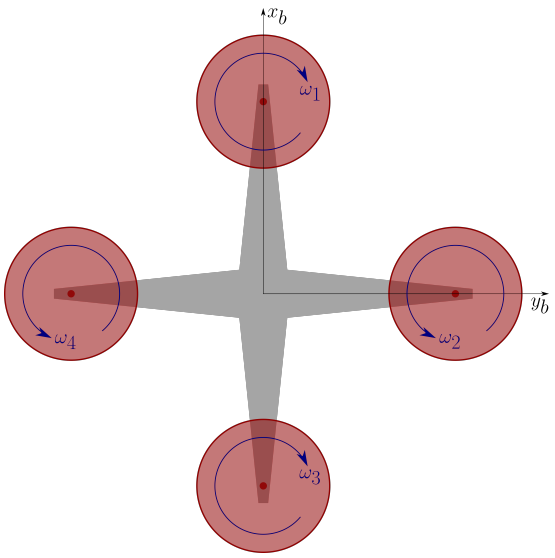



Figure 4.1: "Plus" configuration

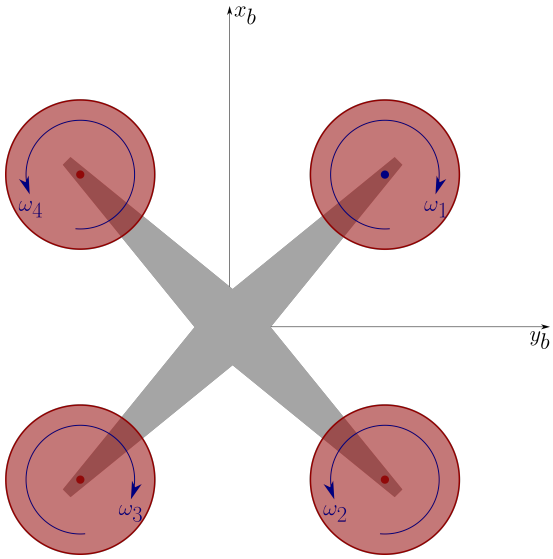



Figure 4.2: "Cross" configuration

**Plus cfg** The "plus" configuration has two of the booms parallel to the body-x axis and the other two parallel to the body-y $y_b$, as in a plus sign "+" or as figure 4.1 illustrates. This is the selected architecture.

**Cross cfg** An alternative is to make the axis a bisector of two contigous arms, so that the vehicle resembles a "×" sign when seen from the top. Of course more exotic configurations exist, but in the scope of the present work are of no interest whatsoever for the time being.

### 4.1.2 Control architecture

As mentioned previously, the controls are represented by the four rotors' speeds, on which the respective forces and moments derive.
A brief note on rotation directions for the rotors: the modulation of the single velocities is required to modulate the thrusts, but reaction torques due to gyroscopic effects arise, causing the system to turn along the z-axis. The most common strategy to avoid this yawing action is to alternate the directions of rotation of the rotors: 1 and 3 are clockwise, while 2 and 4 are counter-clockwise. This is reflected in formulae by the $(-1)^{i+1}$ term in (4.2). The general control

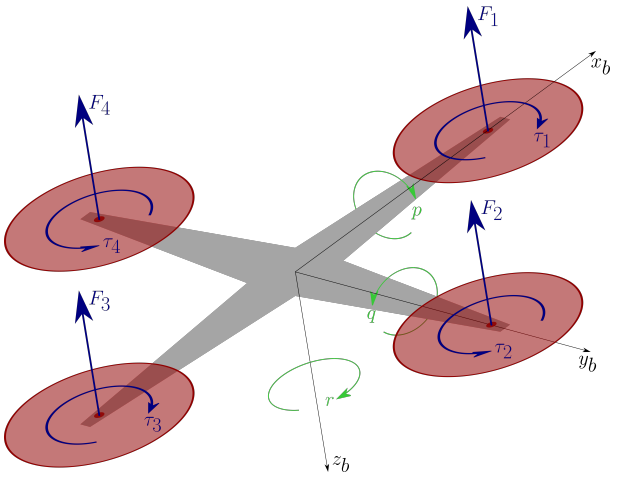



Figure 4.3: Forces and moments

strategy envisions four control inputs depending on the rotor speeds, defined as follows:

**Total thrust** $u_1$ is the sum of all rotor forces. Referring to (4.1),

$$u_1 = T_b = \sum_{i=1}^{4} K_T \omega_i^2 \tag{4.3}$$

Knowing the maximum and minimum rotation speeds for the rotors, the limit control actions can be easily evaluated: in the case of the total trhust, the limits are:

$$u_{1,max} = 4K_T \omega_{max}^2 \tag{4.4a}$$
$$u_{1,min} = 4K_T \omega_{min}^2 \tag{4.4b}$$

**Rolling moment** $u_2$ is the torque generated by the even-numbered rotors, that is the ones placed along $y_b$. They generate a moment along $x_b$ as follows:

$$u_2 = \tau_\varphi = l(-F_2 + F_4) \tag{4.5}$$

Again, the limit controls available amount to

$$u_{2,max} = lK_T \omega_{max}^2 \tag{4.6a}$$
$$u_{2,min} = -u_{2,max} \tag{4.6b}$$

**Pitching moment** $u_3$ a similar equation can be obtained for the moment along $y_b$:

$$u_3 = \tau_\theta = l(F_1 - F_3) \tag{4.7}$$

Due to the symmetry of the vehicle, the pitch dynamics is the same as the roll dynamics, so for the limits equations 4.6 are valid.

**Yawing moment** Moment along $z_b$ is generated by taking advantage of the directions of rotation of the rotors: by using (4.2),

$$u_4 = \tau_\psi = \sum_{i=1}^{4} (-1)^{i+1} K_q \omega_i^2 \tag{4.8}$$

For the limits, we have

$$u_{4,max} = l K_q \omega_{max}^2 \tag{4.9a}$$
$$u_{4,min} = -u_{4,max} \tag{4.9b}$$

The above equations can be gathered in a much more practical matrix form as follows:

$$\mathbf{U} = \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{Bmatrix} = \begin{bmatrix} K_T & K_T & K_T & K_T \\ 0 & -lK_T & 0 & lK_T \\ lK_T & 0 & -lK_T & 0 \\ K_q & -K_q & K_q & -K_q \end{bmatrix} \begin{Bmatrix} \omega_1^2 \\ \omega_2^2 \\ \omega_3^2 \\ \omega_4^2 \end{Bmatrix} \tag{4.10}$$

With a slight abuse of notation, we can further compact (4.10) into

$$\mathbf{U} = [\mathbb{K}]\ \boldsymbol{\omega}^{\mathbf{2}} \tag{4.11}$$

Albeit not modeled, the electric motors powering the rotors are fed a Pulse Width Modulated (PWM) signal with a duty cycle proportional to the desired speed with respect to the maximum speed. Said desired speeds are generated by a Motor-Mixing Algorithm (MMA) which essentially inverts (4.11), taking the control inputs generated by the autopilot and outputting the PWM signal for each motor. In formulae:

$$\mathbf{PWM_{req}} = \frac{1}{\omega_{max}} \boldsymbol{\omega_{req}} = \frac{1}{\omega_{max}} \sqrt{[\mathbb{K}]^{-1} \mathbf{U}} \tag{4.12}$$

### 4.1.3 Preliminary assumptions

In order to simplify the equations, some assumptions have been made:

- the flexibilty of the small structure is not taken in consideration;
- also the rotors are not flexible, so no flapping effect takes place;
- the Earth is considered flat and non rotating, as is often done when such phenomena contribute largely insignificant accelerations of small and relatively slow aircraft;
- the ground effect is negligible, as is the wind;
- the motors are not modeled (yet), and therefore do not introduce delays in the dynamics;
- the inertia matrix $J_b$ is symmetrical:

$$J_b = \begin{bmatrix} J_{xx} & 0 & 0 \\ 0 & J_{yy} & 0 \\ 0 & 0 & J_{zz} \end{bmatrix}.$$

### 4.1.4 Translational dynamics

We will study the motion on the system from an inertial reference frame, that is the classical North-East-Down ("NED" hereafter) triad, which will be noted in the formulae by a "G" superscript for "ground", to which it is fixed. Therefore, $\mathbf{X}^G = \{N, E, D\}^\intercal$ will indicate the coordinates. The dynamics is organized according to Newton's well-known formula:

$$\ddot{\mathbf{X}^G} = \frac{1}{m} \sum \mathbf{F}_i \tag{4.13}$$

The total force in the ground frame is given by the sum of the elements described as follows.

**Gravity** not much is to be said on gravity: it goes down.

$$\mathbf{F_g} = \begin{Bmatrix} 0 \\ 0 \\ mg \end{Bmatrix} \tag{4.14}$$

**Drag** is modeled as proportional to the velocity in each direction

$$\mathbf{F_d} = \begin{bmatrix} k_{dx} & 0 & 0 \\ 0 & k_{dy} & 0 \\ 0 & 0 & k_{dz} \end{bmatrix} \begin{Bmatrix} \dot{X^G} \\ \dot{Y^G} \\ \dot{Z^G} \end{Bmatrix} \tag{4.15}$$

**Total thrust** , or the sum of each rotor' thrust, is aligned with the body z-axis (i.e. $z^b$) and has to be rotated in ground reference, that is:

$$\mathbf{F_T}^G = R_b^G(\varphi, \theta, \psi) \begin{Bmatrix} 0 \\ 0 \\ -u_1 \end{Bmatrix} \tag{4.16}$$

The rotation matrix from body to ground[1] frame is defined as

$$\begin{aligned} R_b^G(\varphi, \theta, \psi) &= (R_x(\varphi) R_y(\theta) R_z(\psi))^\intercal = \\ &= \left( \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix} \right)^\intercal \end{aligned} \tag{4.17}$$

Remember that $u_1 = K_T \sum_{i=1}^{4} (\omega_i^2)$. Also, since we are using NED and the total force is pointing upwards, the minus sign is required.

For completeness, the three contributions are gathered in (4.13) so as to yield:

$$\begin{Bmatrix} \ddot{X^G} \\ \ddot{Y^G} \\ \ddot{Z^G} \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ g \end{Bmatrix} + \frac{1}{m} \begin{bmatrix} k_{dx} & 0 & 0 \\ 0 & k_{dy} & 0 \\ 0 & 0 & k_{dz} \end{bmatrix} \begin{Bmatrix} \dot{X^G} \\ \dot{Y^G} \\ \dot{Z^G} \end{Bmatrix} + \frac{R_b^G(\varphi, \theta, \psi)}{m} \begin{Bmatrix} 0 \\ 0 \\ -u_1 \end{Bmatrix} \tag{4.18}$$

#### Modelization

This subsystem represents equation (4.18). The block `Rb2G` calls a simple MATLAB function tasked with generating the rotation matrix for the body-to-ground reference rotation.
The summation block at the centre generates the acceleration vector $\ddot{\boldsymbol{X}}^{\boldsymbol{G}}$ which is fed to a second-order integrator. The choice for this type of block was driven by the increased solver performance it yields with respect to having two first-order integrators in series, as claimed by The MathWorks.[2] The derivative of the position, $\dot{\boldsymbol{X}}^{\boldsymbol{G}}$ or `dx` as it is labeled on the block, is also output for the generation of the drag forces according to the simplified model of (4.15).

[1] A simple yet effective explanation can be found at http://www.chrobotics.com/library/understanding-euler-angles

[2] See the Simulink documentation page for `Zero-Crossing Detection`.

