UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Supervised Learning for Sensorless Wake Sensing in Formation Flight

Dissertation presented by
Alban LEFÈVRE

for obtaining the Master's degree in
Electro-mechanical Engineering

Supervisor(s)
Philippe CHATELAIN, Julien HENDRICKX

Reader(s)
Jean-Charles DELVENNE, Marco RUIZ , François WIELANT

Academic year 2017-2018

Abstract

Today, the use of drones in civil and military applications will only increase; many more drones are therefore expected to be airborne in the next years. One main drawback of existing drones is their operational range. The energy saving benefits of formation flight have been known for some time, but could not yet be applied to civil aircrafts due to obvious safety requirements. Fixed-wing drones are, on the other hand, less safety-sensitive, and could be the first field in which formation flight becomes applied to extend their range. The main problem to positioning an aircraft in a wake is called *wake sensing*. The goal of this master's thesis is to use artificial intelligence to perform wake sensing without additional sensors. In order to accomplish this, an existing open-source simulator is first modified to add the physics of the wake generated by the leader aircraft. Then, a simplified simulation environment is used to develop a proof of concept that artificial intelligence algorithms have to potential to help sensing the wake of the leader.

Contents

List of Figures

Acronyms

AI Artificial intelligence

ANN Artificial Neural Network

BFS Breadth-first search

CSP Constraint satisfaction problems

DFS Depth-first search

ESA European space agency

FOL First-order logic

KB Knowledge base

NN Neural network

PL Propositional logic

SITL Software in the loop, name of the simulator used in the context of this work

Symbols

α angle of attack

a_0 Lift-curve slope

AR aspect ratio

β sideslip angle

b wingspan

Γ circulation

γ circulation per unit length

c wind chord

$\bar{c}$ mean chord

C_L lift coefficient

C_D drag coefficient

C_{D_0} parasitic (or zero-lift) drag coefficient

δ_a aileron deflection

δ_e elevator deflection

δ_r rudder deflection

ρ air density

s wing surface

U_∞ airspeed

U_{eff} effective airspeed

w downwash

w_u upwash

Chapter 1

Introduction

In a world where petrol slowly becomes a scarce resource with rising prices, it might seem illogical that the demand in air transport of both passengers and freight keeps increasing. The current forecasts of the International Air Transport Association (IATA) about the evolution of air transport indicate a doubling of the amount of passengers in the next 20 years [12] and a yearly growth of $4,5\%$ of the air freight demand for the next five years [19]. This increase in demand, associated with the rising prices of fuel, will require that air transport becomes more and more efficient. Efficiency has indeed become one of the main concerns when designing a new type of aircraft. Airplanes currently in development by the leading manufacturers (for example, the Boeing 777x) focus on bringing new levels of fuel efficiency to airlines.

In parallel to this increasing need for "traditional" air transport, the development of drones provide new means of transport of air cargo, which could fulfill the ever increasing demand in home delivery after online purchases. After all, the drones logistics and transportation market is expected to reach $11,2$ billion USD in 2022 and $29,06$ billion USD in 2027. It is not by accident that Amazon, the current world leader in e-commerce, is studying a new program of drone delivery (Amazon Prime air). In the world of drones too, the focus is slowly being made on efficiency. Not because of the price of fuel, but to increase the range of these drones. That's why, next to the "traditional" multi-copters, Amazon Prime Air develops fixed-wing drones as well as hybrids, who can take-off and land vertically while flying like a fixed-wing drone, therefore increasing the in-flight efficiency.

Other than the technical improvements of airplanes and drones to reach a higher level of efficiency, by introducing more aerodynamically efficient shapes or better engines, one could also study the improvements that could be done in the organization of the flight itself. Today, for example, when airline pilots plan their flights, they study the weather previsions not only to avoid potentially dangerous conditions but also to save fuel, namely by adapting their altitude to benefit from tailwinds whenever possible [3]. But there is another way, well known to migratory birds, to save energy in flight: **formation flight** (see figure 1.1). Because of air traffic safety rules, it is unlikely that airplanes will be allowed to fly in close formations in the near future. Extended formation flight, where the aircrafts are miles apart could be safely considered, but is not yet used in commercial flights. Drones, however, might profit from close formation flight, as they are unmanned vehicles.

1.1 Interest of formation flight

To understand the interest of formation flight, it is necessary to go back to the basics of aerodynamics. When an airfoil move through a stream of air, local changes of velocity appear [6]. By the theorem of Bernoulli, a change of speed in a fluid generates a change in pressure.

Figure 1.1: Birds flying in V-formation

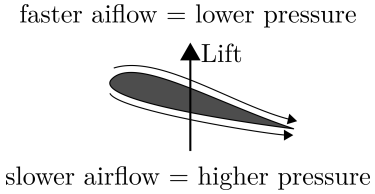


Figure 1.2: Pressure changes with speed of the airflow, generates lift

In normal flight, when the aircraft cruises along while battling against gravity, the speed above the airfoil is greater than the one underneath the airfoil. This results in a difference in pressure between the upper and lower part of the wing, therefore generating an upwards force called the *lift* (see figure 1.2).

When a wing produces a lifting force, this force generates a reaction force in the opposite direction, applied to the air going past the wing. So, the airflow is accelerated downwards locally along the airfoil. This downwards momentum is called the *downwash*, and has a upwards counterpart called *upwash* situated to the right and the left of the wing (see figure 1.3). A vortex is also appearing around each wingtip that reconnects the upwash and downwash zones.

If an airplane were to fly in the zone of the upwash, he would be able to save energy, because he would be flying in an airflow that already has an upwards movement. To analyze this more precisely, let us refer to figure 1.4 (a). The aerodynamic forces of lift and drag are illustrated for an airfoil moving in the air at a given angle of attack (α). The angle of attack is formed by the orientation of the airflow (represented by U_∞) with respect to its motion in the air (On this figure, this angle is exaggerated to ease readability). The downwash generated by the airfoil itself is noted w and impacts the airflow in a manner that the effective angle of attack (α_{eff}), which is the one used to compute the forces on the wing, is smaller than α. The lifting force generated by the moving airfoil is therefore perpendicular to the effective airflow (represented by U_{eff}). As it is angled backwards, this lifting force then decomposes in a vertical force, that will counterbalance gravity, and a backwards force called the *induced drag*. Now let's look at figure 1.4 (b). When the airfoil flies in the same configuration but in an airflow presenting an upwash w_i, it will counterbalance the effect of the downwash generated by the wing. Therefore, the effective angle of attack is bigger, and, as illustrated, the drag generated is lower. Since drag, in cruise flight, is the only force the thrust of the engine must cancel, the airplane flying in an airflow with an upwash can save energy.

1.2 Wake sensing

In the rest of this document, when talking about formation flight, the airplane or drone in front that generates the wake we are interested in will be called the **leader**, while the drone or airplane flying the wake of the leader will be designated by the term **follower**. While downstream of the