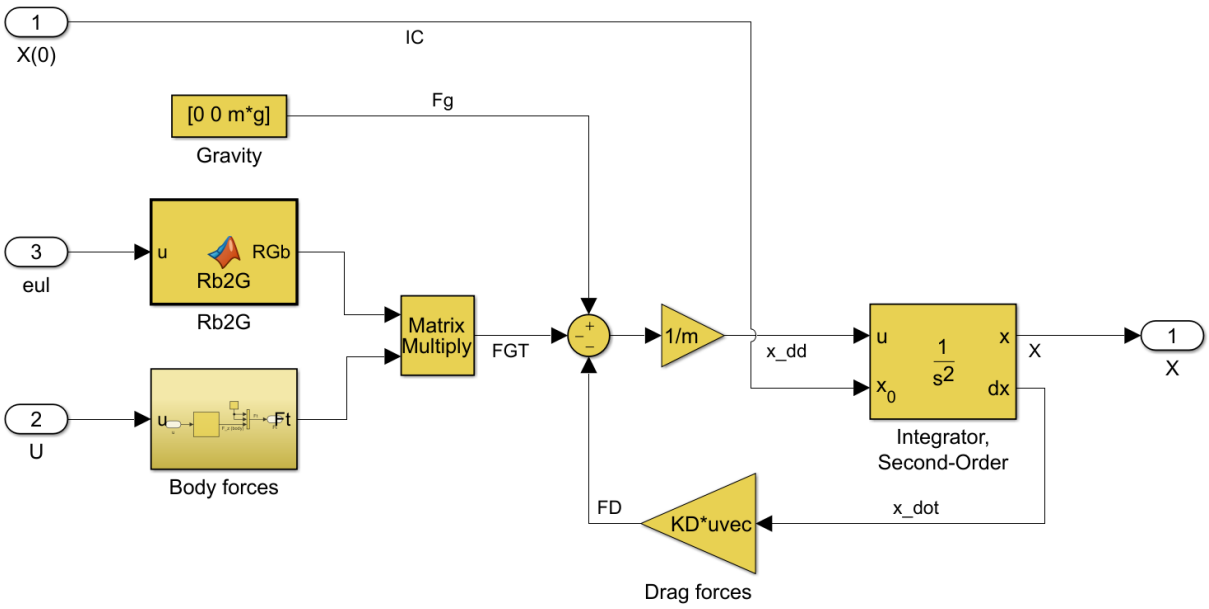



Figure 4.4: Translational dynamics subsystem

### 4.1.5 Rotational dynamics

#### A note on rotational velocities

In a very specific situation where the system is in stable hover and not rotating, it can be asserted that the angular velocities *along each axis*, that is $p$, $q$, $r$, coincide with the derivatives of the Euler angles:

$$\boldsymbol{\omega} = \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = \frac{d}{dt} \begin{Bmatrix} \varphi \\ \theta \\ \psi \end{Bmatrix} \tag{4.19}$$

This makes it very easy to read and use the rates picked up by gyroscopes, which are fixed with the frame of the vehicle.
In reality, this steady-state relation is most often not true and the wider picture is more complicated: we want to study the rotation of the body from a body-centered NED (inertial) frame. Now, to switch from a generic, rotated configuration to NED three consecutive rotations are required, which means that each element of the angular velocities vector will undergo a growing number of rotations through intermediate systems.

Roll does not need any rotation:

$$\dot{\varphi} = p$$

pitch rate must be rotated into the vehicle-1 frame:

$$\dot{\theta}_b \xrightarrow{R_x(\varphi)} \dot{\theta}_1 = q$$

and yaw rate has to be rotated to the inertial frame, so

$$\dot{\psi}_b \xrightarrow{R_y(\theta)} \dot{\varphi}_2 \xrightarrow{R_x(\varphi)} \dot{\psi}_1 = r$$

To explicitate this transformation, we can write:

$$\boldsymbol{\omega} = \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = R_x(\varphi) \left( R_y(\theta) \begin{Bmatrix} 0 \\ 0 \\ \dot{\psi} \end{Bmatrix} + \begin{Bmatrix} 0 \\ \dot{\theta} \\ 0 \end{Bmatrix} \right) + \begin{Bmatrix} \dot{\varphi} \\ 0 \\ 0 \end{Bmatrix} = S \begin{Bmatrix} \dot{\varphi} \\ \dot{\theta} \\ \dot{\psi} \end{Bmatrix} \tag{4.20}$$

where a matrix has been defined to perform rotation:

$$S(\varphi,\theta) = \begin{bmatrix} 1 & 0 & -\sin\theta \\ 0 & \cos\varphi & \sin\varphi\cos\theta \\ 0 & -\sin\varphi & \cos\varphi\cos\theta \end{bmatrix} \tag{4.21}$$

A reader armed with paper and pencil can clearly see how, for zero pitch and roll, $S(\varphi,\theta)$ becomes the identity matrix $\mathbb{I}^3$, and equation (4.20) decays into (4.19).
Of greater interest, however, is the inverse of S, allowing to project a vector of angular rates estimated by sensors on the inertial body frame:

$$S^{-1}(\varphi,\theta) = \begin{bmatrix} 1 & \sin\varphi\tan\theta & \cos\varphi\tan\theta \\ 0 & \cos\varphi & -\sin\varphi \\ 0 & \frac{\sin\varphi}{\cos\theta} & \frac{\cos\varphi}{\cos\theta} \end{bmatrix} \tag{4.22}$$

We can now proceed with the study of the rotational dynamics.

### Moments on the frame

In a similar manner to what has been done for the translational dynamics, the rotational equations can be found as a sum of three contributions:

$$J_b\boldsymbol{\omega} = \boldsymbol{\tau_m} - \boldsymbol{\tau_g} - \boldsymbol{\omega} \times J_b\boldsymbol{\omega} \tag{4.23}$$

**Motor torques** are the torques due to the controls:

$$\boldsymbol{\tau_m} = \begin{Bmatrix} \tau_\varphi \\ \tau_\theta \\ \tau_\psi \end{Bmatrix} = \begin{Bmatrix} u_2 \\ u_3 \\ u_4 \end{Bmatrix} \tag{4.24}$$

**Gyroscopic torques due to propellers** Let $J_r$ be the rotational inertia of the single rotor, $G_z$ the global z-axis versor (i.e. $G_z = \{0,0,1\}^\intercal$) and $\omega_i$ the angular velocity of the $i^{th}$ rotor. Then

$$\boldsymbol{\tau_g} = \boldsymbol{\omega} \times G_z \sum_{i=1}^{4} J_r(-1)^{i+1}\omega_i \tag{4.25}$$

represents the gyroscopic torques generated by the propellers rotation when coupled with the rotation of the frame. Keep in mind that the boldface $\boldsymbol{\omega}$ represents the angular velocities with respect to the frame $p$, $q$, $r$ and that the signs in the sum have to be alternated to account for the different spin directions of the propellers as described in section 4.1.2. The development of the cross product yields

$$\boldsymbol{\tau_g} = \begin{Bmatrix} \dot\theta \\ -\dot\varphi \\ 0 \end{Bmatrix} J_r \sum_{i=1}^{4}(-1)^{i+1}\omega_i \tag{4.26}$$

**Gyroscopic torques due to rigid body rotation** are described by classical, rigid body dynamics:

$$\boldsymbol{\omega} \times J_b\boldsymbol{\omega} = \begin{Bmatrix} \dot\theta\dot\psi(J_z - J_y) \\ \dot\psi\dot\varphi(J_x - J_z) \\ \dot\theta\dot\varphi(J_y - J_x) \end{Bmatrix} \tag{4.27}$$

Again, for the sake of readiness of information, the above formulae are gathered in a single equation, extension of (4.23):

$$J_b \begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = \begin{Bmatrix} u_2 \\ u_3 \\ u_4 \end{Bmatrix} - J_r \begin{Bmatrix} \dot\theta \\ -\dot\varphi \\ 0 \end{Bmatrix} (\omega_1 - \omega_2 + \omega_3 - \omega_4) - \begin{Bmatrix} \dot\theta\dot\psi(J_z - J_y) \\ \dot\psi\dot\varphi(J_x - J_z) \\ \dot\theta\dot\varphi(J_y - J_x) \end{Bmatrix} \tag{4.28}$$

### Modelization

Again in a similar manner to the translational dynamics, the rotational one is modeled on (4.23). Figure 4.5 presents its block diagram. As opposed to its peer subsystem, a second-order

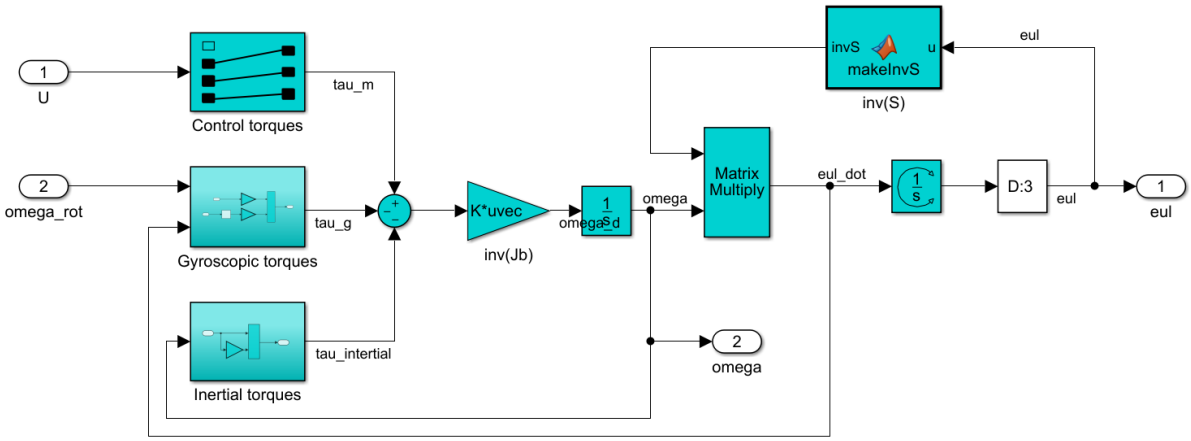



Figure 4.5: Rotational dynamics subsystem

integrator could not be employed because of the rotation operation sandwiched between the two integrations. Said operation performs the transition between body-referenced angular rates and Euler angular rates for the sake of the exportation as a state.

## 4.2 High-level Simulink® implementation

Figure 4.6 shows the high-level construction of the model. The blocks `D:#` are called "signal

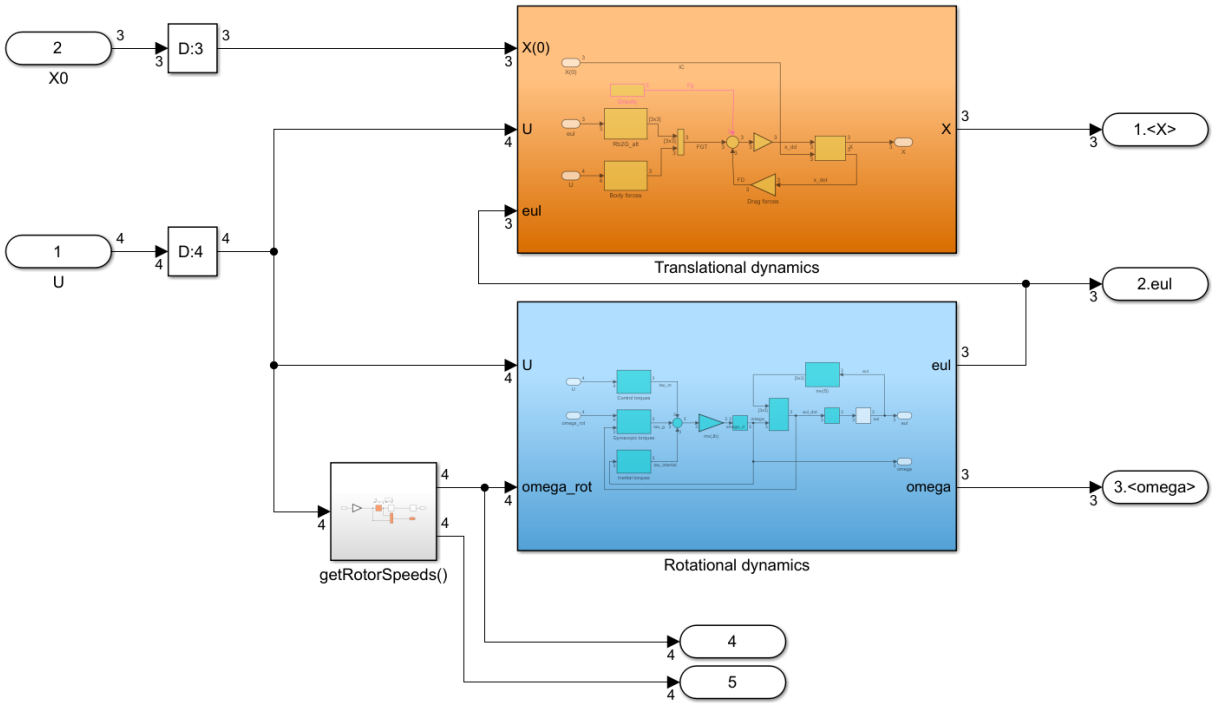



Figure 4.6: Simulink Dynamics

specification" and assure that the signal traversing them has the specified size. This caused

annoying[3] problems at compilation time, when the signals were still having unspecified size and drove other blocks downstream to have unspecified sizes as well. The discovery of the existance of this block was very welcome by the author.

**Inputs**

1. **U** is the control input vector, as introduced in section 4.1.2;
2. `X0` is the initial position vector. Originally introduced for testing, its usefulness will be fully explained in later chapters.

**Outputs**

1. `X` vector, containing the position in the ground frame;
2. `eul` vector, gathering the Euler angles;
3. `omega` vector, with the three angular rates according to body axes ($p,\ q,\ r$);
4. the angular velocities of the rotors, for monitoring purposes;
5. output 5 is another control watch, to be explained in a while.

### 4.2.1 Evaluation of rotor speeds

$$\vec{\omega^2} = \sqrt{\mathbb{K}^{-1}\vec{u}}$$

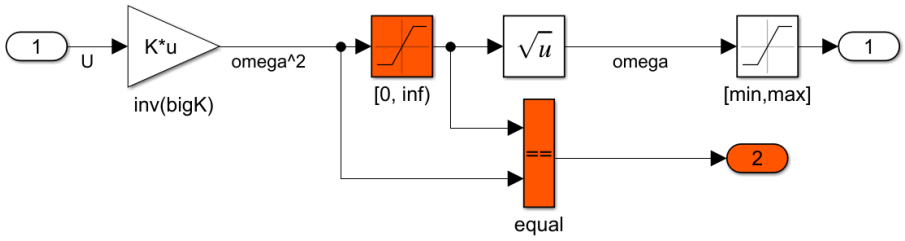



Figure 4.7: Rotor speeds evaluation subsystem

The subsystem[4] labeled with "`getRotorSpeeds()`" implements the reverse relation between the input controls and the rotors' angular velocities expressed by (4.11) and recalled in the annotation. Figure 4.7 shows its working and the constraints imposed by means of two saturations:

- the first limits the argument of the square root to be zero or more in order not to produce complex output that would render the rest of simulation meaningless. Its range is $[0, +\infty)$ for each of the four elements of its input, representing each row of the $\mathbb{K}$ matrix;

- the second keeps the requested rotor velocities between the minimum (0, in this case) and maximum possible values (920 rad/s).

In orange is a comparison between input and output of the first saturation block, which will output a logical `true` if the block is actually clamping any of the values. This result is the previously introduced 5$^{\text{th}}$ output and serves debugging purposes.

[3] **very** annoying

[4] In hindsight, this block could have been included into its only user, which is the "`Gyroscopic torques`" block in 4.5 for the sake of clarity.

### 4.2.2 Model Testing and Validation

Before proceeding with the control system implementation, some tests have been carried out to validate the model. All the runs have been performed across a 10-second time span, without any control whatsoever. In the following figures, the 3D view on the left has the $Z^G$ axis inverted in order to represent more intuitively the trajectory. On the right, instead, it has been kept aligned with the orginial NED triad. The reader is advised to keep this in mind to avoid possible confusion.

#### Free fall

First of all, a free-fall test: setting all the control inputs to zero, the system was expected to accelerate in the positive-z direction (which is down).

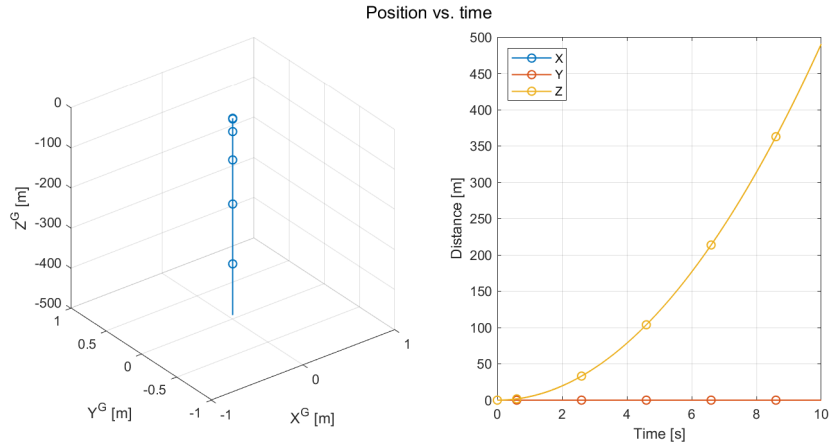



Figure 4.8: Free falling test

#### Hovering

The sole control input is $u_1 = mg$. No other command is given and no disturbing action enters the system, which remains stable.

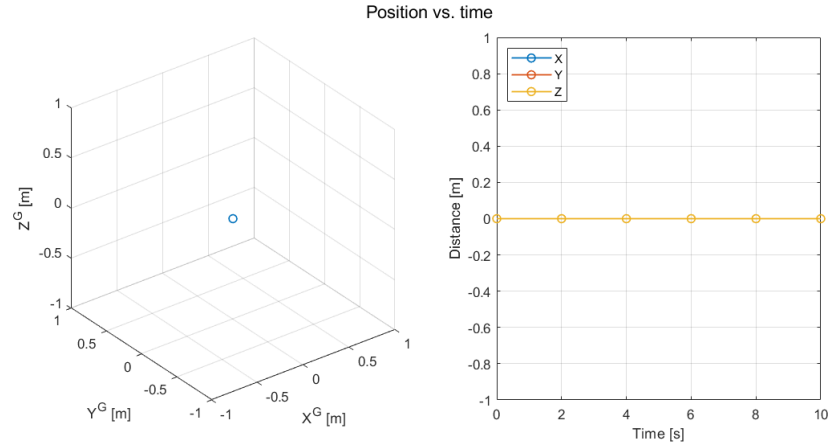



Figure 4.9: Hovering test

#### Small roll angle

In addition to the vertical force command, the initial conditions for the Euler angle integrator have been modified to include a nonzero angle: $\varphi = 5°$. The tilting will produce a sideways force

along the body-y axis, and as a side effect, the system will accelerate downwards because the vertical component of the force is not able to compensate for gravity anymore.

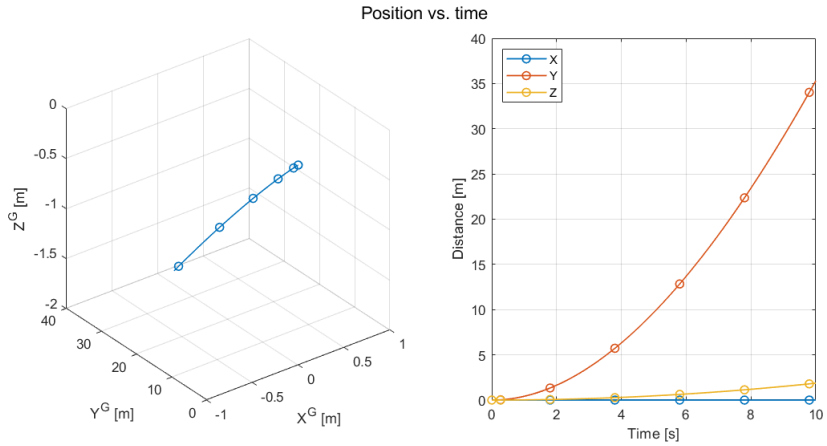



Figure 4.10: Nonzero roll angle

### Small pitch angle, balanced

Under the same conditions as before (except now the pitch is 5 degrees instead of roll, in a poor attempt to make the presentation less boring to the reader), the thrust input has been corrected to account for this angle:

$$u_1 = \frac{mg}{\cos\theta_0}, \quad \theta_0 = 5^\circ$$

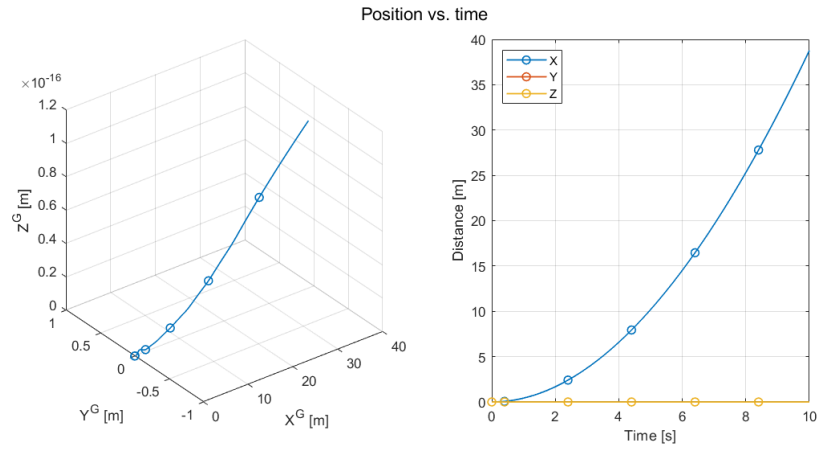



Figure 4.11: Nonzero pitch angle with balancing thrust

### Helix

To conclude, the inputs have been set so as to yield a slight climb coupled with a half-turn yaw with a rate of 10 rad/s:

$$u_1 = 1.05 \cdot mg, \quad u_4 = J_{zz}\frac{\pi}{10}$$

Figure 4.12, right shows the working of the "`wrap state`" option in the Euler angle integrator block (the last to the right in figure 4.5, with a circling arrow around the `1/s` symbol): when the integrated value reaches $\pi$, instead of continuing, it is reset at $-\pi$ and keeps growing from there (or viceversa, of course). This prevents the output of meaningless multiples of $\pi$.

On the left, instead, is the trajectory traced by one arm of the quadcopter: since no movement of the centre of mass is in place (with the exception of the vertical translation), it has been necessary to visualize the trajectory otherwise.

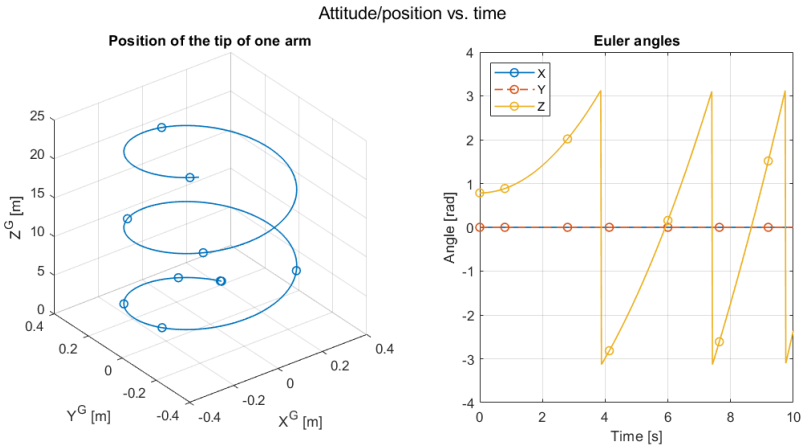



Figure 4.12: Ascending helical trajectory

# 5: Control system structure

## 5.1 Basics of PID control

This section introduces some of the basic concepts that have been useful in the present work. Of course, the reader who is fluent in this subject can skip it without repercussions on their understanding of the chapter.

### 5.1.1 Continous-time formulation

The general concept of PID control is to produce a control output to be fed to the controlled system based on the behaviour of said system with respect to time. The idea can be visualised as in figure 5.1. The output of the controlled process is measured and compared to a reference

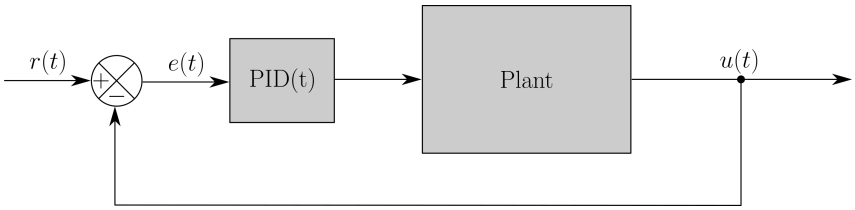



Figure 5.1: Block diagram of basic PID control

signal (the command, or desired output), generating what is referred to as an error:

$$e(t) = r(t) - u(t) \tag{5.1}$$

This error is then evaluated in (a combination of) three ways, each of whose result is multiplied by a dedicated gain and added to the others to produce the control action. Again, this architecture can be visualised in figure 5.2. Those three ways, to whom the controller ows its name, are:

**Proportional (P)** is a measure of the instantaneous error, and its associated gain is $k_p$:

$$P(t) = k_P e(t) \tag{5.2}$$

**Integral (I)** by integrating the error in time we can quantify its history and account for when it has been different from zero in the past:

$$I(t) = k_I \int_0^t e(\tau) d\tau \tag{5.3}$$

The gain $k_I$ scales the relevance of this term when confronted to the other terms.

**Derivative (D)** accounts for the time derivative, hence the rate of change, of the error:

$$D(T) = k_D \frac{de(t)}{dt} \tag{5.4}$$

If we identify the output of the controller with $c(t)$, we can gather equations (5.2), (5.3) and (5.4) as:

$$c(t) = k_P e(t) + k_I \int_0^t e(\tau)d\tau + k_D \frac{de(t)}{dt} \tag{5.5}$$

Not all the actions are compulsory, and in fact they can hinder the quality of the response of the

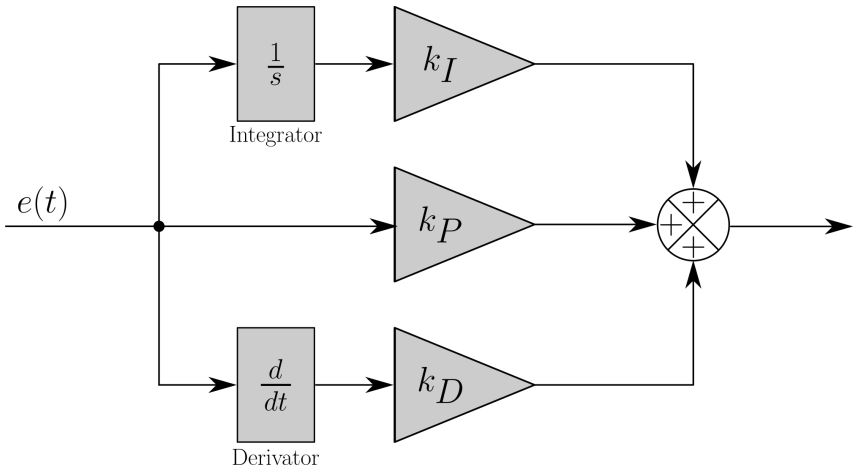



Figure 5.2: Block diagram of a PID controller

controlled system in some cases: for example, a large derivative action will generate an overshoot in the response to a step command, or the extensive use of integral control can produce control actions that saturate the actuator (i.e. require more action than the actuator can generate), with increasingly negative effects on control performance. For this reason, which combination of P,I and D to employ is a decision to be carefully taken in close relation with the system to be controlled.