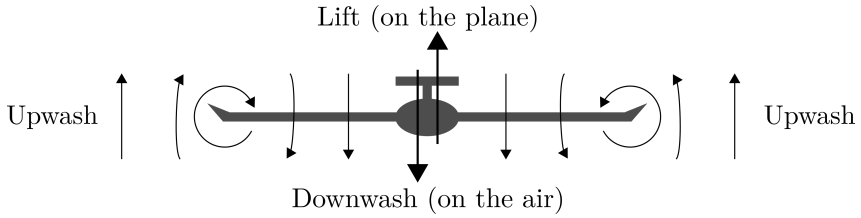


Figure 1.3: Vortex, downwash and upwash behind the aircraft

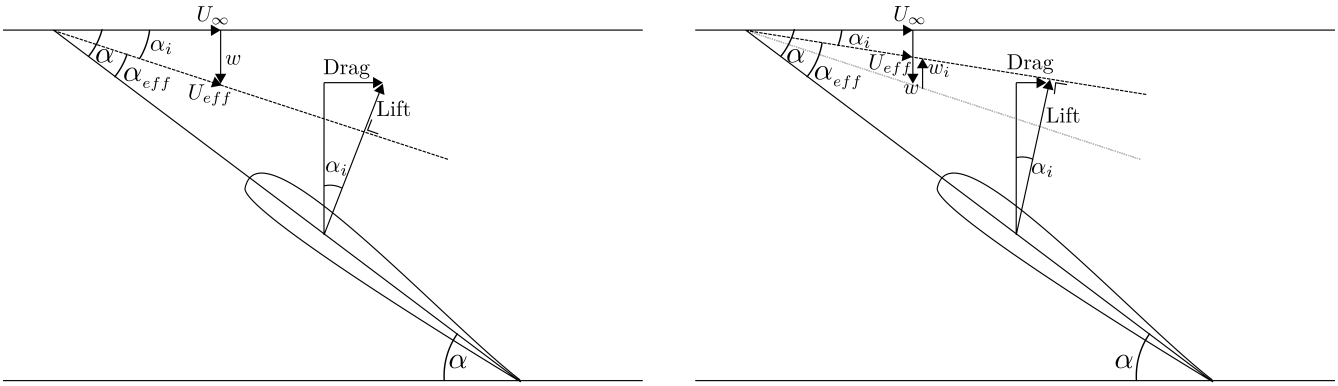


Figure 1.4: Aerodynamic forces on an airfoil without (a) and with (b) upwash from a wake

aircraft, the wake can extend for miles, in the plane formed by the axis along the airfoil and the vertical axis of the aircraft, the positive effect of the upwash on the drag of the follower is unfortunately limited to a zone close to the tip of the wings of the leader. This fact is illustrated in figure 1.5, which is a plot of the upwash speed along the axis of the airfoil of a drone. The characteristics of the drone used for the plot are that of the drone used later in this work.

Since the zones of strong positive upwash are relatively narrow, it is very important that the follower is able to position itself in the wake of the leader as precisely as possible. Doing this requires multiple abilities for the follower:

- Being able to maintain the drone stable
- Knowing where it is situated in the wake
- Knowing where is the ideal position in the wake (to save as much energy as possible)
- Being able to control the drone to move from the current to the ideal position.

The first and last requirements are already met today by many auto-pilots, some of which are open source, such as PX4 or ArduPilot. Finding the ideal position of the follower drone is also already feasible: it corresponds to the position of minimal drag. Today, we have physical models (see section 2.1) of the wake generated by aircrafts that are accurate, and allow us to compute the ideal position in the wake for the drone to fly in. However, finding out where the drone is flying in the wake currently, is not as straightforward as one might think. In perfectly calm air conditions, the wake will be as the models describe, and positioning the follower with respect to the leader is enough to know its position in the wake. In real conditions, however, the wake will be deformed due to multiple effects, such as turbulence, wind, or wake decay [2]. A beautiful example of such a deformation is Crow's instability (see figure 1.6).

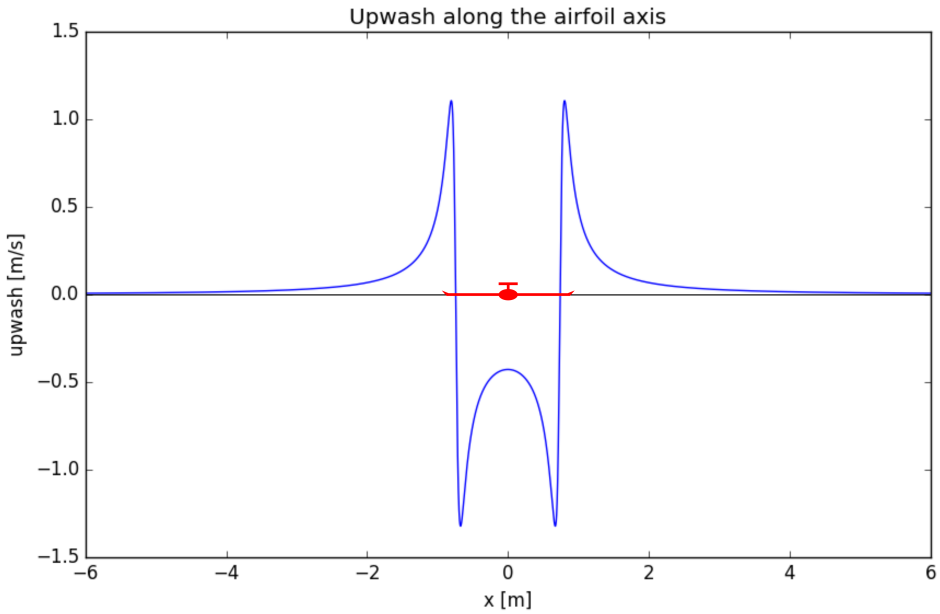


Figure 1.5: Upwash speed along the airfoil of the leader drone (in the center, in red)