### 5.1.2 Tuning

The term "tuning" refers to the decision of the gain values to employ. It is not usually an easy task, since the controlled systems are usually nonlinear and therefore require some fixed-point linearized approximation outside of whose validity the controller is not able to perform as wished. Neither the calculation, once the linear model is available, is always intuitive, and often requires trial-and-error on the phisical hardware which can be costly if not downright hazardous: the availability of a digital model on which to perform simulations can greatly reduce the amount of time and resources necessary to tailor the gains to a desired performance. After the approximate gains are found via the simulations, the real controller can be initialised with those values and validated on the hardware with considerable savings, especially in terms of risk since tuning from scratch can bring the system on the brim of instability.

Probably the widest-used tuning method is Ziegler-Nichols, [11], which foresees the setting of all gains to zero, and the gradual change of the proportional term until a gain $k_u$ (or ultimate gain) such that the system shows stable, periodic oscillations in the response. After measuring the period of the oscillations $T_u$, the gains can be calculated as a function of the two parametres. An alternative form to equation (5.5) is the classical form, which makes use of time constants:

$$c(t) = k_P \left( e(t) + \frac{1}{T_i} \int_0^t e(\tau)d\tau + T_d \frac{de(t)}{dt} \right) \tag{5.6}$$

which, when raffronted with (5.5), yields a different way of writing the gains:

$$k_I = \frac{k_P}{T_i} \tag{5.7}$$

$$k_D = k_P T_d \tag{5.8}$$

| | $k_P/k_u$ | $T_i$ | $T_d$ | $k_I$ | $k_D$ |
|---|---|---|---|---|---|
| P | 0.5 | | | | |
| PI | 0.45 | 0.8 $T_u$ | | 0.54 $k_u/T_u$ | |
| PD | 0.8 | | 0.125 $T_u$ | | 0.1 $k_u T_u$ |
| PID | 0.6 | 0.5 $T_u$ | 0.125 $T_u$ | 1.2 $k_u/T_u$ | 0.075 $k_u T_u$ |

Table 5.1: Reference coefficients for PID tuning according to Ziegler-Nichols

The parametres for some architectures can be found in table 5.1.

**Automatic tuning with MATLAB** Since some early attempts to tune manually each controller using the above method have had results of limited quality, the automated tuning application within Simulink has been employed with satisfactory outcomes. Each PID block can access a routine from the Simulink® Control Design™ which linearises the plant connected downhill and suggests the gains based on the simulated response, which the user can fine-tune with respect to rise time and robustness of the control.

## 5.2 Philosophy

The control system relies, to date, solely on PID control. It is organized by concentric control loops, each one providing the setpoint for the next or, eventually, the control inputs for the system itself:

- the outermost provides an attitude request based on the distance from the goal,
- the middle one provides an angular rate request based on the angular distance from said command,
- and the last loop listens to these rate commands in order to generate the actual control inputs $(u_2, u_3, u_4)$ to the plant, which is constituted by the four rotors with their respective motors (the actuators) and the frame of the UAS (the aircraft).

A further controller regulates the altitude by commanding the cumulative rotor thrust $u_1$, without interacting with the others.
At the time of the actual hardware assembly, a fourth array of controllers will be required: the propellers have to be spun at an exact speed, which is calculated by inverting equation 4.11. The input to each electrical motor, which is usually a Pulse Width Modulation (PWM) voltage signal, will have to be generated by another PID (likely only PD, due to the almost instantaneous nature of the rotor dynamics). So far, however, we are not interested in the modelisation of the actuators and therefore will skip the inverse dynamics calculation and rotor speeds estimation, unless strictly necessary as discussed in section 4.2.1.

## 5.3 Structure

Figure 5.3 shows the top-level structure of the controller. The explanation will proceed from the inner loop outward, so as to evolve the augmented model at each step.

### 5.3.1 First Loop

The innermost loop generates the torque commands based on the error between each attitude angle and the commanded received from the second loop:
Figure 5.4a details the lowest-level structure, embedded in the corresponding block of 5.3.
A Proportional + Derivative (PD) architecture has been preferred over the PID configuration in

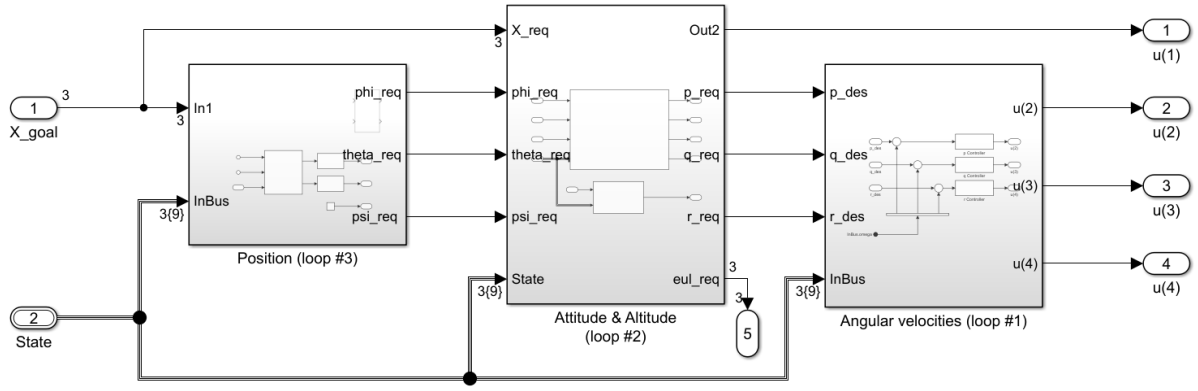
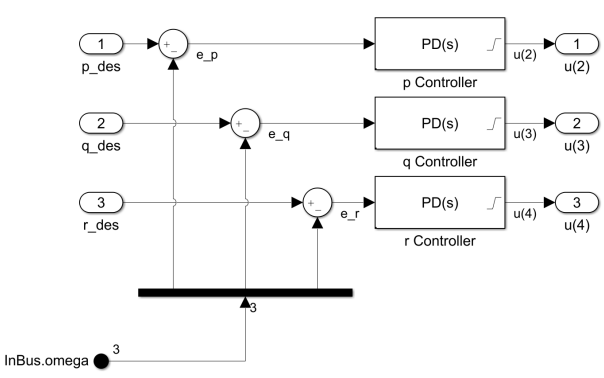
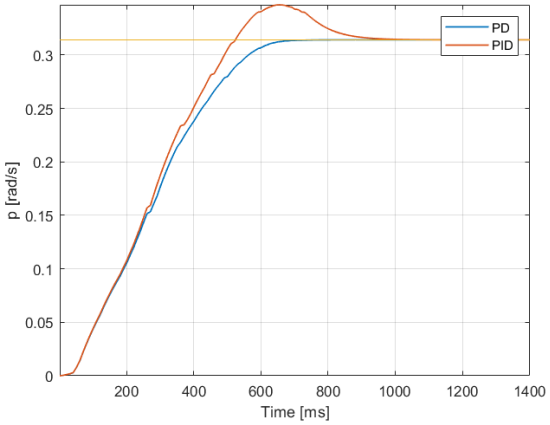



Figure 5.3: Structure

pursuit of a better behaviour: as figure 5.4b clearly shows, the addition of the integral term would cause a different dynamics, with a longer settling time and, most importantly, an overshoot that could interfere with the precision of the control in the wider picture.




(a) $p, q, r$ control loop




(b) PD vs. PID comparison

Figure 5.4: Angular rates control loop

The comparison has been run for the roll dynamics ($p$), but since the system is symmetrical in all respects the same conclusion works for the pitch dynamics ($q$) as well. Furthermore, due to the similarity in the rotational behaviour as can be understood from the previous chapter (Section 4.1.5 in particular), these considerations also apply to yaw ($r$).

All the three controllers have been saturated to the minimum and maximum amounts allowable for each command, as calculated through equations (4.4), (4.6) and (4.9). The resulting limitations, along with the controller coefficients, are summarised in Table 5.2.

### 5.3.2 Second Loop

The middle loop controls the attitude based on the reference given by the position control and the altitude based upon the reference altitude. See figure 5.5 for the block diagram.

#### Attitude

Figure 5.6a shows the inner working of the attitude control. The structure is exactly the same from 5.4a, and given how intimately connected, though hierarchically superior to it it is, this will

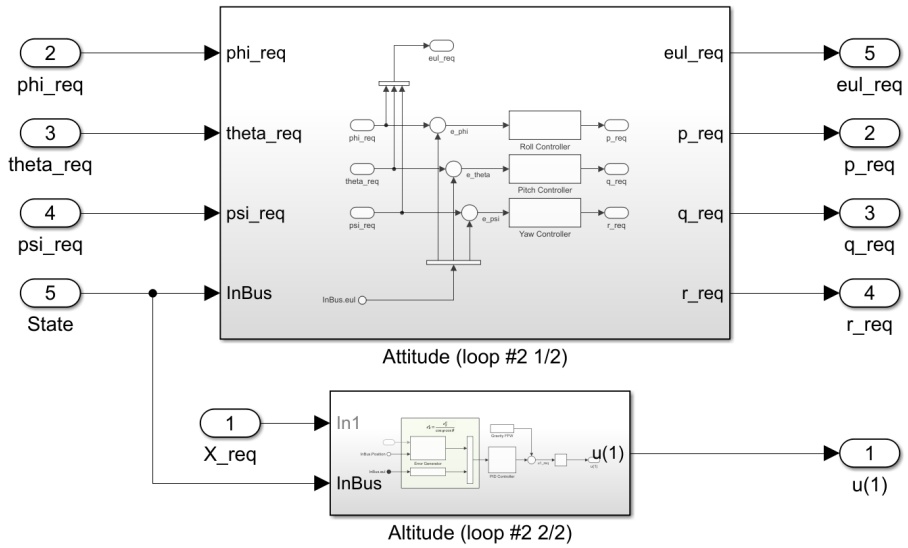
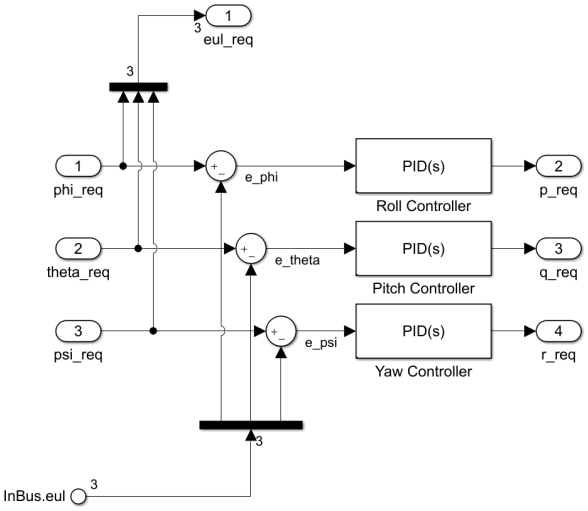
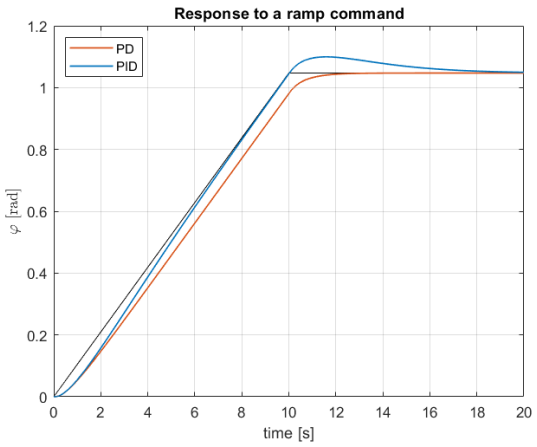



Figure 5.5: $\varphi, \theta, \psi$ control loop

not come as a surprise. The only difference is one outport gathering the requested Euler angles and exporting them for plotting.




(a) Attitude control loop




(b) Respone to a ramp roll command, with/out integral control

In fact, the same considerations about wether or not to employ Integral control have arisen. This time though, the decision is put off until the presentation of the next loop: check out ramp at 5.6b shows the difference.

### Altitude

In figure 5.7 is the inner working of the altitude control. There is no particular reason to prefer the collocation of the altitude control in any loop. In fact, it could have been placed outside of the three loops: in any case, it does not interact with any of the other controllers. The controller sports a slightly different structure to the ones seen until now: the block labeled "Error Generator" receives the current position from the InBus port and the position requirement from inport 1. The reason to gather this small step into a further subsystem was to hide some minor algebraic blocks to avoid cluttering the view more than necessary. The task of the block, however, is to take the third element of each input (which are both vectors) and subtract them, so as to have

the altitude error in the ground frame:

$$e_Z^G(t) = Z_{req} - Z^G(t)$$

The next step is the rotation in the body frame, since the action is fixed to it. For this reason, the third input brings in the system the Euler angles vector, from which $\varphi$ and $\theta$ are taken:

$$e_Z^b(t) = \frac{e_Z^G(t)}{\cos\varphi\cos\theta} \tag{5.9}$$

The transformation is also annotated in the model.

Another interesting aspect is the feed-forwarding of gravitational action: by adding it downhill of the controller, we can actually rid it of a control action that is always present and focus it on the control of the altitude. Figure 5.8 shows the performance of the controller in the two situations with response to a step command. Finally, the output is saturated to prevent requesting excessive

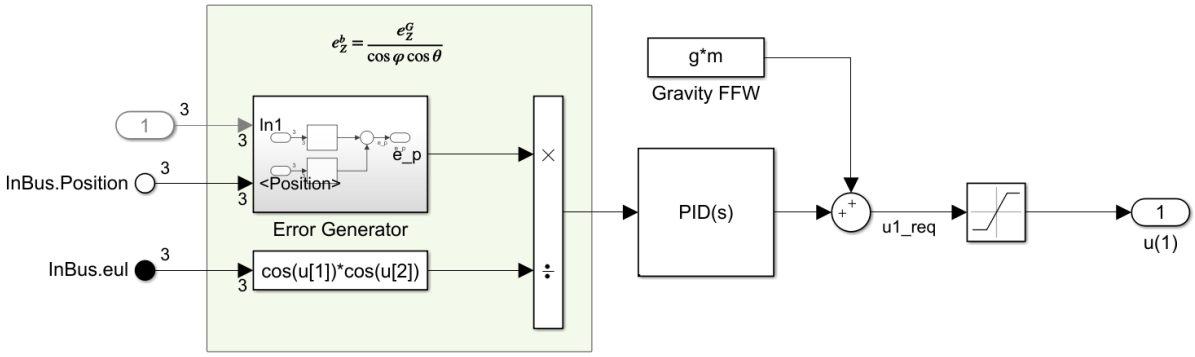



Figure 5.7: Altitude control

actions which could not be delivered by the motors. The formula is (4.4) and the results, along with the coefficients, are given in table 5.2.

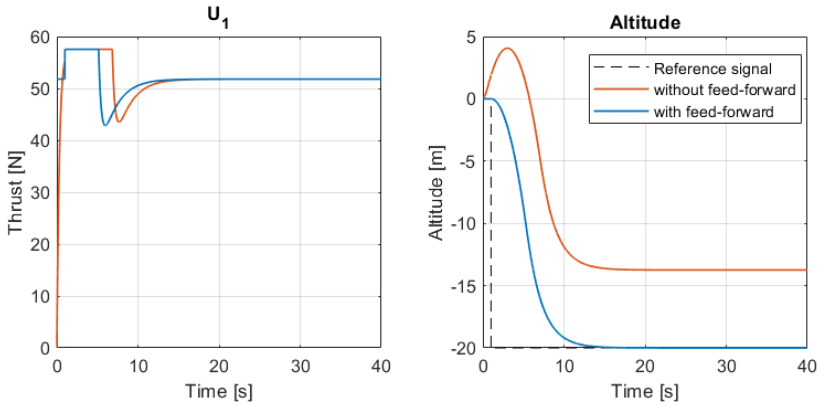



Figure 5.8: Altitude controller performance with and without gravity feedforward

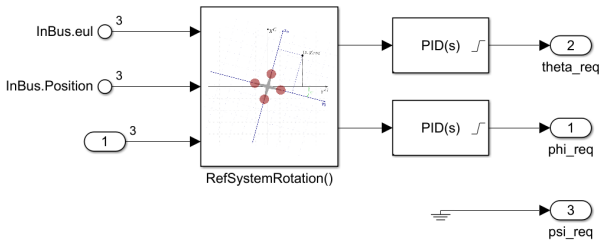
### 5.3.3 Third Loop

The last, outermost loop is tasked with the generation of pitch and roll commands based on the X-Y position error, so as to tilt the rotor thrust and drag the system toward the destination. Since the motion is tracked in the ground reference and the control forces act on the frame, a rotation needs to be put in place, as done in the altitude control:

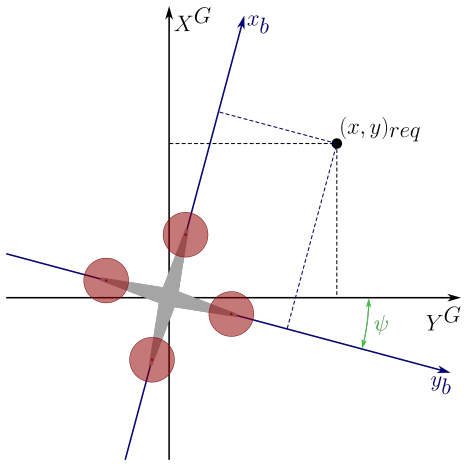
$$e_{X,Y}^b(t) = R_z(\psi)e_{X,Y}^G(t), \qquad e_{X,Y}^G(t) = \begin{Bmatrix} X_{goal} - X^G(t) \\ Y_{goal} - Y^G(t) \end{Bmatrix} \tag{5.10}$$

This is the reason why figure 5.9a shows an input port carrying the Euler angles vector, and the block `RefSystemRotation()` there depicted executes exactly (5.10). Figure 5.9b is the mask carried by the subsystem, visualising the need for the rotation: the $X, Y$ components of the position error evaluated along the ground frame are different from the ones considered through the body frame.

Again, table 5.2 contains information on this controller. There is also a limit on the output: this prevents the system to be tilted excessively, either sideways or longitudinally, resulting in excessive speeds. While having both $\varphi$ and $\theta$ constrained to $\pm 5°$ for the presented work, the system shown acceptable results even when limited to $\pm 25°$.




(a) Position control




(b) Rotation

### Heading control

From inspection of figure 5.9a, it is easy to notice how the heading command signal, $\psi_{req}$, is grounded as in an electrical scheme. In the Simulink notation, this block generates a zero constant signal. An attempt to build a better control logic (or a control logic at all) has not produced any significant advantage at the time of writing and is left for future work. However, the idea is to instruct the $\psi$ controller in loop #2 to direct the system head-on toward the next waypoint: the direct consequence is that the position error with respect to the body frame is almost entirely along the body-x axis, letting the X-translation (pitch) controller to work on reducing the distance on its own, and using the Y (roll) controller almost exclusively for course-correction actions. The flight dynamics, therefore, would resemble more closely that of a fixed-wing system.
In this situation, the goal $\psi$ would be calculated as the arctangent of the error components in the body frame:

$$\psi_{req} = \arctan\left(\frac{e_Y^b}{e_X^b}\right) \tag{5.11}$$

However, as stated above, this feature is not yet implemented because of numerical issues in the simulation to be further investigated. It doesn't seem unlikely that the implementation of discrete control logics will allow the simulation to be more favourable in this respect.

| | P | I | D | min | max |
|---|---|---|---|---|---|
| $p = q$ | 0.194 | | 8.330e-3 | -4.46 Nm | 4.46 Nm |
| $r$ | 0.471 | | -0.339e-2 | -1.485 Nm | 1.485 Nm |
| $\varphi = \theta$ | 1.571 | 0.423* | 0.594 | | |
| $\psi$ | 2.087 | 0.839* | 0.284 | | |
| $Z$ | -8.282 | -0.748* | -18.193 | 0 | 57.54 N |
| $X = Y$ | 1.002e-3 | 1.115 | 9.434e-3 | -5° | 5° |

Table 5.2: Relevant coefficients for the angular rates control loop

# 6: Complete model and simulation

Figure 6.1 shows how the model and controller have been put together and joined with blocks called `WptSelector`, which works as illustrated below, and Output which exports the results of the simulation to the MATLAB workspace for plotting and examination.

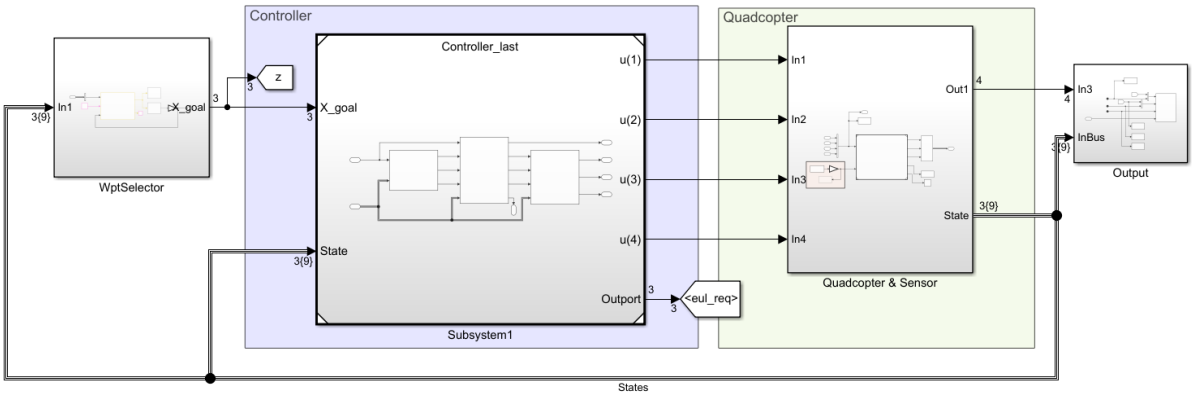



Figure 6.1: Quadcopter model

## 6.1 Waypoint selection

The `WptSelector` block, illustrated in figure 6.2, receives the waypoint sequence calculated by the algorithm of part I as input and outputs the current waypoint for simulation. The output is held constant until the system is within a predetermined distance from the waypoint, at which point an internally-stored index is incremented by the `getIndex()` block and used as a key to extract the next set of coordinates from the lookup table loaded with the sequence. When the index reaches the last point, that is when it equals the number of rows of sai table, the simulation stops. The initial index is 2, since the first waypoint has been set to the starting position and

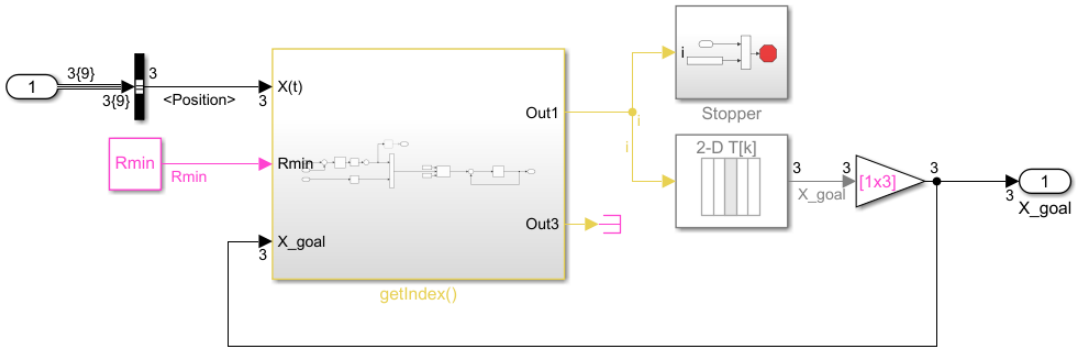



Figure 6.2: The waypoint selection block

its coordinates used to initialize the phisical model.

For all the next results, $R_{min}$ has been set to 1 metre.

### 6.1.1 Waypoint enrichment

As will be seen in the next section with the single-waypoint simulation, the overshoot on position is quite large if the guidance is only based on the calculated waypoints: large position errors will cause large overshoots and hinder the precision of the command. To overcome this problem, each path segment has been enriched with more, secondary waypoints interpolated between the two extremes to make sure guidance is smooother and the overshoots in position are reduced. For all the next simulation runs, the "secondary" waypoints are distributed every 5 metres.

## 6.2 Simulation

### 6.2.1 Single-waypoint tests

The first test to be carried out has been a single waypoint at $X, Y, Z = 20, 30, -12$, starting from the origin of the reference $(0, 0, 0)$. The trajectory is visible along with its projections in figure 6.3 and the time history of the simulated variables, namely coordinates, Euler angles, control inputs and angular rates, are plotted in figure 6.4.