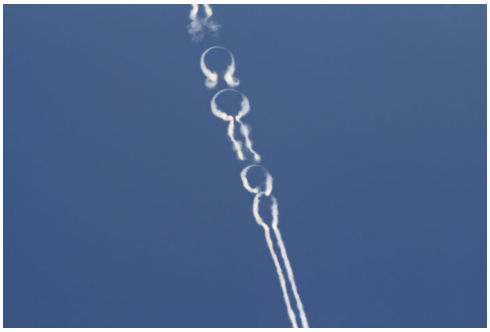


Figure 1.6: Illustration of Crow instability (source: cloud appreciation society)

With these deformations, it becomes virtually impossible to use physical models to describe the wake. First of all, the effects that deform the wake should be known, requiring an enormous amount of sensors to study the atmospheric conditions in which the aircraft flies. Then, the model should be computed in real time, requiring huge amounts of processing power. It is therefore unrealistic to rely on a physics model and the position of the follower with respect to the leader to compute the position of the follower in the wake. The solution is therefore **wake sensing**: the principle is to analyze the current effect of the wake on the follower to deduce its current position in the wake.

1.2.1 State of the art in wake sensing

Today, only a few approaches have been studied to control aircraft positions in formation flight in order to save energy. Most of them are inherently not robust as they don't cover the matter of wake sensing. A first method relies purely on the position of the follower with respect to the leader, therefore assuming perfect atmospheric conditions. These methods have sometimes proven effective in real-life tests. For example, NASA successfully relied on positioning the follower with respect to the leader to test autonomous formation flight using two F/A-18 airplanes and obtained peak drag reductions of 50% [26]. The obtained success was namely due to the very close distance (down to 2 wingspans) between the leader and follower, therefore not allowing the wake to deform significantly with the atmospheric conditions. These energy savings are thus done at the price of seriously increased safety concerns.

A second method of energy saving in formation flight without wake sensing used minimum-finding algorithms for the follower to explore the wake and find the minimal drag position. In [4], an aircraft flight control system for formation flight with F-16 aircrafts is proposed. This control system is separated in two parts: an attitude controller, and an extremum-seeking control scheme. The latter relies on using a model-based measure of the upwash at the center of gravity of the aircraft as input of an extended Kalman filter. This filter is then used to compute the necessary change of position of the aircraft to come closer to a minimum drag point.

In [20], a neural network method was designed to learn in-flight to compute an estimate of the distance between the centers of two UAVs that are flying in formation. The designed method uses the difference in GPS measurements between the two UAVs to train the neural network in flight. The neural network uses states of the aircraft controller and the deviation between the follower's position and the desired formation flight trajectory as inputs, and outputs the said distance between the two aircrafts.

To the author's knowledge, only one method involving wake sensing has been developed today. It arose from the observation that flying in the wake of an aircraft will change the pressure distribution around the airfoil of the follower. Since the physics of wakes is well known, it should therefore be feasible to mathematically deduce the position of the follower in the wake from the observed pressure distribution. This naturally implies that there needs to be a measure of that pressure distribution along the airfoil. Pr. Jeff D. Eldredge and his laboratory at UCLA, for example, work on using wing-distributed pressure sensors to extract the leader's wake parameters [10]. From a physical model of the wake (horseshoe vortex model), they have developed mathematical estimators based on Kalman-type filtering methods and particle filters. First, the authors have deduced, from the horseshoe vortex generated by the leader and a Prandtl lifting-line analysis on the follower, the form of the distribution of differential pressure coefficients. With this, they decided on a state representation and used it to develop both a Kalman filter method and a particle filter method that outputs the parameters of the wake based on the differential pressure coefficient's distribution. The results obtained with this method

show positive prospect in static configurations, when neglecting the aircraft's dynamics. Even though the developed estimators are not very able to estimate the strength of the wake, they are accurate to estimate the relative wake location.

The major drawback of the distributed sensor method lies in the sensors themselves. Not only would it make drone construction more complex by adding the sensors, it would also be more expensive, and could generate perturbations of the aerodynamic properties of the aircraft. Moreover, it excludes all already existing drones of being able to perform effective formation flight. Finally, Kalman filters have a time complexity of at least $\mathcal{O}(n^{2,376})$ [17], meaning that if many iterations are needed to obtain a good estimate (which will probably be the case when taking dynamics into account), the processing power and time required to perform the Kalman filtering will be substantial.

1.2.2 Interest of machine learning

The ideal solution for wake sensing should then require no extra sensors on board of the drone, and be of reasonable computational complexity. All this while performing a mapping from the observable states of the drone (accelerometer, GPS, gyroscopes, commands of control surfaces, ...) to a position in the wake of the leader. Today, there is a power full tool whose main advantage is to be able to perform mapping from one state-space to another when doing it mathematically is unfeasible: artificial intelligence, and more particularly **Machine Learning**. If training a machine learning algorithm requires serious computational power and a lot of time, once it is trained, it requires very little computations to generate an output from a given input. Therefore, if one were able to perform the training in a simulation environment and then transposed the trained machine learning algorithm in a drone, it would not require any particularly powerful hardware.

Performing wake sensing using a machine learning approach without additional sensors is the approached proposed for study in this work.

1.3 Objectives

This work is organized in two main goals. First, an open-source drone simulator will be adapted to enhance the physics of the simulator with the physics of flying in a wake. This has a double objective in mind: first, the study of the behaviour of the wake behind the leader drone, and the effect of that wake on the follower drone. Although these effects are well-documented, we will see how to, starting from an already existing code, modify this code in the most effective possible way to add the wake physics. Secondly, it prepares a more realistic simulation tool for future works at the UCLouvain relating to formation flight of drones.

The second goal is to analyze the feasibility of a wake sensing solution using an artificial intelligence algorithm. This work will be separate from the modification of the simulator, and will be done in a simplified environment. Nevertheless, it will be prepared in Python so that it can eventually be transposed to a direct usage in the complete simulation environment. For this second part, some machine learning algorithms will be presented in order to select the most appropriate for the wake sensing problem. Then, more details will be given on the chosen algorithm and its parameters. Finally, the algorithm will be tested on the simplified wake sensing problem.

1.4 Tools

The first objective of this work being to modify an open-source simulator, it is necessary to present the said simulator. There are three separate software used in this work: ArduPilot and SITL (both part of the ArduPilot environment) make up for the simulation itself. The third software is Dronekit, an independent Python library that runs scripts and communicates with the ArduPilot environment. All of these are open-source.

1.4.1 SITL

SITL (Software in the Loop), is a flight simulator, part of the ArduPilot ecosystem. Written in C++, this is where the physics of the flying drone is implemented, as well as the outputs of simulated sensors. Therefore, the first part of this work is done mainly with SITL, for which the source code must first be understood, and then enhanced with the wake physics.

1.4.2 ArduPilot

ArduPilot itself is an open-source auto-pilot, installable on different types of hardware, but initially developed for Arduino (thus, the name). It can control aerial, ground, aquatic and sub-aquatic vehicles. In the aerial vehicles section can be found multi-rotor drones as well as fixed-wing drones (which will be used in the context of this work). In this work, no modifications will be made on ArduPilot, but it will be used to control the drone simulated in SITL. ArduPilot can be seen as a low-level controller: It controls the attitude of the drone to follow the mission that has been planned. These missions usually take the form of rally points that the drone must fly to at a given speed.

1.4.3 Dronekit

Dronekit is a Python suite that interacts with ArduPilot. It allows the user to run Python scripts that can send high-level mission commands to ArduPilot, such as a new point it must go to, or a speed it must fly at. It usually runs on an independent bit of hardware (for example, a Raspberry Pi) that communicates with the hardware running ArduPilot via a wired or wireless connection. This means that if Dronekit were to crash, the auto-pilot would still run. And since ArduPilot is equipped with a low-battery emergency landing feature, the drone would always be safe.