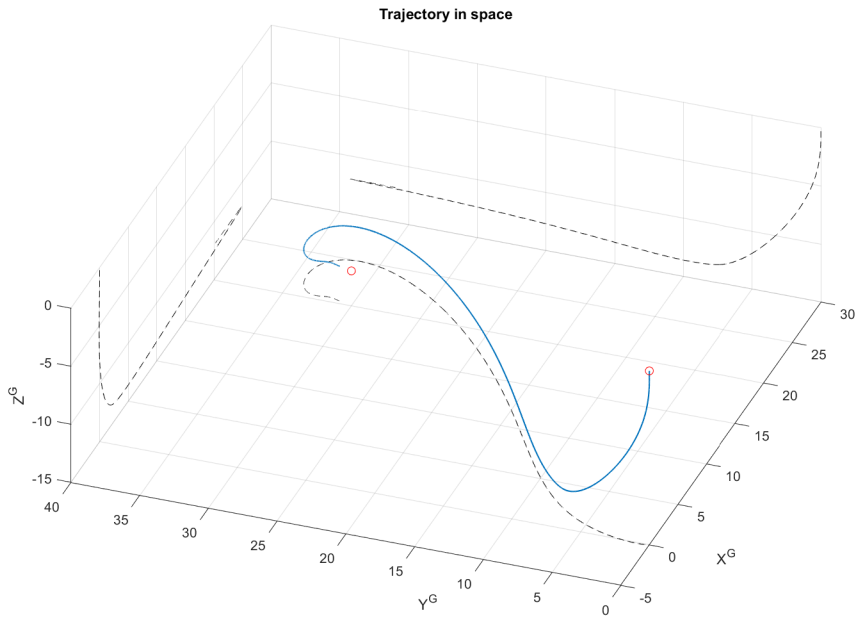



Figure 6.3: Trajectory for a single-waypoint simulation. The dashed lines are the trajectory projection on each plane, for ease of viewing.

### 6.2.2 Large number of points without waypoints enrichment

Figure 6.5 shows a simulation between 102 random points ran without this smoothing routine. Note how, once at each point, the altitude of the next is reached in a much shorter time compared to the other coordinates, risking collision with terrain objects. In the absence of a collision avoidance system, the importance of a thicker, more homogeneous distribution of waypoints is made evident especially from the last segment, where the trajectory enters the terrain exactly for this reason.

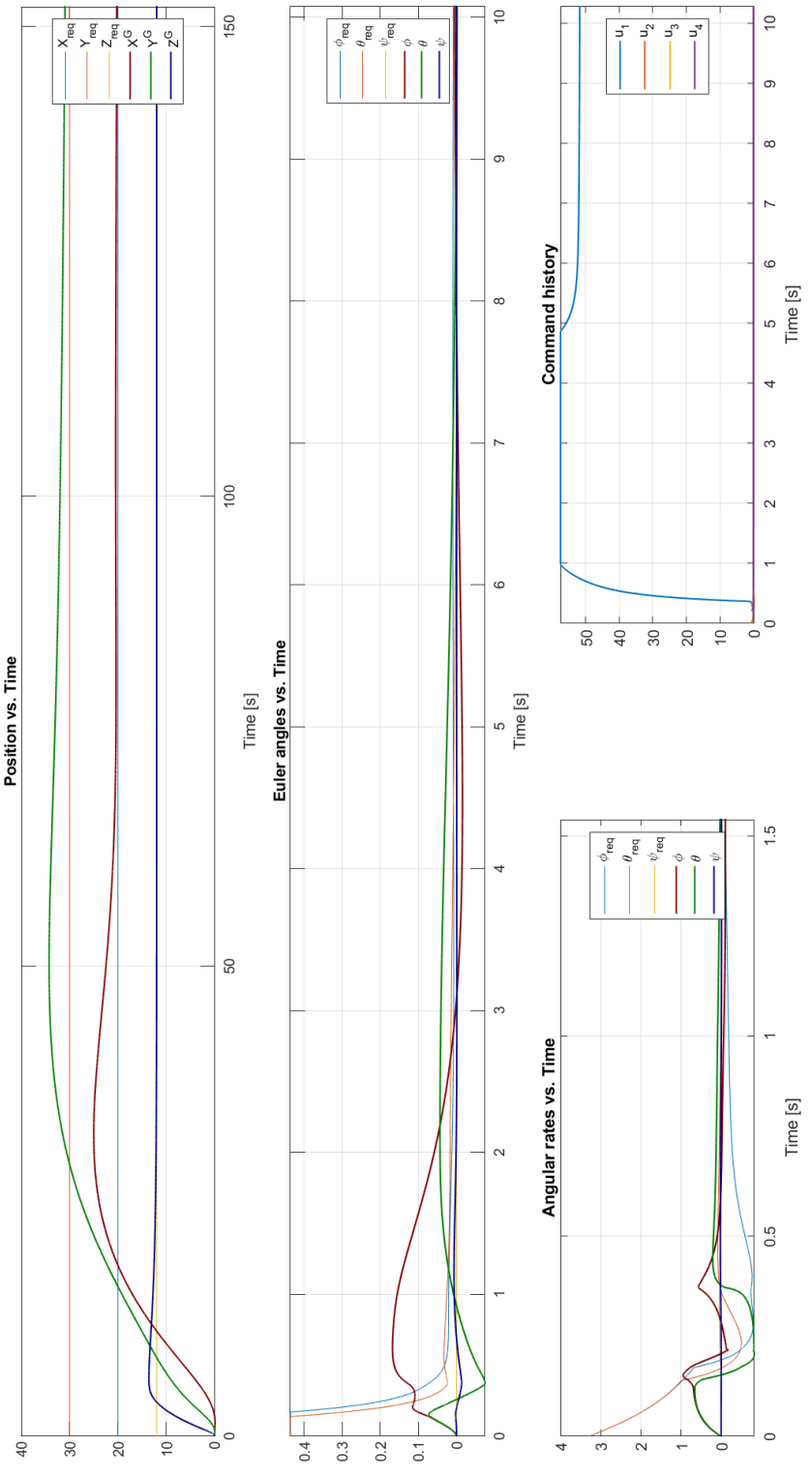
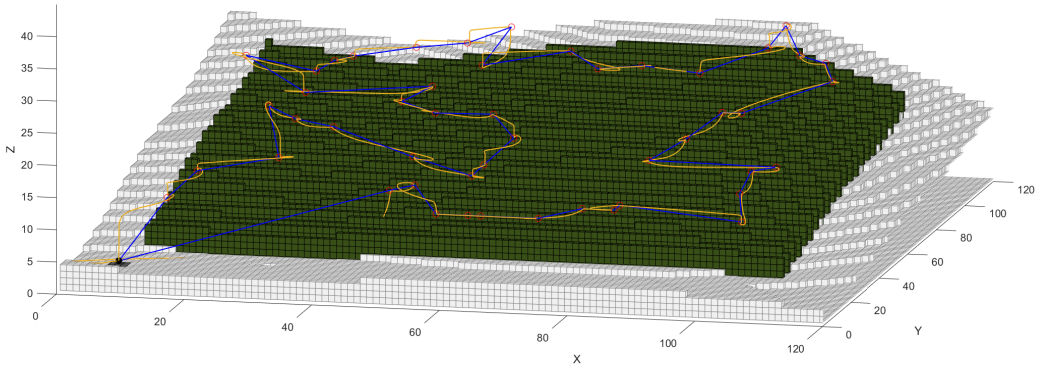
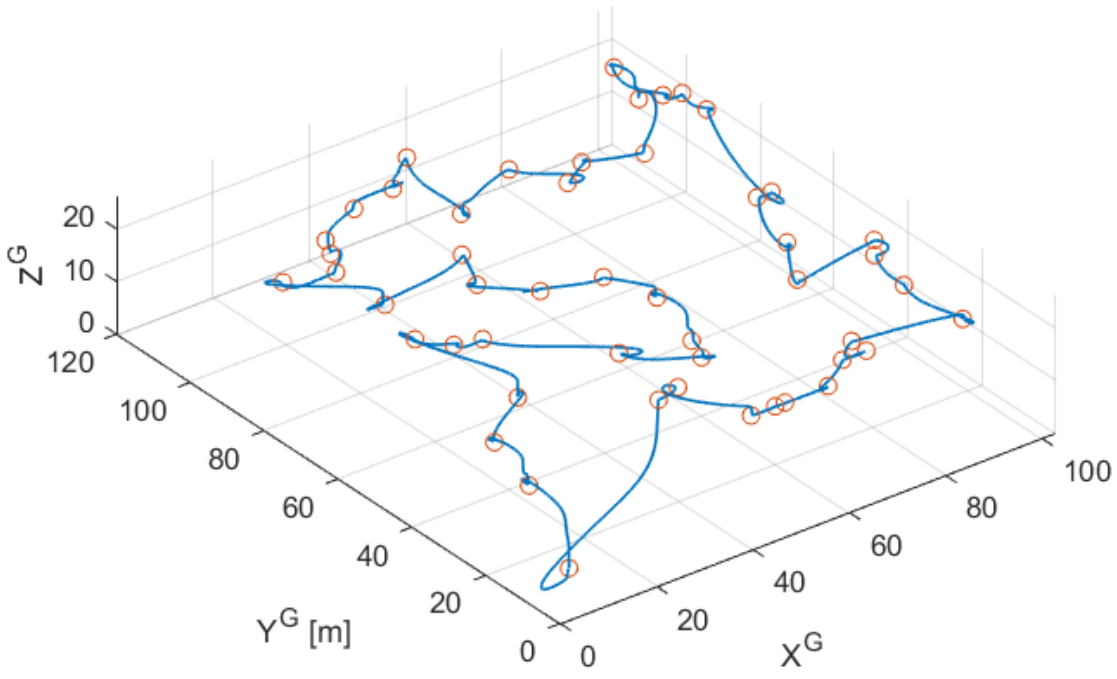



Figure 6.4: Time history of relevant simulation parametres for a single waypoint test. The plots have been zoomed in time in order to present some relevant information




(a) The 102-point path projected on the vineyard. Notice how the trajectory often collides with the ground.

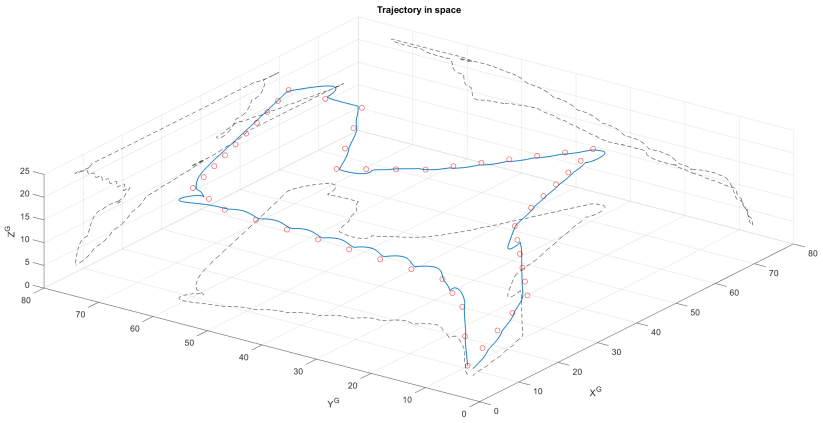



(b) The 102-point trajectory

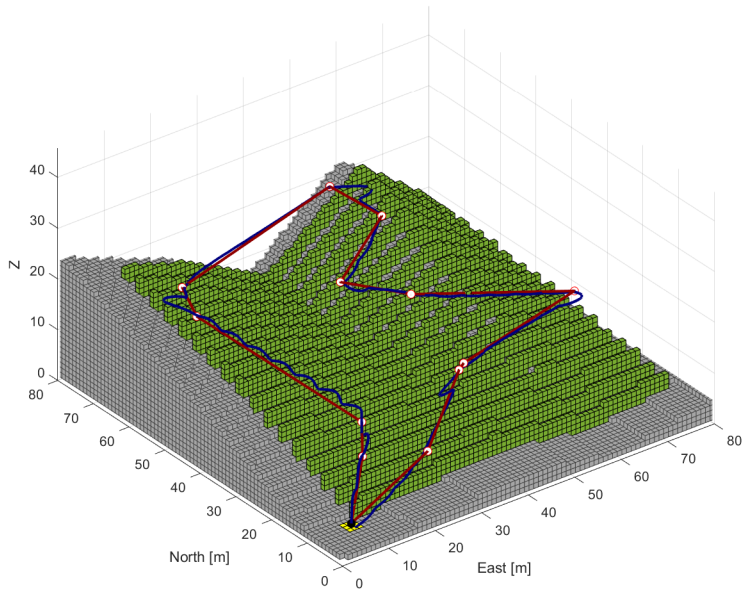
Figure 6.5: First Complete test with 102 waypoints

### 6.2.3 12-waypoint case

The system has been simulated on the same map and waypoints as of chapter 2. What follows is the 12-point scenario, one waypoint every 5 metres yielded satisfactory results and will be seen in figures 6.6 for the trajectory and 6.7 for the simulation parametres as made for the single-waypoint case.