Having an independent Python script means that one could program a machine learning algorithm with it, to perform wake sensing without harming the auto-pilot's need for a fast response to an attitude change. This is why Dronekit is of interest in the context of this work.

Chapter 2

Physics of Formation Flight: Modifying SITL

The first objective of this work is to enhance the SITL simulator by adding the effects of the wake of the leader drone on the follower. In this chapter, the thinking behind the modifications done in the simulator is explained. First, the model of the wake generated by the leader and the model of the aerodynamics influence of the wake on the follower will be presented, along with the hypotheses made on the wake. Next, for every aspect of the influence of the wake on an aircraft (lift, drag, sideslip and rolling moment), a point will be made about how the wake-less physics are computed in the simulator. Then, for each one, the theory of the physical influence of the wake on these effect will be explained, and an explanation of how these effects have been added in the simulator will be given. The contents of this section are inspired by [5], [6], [7] and [10].

2.1 Wake description

This first section is deditaced to the modelization of the wake and its influence, as well as the simplification hypotheses that were made. From hereon, it is important to define the axes of reference of the aircraft's attached frame, since they will be used in the physics formulas later. For an illustration of the orientation of these axes, see figure 2.1. This figure also shows the definitions of *roll*, *pitch* and *yaw*. The hypothesis made for the added wake physics in the simulator are :

1. Perfect aerodynamic conditions: no wind or other turbulence are taken into account. Therefore, no deformations of the wake are simulated.
2. Neglect dissipation of the wake behind the leader: we consider that once the wake is shed, it stays the same at any distance behind the leader.
3. Consider that there's enough distance between the two drones for the wake of the leader to roll-up completely (see section 2.1.1).
4. Horseshoe vortex for the leader's wake description (see section 2.1.1).
5. Prandtl lifting line for wake effect modelization on the follower (see section 2.1.2).
6. The drones have elliptically shaped wings (The wing's chord has the form: $c(y) = C_c\sqrt{1-(2y/b)^2}$).
7. Neglect leader's dynamics: in the simulator, the position of the leader is represented as a drone with known and perfectly stable altitude, heading and position.

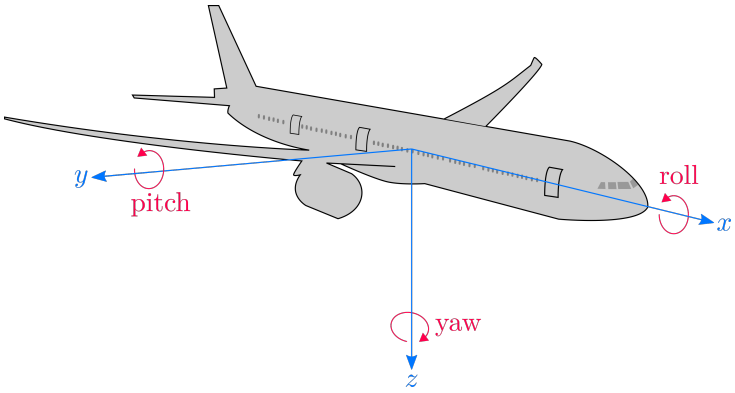


Figure 2.1: Definition of axes of the aircraft's frame

Parameter	Symbol	Value
Wing surface	s	0.45 $[m^2]$
Wingspan	b	1.88 $[m]$
Oswald's efficiency	e_0	0.9
Mass	m	2 $[kg]$
Zero-lift drag coefficient	C_{D_0}	0.56

Table 2.1: Drone parameters for simulations

The characteristics of the drones used later in all simulations (both leader and follower) are found in table 2.1. These values come from the SITL simulator's standard *ArduPlane* vehicle parameters.

2.1.1 Leader model (horseshoe vortex)

If, as stated in hypothesis 3, we consider that enough distance is maintained between the two aircrafts flying in formation, when the wake of the leader reaches the follower, it has had time to roll-up. This phenomenon of rolling-up is illustrated in figure 2.2. In that case, and in perfect atmospheric conditions (hypothesis 1) a horseshoe model is a viable model to represent the wake of the leader [10]. In truth, we consider here a semi-infinite horseshoe vortex because we also neglect the reconnexion of the horseshoe legs far behind the leader. A horseshoe vortex model consists of two vortices centered on the y axis, at a distance of $b_0/2$ and $-b_0/2$ of the center of the airfoil. For elliptically loaded wings (hypothesis 6), the distance b_0 is of $\frac{\pi}{4}b$. Each of these vortices have the same magnitude of circulation Γ_0 but are of opposite rotational direction. An illustration of the horseshoe vortex model is given in figure 2.3.

To describe the upwash that is generated by these two vortices, we will consider the low-order algebraic model [5] given by :

$$u(r) = \frac{\Gamma_0}{2\pi} \frac{r}{r^2 + \sigma^2}$$

With u being the tangential velocity induced by one of the vortices at a distance r of the center of the vortex. Γ_0 is the circulation of the vortex, corresponding to a closed integral of the velocity around the center of the vortex. σ is a parameter tuned to connect more or less smoothly the region in the immediate proximity of the vortex's center. Indeed, if σ is set to 0, since the speed is tangential, the center of the vortex will correspond to a mathematical discontinuity. Usually, σ is set somewhere between 2% and 5% of the wingspan. In this work, it was set arbitrarily at 3.5%. Taking into account the two vortices together, one simply needs to add the u from both vortices together. Of course, since the vortex have opposing direction circulations, their

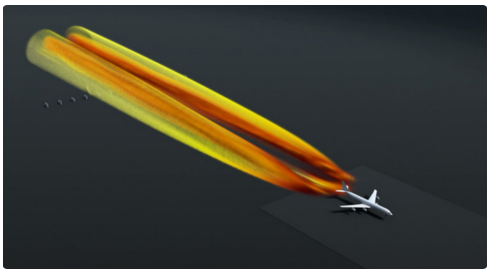


Figure 2.2: Wake roll-up illustration [11]

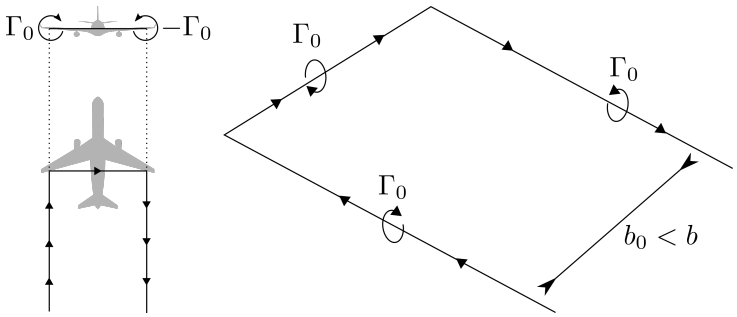


Figure 2.3: Horseshoe vortex representation

signs are opposite. The upwash generated by the two vortices was previously illustrated in figure 1.5. Here, we can see the effect of σ, without which the centers of the two vortices (around the wingtips) would simply be vertical asymptotes.

2.1.2 Follower model (Prandtl lifting line)

By flying in the wake, the effects of the upwash will change the pressure distribution along the wings of the follower. Therefore, it needs a more precise model than the horseshoe model. Here, we use Prandtl's lifting-line theory. This theory is known to provide good results when the analysis is done in quasi-steady conditions [10] and when the aspect ratio of the wing is moderate or large (not smaller than 4) [6]. In the case of the analyzed drone, the aspect ratio is of $AR = \frac{b^2}{S} = 7,85$.