(a) Trajectory of the 12-waypoints scenario. The dashed lines are the trajectory projection on each plane, for ease of viewing.




(b) 12-waypoints simulation traced on the vineyard

Figure 6.6: Trajectory and visualisation of the 12-waypoints scenario.

Note how the trajectory is still presenting some ripples due to overshoot and spikes in the angular velocities caused by the derivative actions of the respective controllers. Chapter 7 suggests some strategies to overcome these imprecisions.

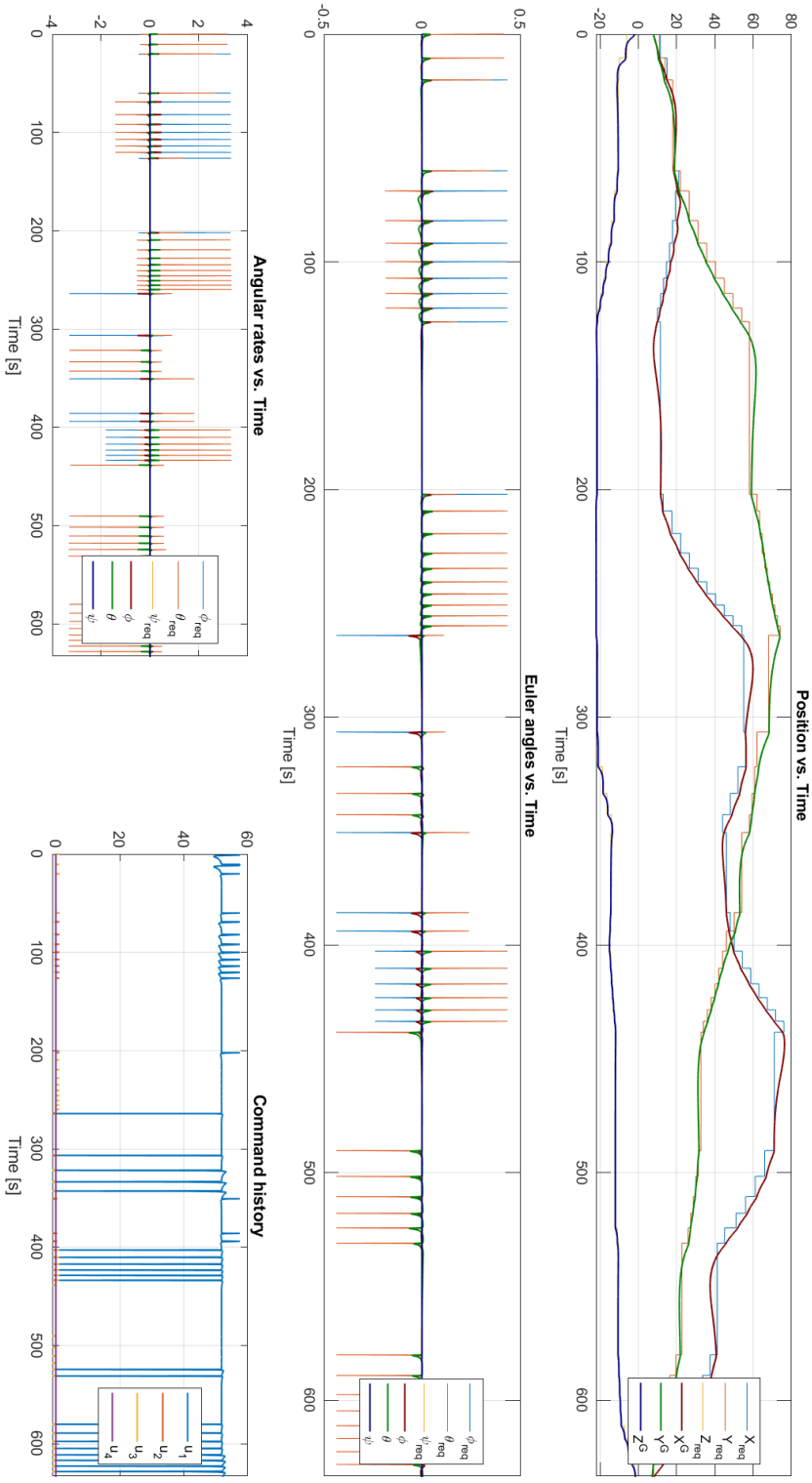
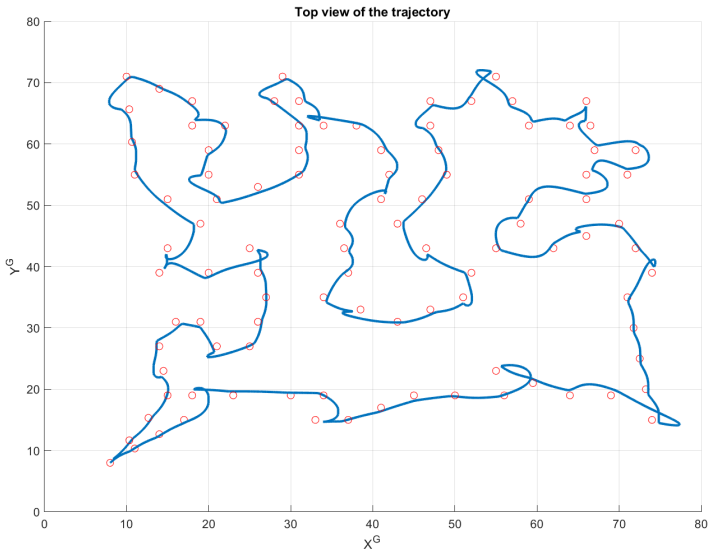


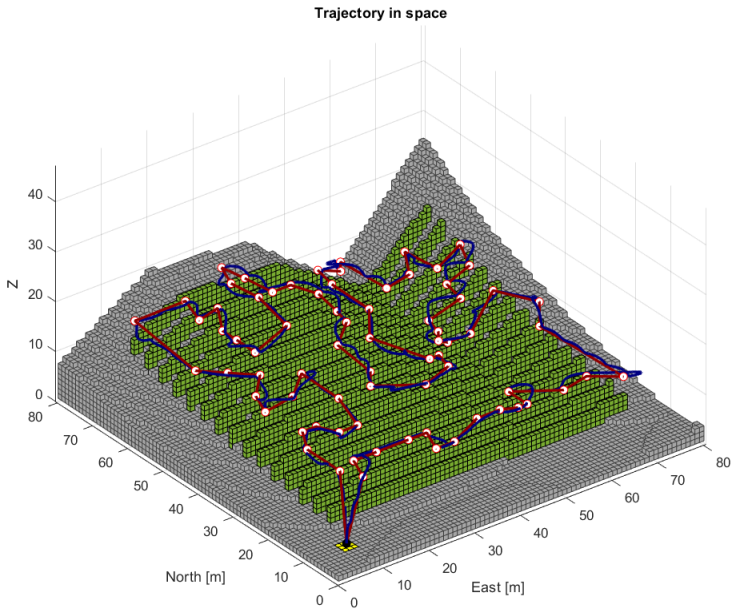

Figure 6.7: Time history of relevant simulation parametres for the 12-waypoint test. The plots have been zoomed in time in order to present some relevant information

### 6.2.4 80-waypoint case

A further simulation has been carried out, as done before, on the second example of cahpter 2. On the 80-point scenario, again, a secondary waypoint has been added every 5 metres. Results can be seen in figures 6.8 for the trajectory and 6.9 for the simulation parametres as made for the previous cases.




(a) Trajectory of the 80-waypoints scenario. The dashed lines and 3D view have been dropped in favour of a clearer trajectory top view.




(b) 80-waypoints simulation traced on the vineyard

Figure 6.8: Trajectory and visualisation of the 80-waypoints scenario.

It can be noted how the minimum radius is too large in this case for the system to actually reach the goal points before being driven to the next, and the 5-metre discretisation is comparable to the average waypoint distance, causing again some imperfections in the control action. This calls for a more intelligent way of smoothing the trajectory that can tailor the amount and distance of secondary waypoints on the situation.

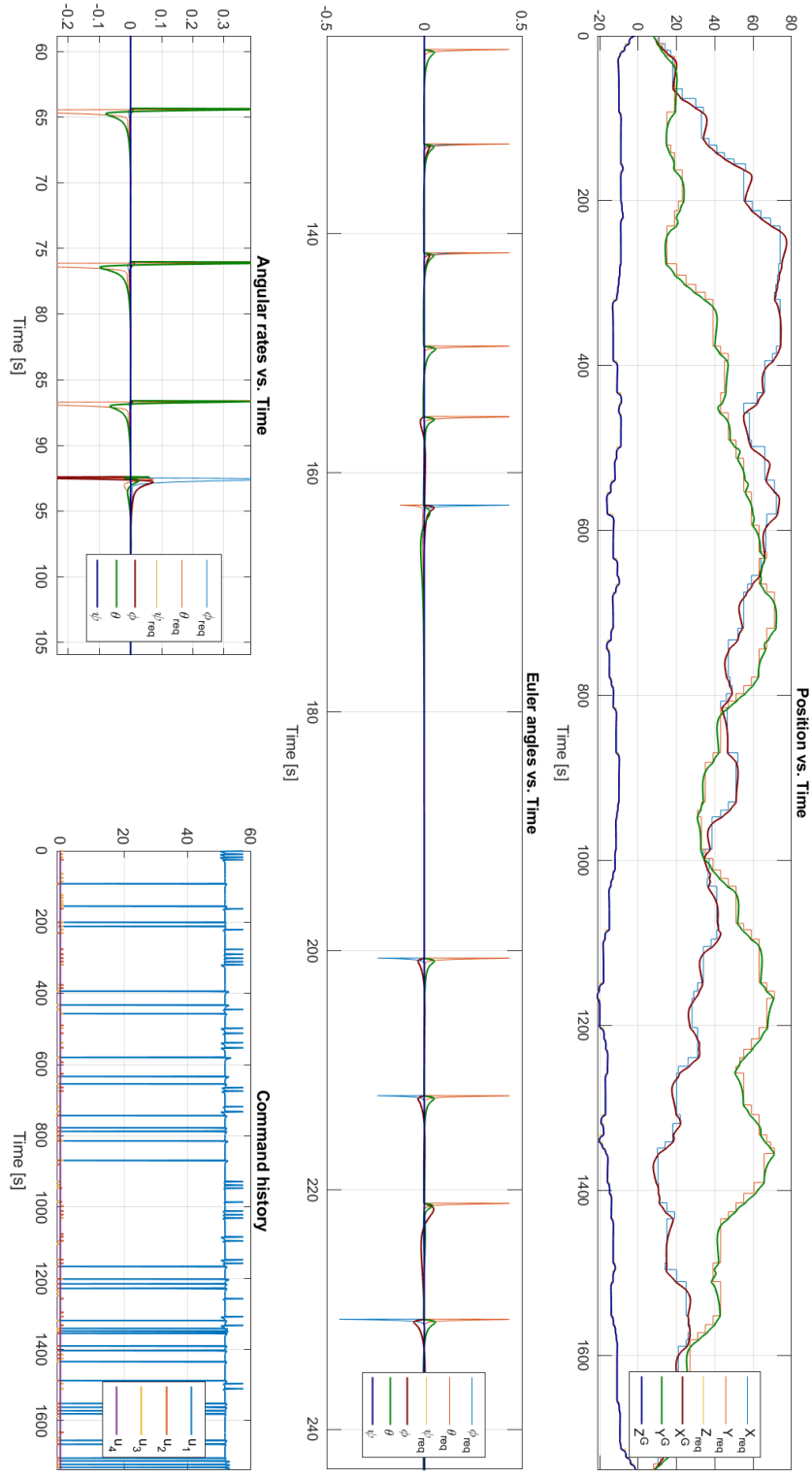



Figure 6.9: Time history of relevant simulation parametres for the 80-waypoint test. The plots have been zoomed in time in order to present some relevant information

### 6.2.5 Double grid test

As a last test, a double grid similar to those traveled for field scanning has been built. Each grid element is 400-by-70 metres and the grid is traveled twice: once forward at 50 metres altitude, then returning toward the origin at 25 metres. Some overshoot can be seen at the corners, but the performance overall is satisfactory for the moment.

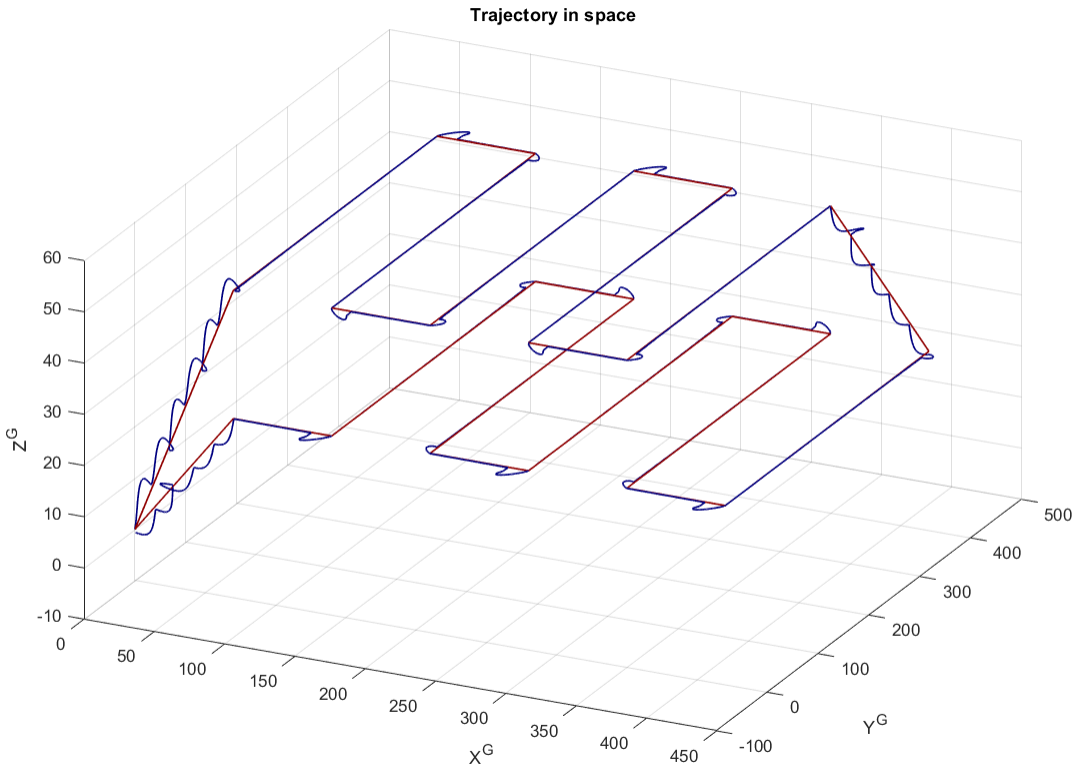



Figure 6.10: Trajectory of the double grid simulation

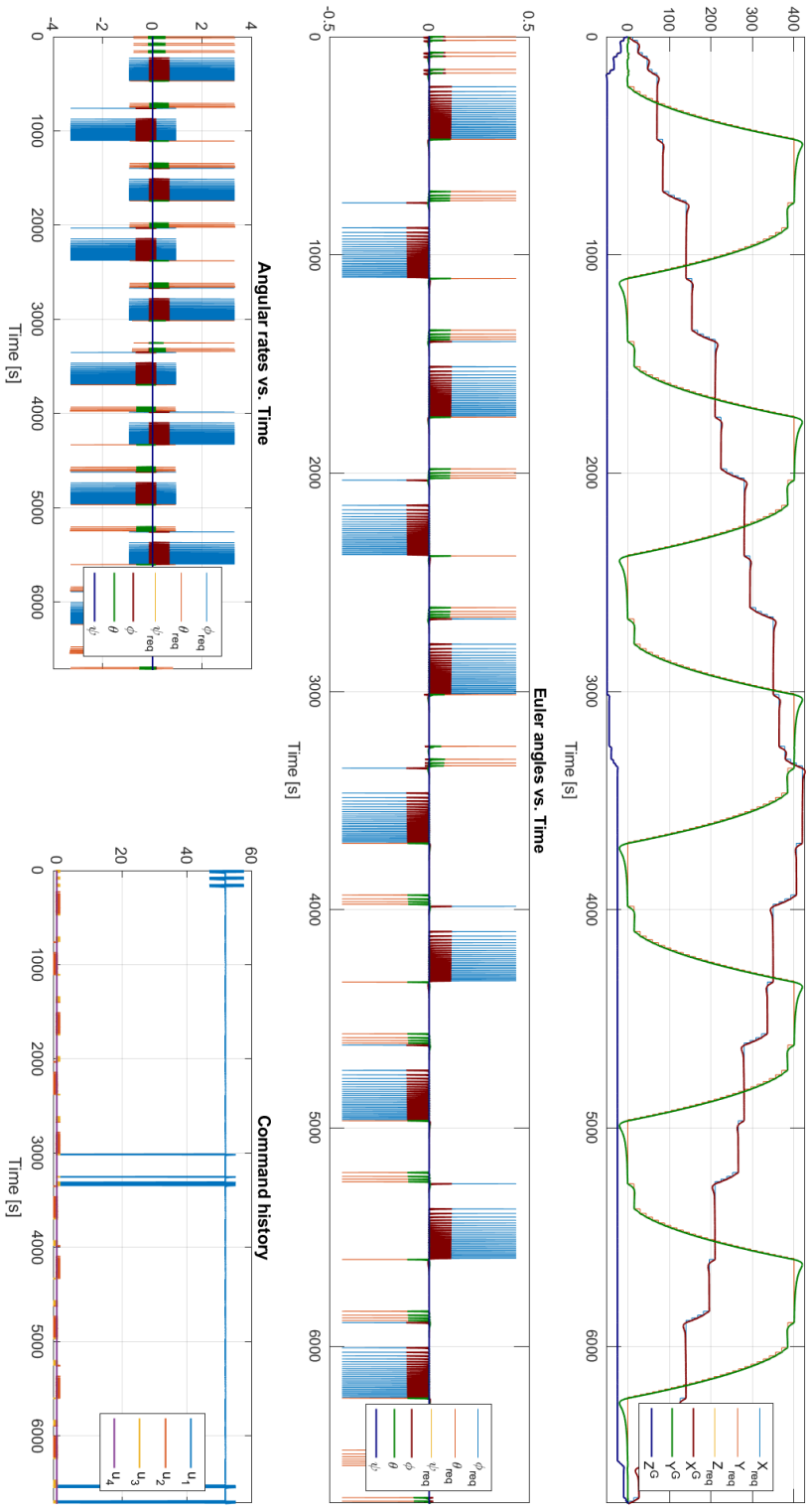



Figure 6.11: Time history of relevant simulation parametres for the double grid test

# Conclusions and Future Work

# 7: Conclusion and Future works

## 7.1 Path planning

In the first part of this work, a way has been devised to take the shortest path through the sequence of waypoints and to generate feasible path segments while avoiding obstacles. It is precisely this latter task, the path planning, which requires some careful *labor limae* both in terms of precision and of computational requirements: the search through the nodes requires that the whole flight domain is memorised, albeit in a simplified and approximate manner, in the planning tool which most likely will be the ground control station (GCS) of the system. Not only is this impacting the memory usage, but it also requires a fairly recent, detailed and up-to-date scanning of the field and surroundings which is fortunately already carried out by the plant-examination platform. Still, the need for a RADAR or LIDAR system is necessary on board for the avoidance of collision with unpredicted objects.

With regard to path planning algorithms, [10] introduces an interesting philosohpy, called Kinematic A*, which foresees the implementation of a simplified aircraft model (the same as for Dubins aircraft) inside the algorithm itself. This allows to replace the grid notation with a "state" notation: instead of looking for the best node as next move, the object of the search is the combination of commands on the aircraft, among the possible ones, that will move the system in the most favourable state. This is done by itegrating the equations of motion from the model in small, fixed amounts of time, and it is worth noting that due to the inclusion of the flight dynamics, wind can also be taken into account, which was impossible with Theta*.

While satisfactory for the moment, the implemented iteration of Theta* has another drawback bound to LOS checking, as remarked in the dedicated section 3.3.1: some of the nodes can be flagged as free of objects, and thence flyable, while actually obstructed, depending on how much they are crossed by the line connecting the two nodes. A stricter checking routine has to be employied, either with a finer grid which would in fact collide with the memory requirements, or with a channel-drawing routine making use of many iterations of Bresenham to draw (or check), instead of a simple line, a minimum-clearance-radius tube that would account for a safe distance from obstacles and for the size of the system itself.

Also, still according to [10], obstacle separation and command optimization are directly embedded by this procedure, enabling them in a simpler manner that would otherwise require post-processing.

One of the potential spinoffs of this methos is that also the heading at each waypoint can be prescribed more easily, as is done with the constrained heading problem introduced in [6], which is also about Dubins vehicles: in case of wind on the field, the correct heading can be selected not just to prevent the uncontrolled scattering of the spray by the wind, but to exploit the wind itself to direct the spray on the target correctly.

Finally, the use of Dubins paths can also avoid the need for path smoothing routines to be executed as post-processing on the results of Theta*.

## 7.2 Guidance and Control

In the second part of the work, a satisfactory model of the quadrotor UAS has been implemented and simulated in different situations before building and simulating the control architecture. Some ideas and extensions have been left out due to time constraints, but are documented in the following for the sake of further development.

**Sensor dynamics & Navigation** The modelisation of sensor dynamics yields a more realistic model, where electrical noise and external disturbances picked up by the sensors can make the game a little harder for the control system and direct its development toward a more field-ready software. The usual sensor package for a UAS is made of:

- a satnav system for navigation;
- an Inertial Measurement Unit (IMU) comprising gyroscopes and linear accelerometers for the precise determination of attitude and velocities;
- a barometer for altitude estimation;
- a magnetometet for magnetic North determination;
- laser/radio sensors for the determination of ground clearance and obstacle avoidance.

**Discrete control** The control structure runs on phisical hardware and is therefore constrained to run discretely at some submultiple of the hardware clock. This is not necessary a disadvantage, as running each control loop at an increasingly high frequency (proceeding inwards) allows the progressive augmentation of the system without the overall performance becoming too "nervous". The frequent update of the outer loops risks to feed highly oscillating commands that could undermine the performance of the downstream controllers (think about how the derivative action responds to an error signal changing even slightly in a short time). It is probably sufficient for the current application to have the position loop run at 10 Hz, the attitue at 100 Hz and the angular velocity at 1 kHz.

**PID upgrades** The classical PID can be improved in some aspects. As stated above, the controllers always produce a peak in the command when either the setpoint or the measurement changes suddenly, due to the nature of the derivative action. This can be limited by limiting the derivative output (either a saturation or a rate limiter in Simulink, for example) or by having it act only on the measurement. This improvement is called Derivative on Measurement (DoM) and was found along with interesting others in Brett Beauregard's Project Blog[1]. While not yet tested on the system, some of these modifications look promising and worthy of a deeper analysis.

**Velocity limitation** Limiting the movement speed of any robotic system in populated environments is important in terms of safety and comfort of the bystanding operators. Once some kind of speed sensor has been implemented (it could also be simply reading the state derivatives during the simulation), the limitations can be enforced at the controller level with different strategies, whose envisioning has not been object of the present work.

**Heading guidance** This topic has been introduced in section 5.3.3 and will render the flight performance more like that of a fixed-wing UAV, with smoother control actions at the advantage of energy consumption.

**Investigate other control strategies** Other control strategies, chiefly for high-level position control , can be enforrced and might allow for online trajectory modification in situations of need such as potential collisions. An example could be Model Predictive Control [12], but again during the span of the present work no further strategies have been examinated.

[1] http://brettbeauregard.com/blog/2011/04/improving-the-beginners-pid-introduction/

# Bibliography


[1] L. Comba, A. Biglia, D. Ricauda Aimonino, and P. Gay, "Unsupervised detection of vineyards by 3d point-cloud uav photogrammetry for precision agriculture," *Computers and Electronics in Agriculture*, vol. 155, pp. 84 – 95, 2018.

[2] D. S. Johnson and L. A. McGeoch, "The traveling salesman problem: A case study in local optimization," *Local search in combinatorial optimization*, vol. 1, no. 1, pp. 215–310, 1997.

[3] G. Gutin, A. Yeo, and A. Zverovich, "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the tsp," *Discrete Applied Mathematics*, vol. 117, no. 1-3, pp. 81–86, 2002.

[4] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of mathematics*, vol. 79, no. 3, pp. 497–516, 1957.

[5] M. Owen, R. Beard, and T. McLain, *Implementing Dubins Airplane Paths on Fixed-Wing UAVs**, pp. 1677–1701. 08 2015.

[6] P. Váňa and J. Faigl, "The dubins traveling salesman problem with constrained collecting maneuvers," *Acta Polytechnica CTU Proceedings*, vol. 6, pp. 34–39, 2016.

[7] S. LaValle, *Planning Algorithms*. 01 2006.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[9] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids,"

[10] L. De Filippis, G. Guglieri, and F. Quagliotti, "Path planning strategies for uavs in 3d environments," *Journal of Intelligent & Robotic Systems*, vol. 65, no. 1-4, pp. 247–264, 2012.

[11] J. Ziegler and N. Nichols, "Optimum settings for automatic controllers," *Transactions of the A.S.M.E.*, vol. 64, p. 759–768, 1942.

[12] J. Rawlings and D. Mayne, *Model Predictive Control: Theory and Design*. 01 2009.