The lifting-line theory can be seen as a more accurate version of the horseshoe model. The principle is roughly the same: assume that the bound vorticity lies on a straight line joining the wingtips. But instead of the horseshoe vortex where the vorticity is considered constant along the line, consider here a varying strength, as illustrated in figure 2.4. At the boundaries of the wings ($y = \pm\frac{b}{2}$), the circulation must be zero $\Gamma\left(\pm\frac{b}{2}\right) = 0$, and by continuity of the vortex tubes, a vortex sheet is shed, with the circulation per unit length of the shed vortex given by $\gamma(y) = \frac{d\Gamma(y)}{dy}$. The downwash velocity (recall figure 1.4) induced by the vortex shed at a point y can be computed by Biot-Savart's law [6] [5]:

$$\delta w_y = \frac{1}{2}\frac{1}{2\pi}\frac{\frac{d\Gamma(y)}{dy'}\delta y'}{y - y'}$$

Where the $\frac{1}{2}$ term issues from the vortex line being semi-infinite. The total downwash at y is then:

$$w(y) = \frac{1}{4\pi}\int_{-\frac{b}{2}}^{\frac{b}{2}} \frac{\frac{d\Gamma(y)}{dy'}}{y - y'} dy' \tag{2.1}$$

This downwash then induces a local change of angle of attack corresponding to:

$$\alpha_e(y) = \alpha(y) - \epsilon(y) = \alpha(y) - \left(\frac{-w(y)}{U_\infty}\right)$$

From the expression of ϵ, it is now possible to find an expression of the circulation $\Gamma(y)$ along the lifting line. To do this, let's consider a_0 as the slope of the lift curve of the wing ($a_0 = 2\pi$ for an elliptical wing). The local lift coefficient is given by $C_L(y) = a_0\alpha_e(y)$, and the lift per unit span can be defined as:

$$l(y) = \rho U_\infty \Gamma(y) = \frac{1}{2} C_L(y) \rho U_\infty^2 c(y)$$

With $c(y)$ the chord of the wing. Therefore, eq 2.1 can be re-written as:

$$w(y) = \frac{1}{4\pi} \int_{\frac{-b}{2}}^{\frac{b}{2}} \frac{\frac{d\Gamma(y)}{dy}}{y - y'} dy = \left[U_\infty \alpha(y) - \frac{2\Gamma(y)}{a_0 c(y)}\right]$$

And by isolating the circulation, one obtains the following *compatibility equation*:

$$\Gamma(y) = \frac{1}{2} a_0 c(y) U_\infty \left[\alpha(y) - \frac{1}{U_\infty} \frac{1}{4\pi} \int_{\frac{-b}{2}}^{\frac{b}{2}} \frac{1}{y - y'} \frac{d\Gamma(y')}{dy'} dy'\right] \tag{2.2}$$

To solve this equation, Prandtl introduced a change of variables: $y(\theta) = -\frac{b}{2}\cos(\theta)$ with $0 < \theta < \pi$. Then, keeping in mind that $\Gamma(\theta)$ has then the border values $\Gamma(0) = \Gamma(\pi) = 0$, one can write the circulation as a Fourier series with sine terms only:

$$\Gamma(\theta) = U_\infty b \sum B_n \sin(n\theta) \implies \frac{d\Gamma(\theta)}{d\theta} = U_\infty b \sum B_n n \cos(n\theta) \tag{2.3}$$

By introducing this expression in the compatibility equation and using Glauert's integral formula, equation 2.2 becomes:

$$\sum B_n \sin(n\theta) = \frac{1}{2} C_{l,\alpha}(\theta) \frac{c(\theta)}{b} \left[\alpha(\theta) - \frac{1}{2} \sum n B_n \frac{\sin(n\theta)}{\sin(\theta)}\right] \tag{2.4}$$

This equation can then finally be solved numerically in order to find the B_n coefficients, and therefore also the circulation. More on this numerical solution in section 2.4

2.2 Lift and Drag

As explained before in section 1.1, flying in a wake modifies the effective angle of attack of an airfoil. It therefore influences the lift and drag forces on the airplane. So, it is necessary to adapt the wake-less simulator by adding this influence.

2.2.1 Wake-less simulator physics

The lift and drag forces on the fixed-wing drone in the simulator are calculated based on the theory described in [1]. The principle used is to compute these forces by the following common formulas, obtained by dimensional analysis from the pressure distribution generated on an airfoil of surface S passing through air with density ρ at a speed U_∞ [6]:

$$F_{lift} = \frac{1}{2} \rho U_\infty^2 S C_L(\alpha) \tag{2.5}$$

$$F_{drag} = \frac{1}{2} \rho U_\infty^2 S C_D(\alpha) \tag{2.6}$$

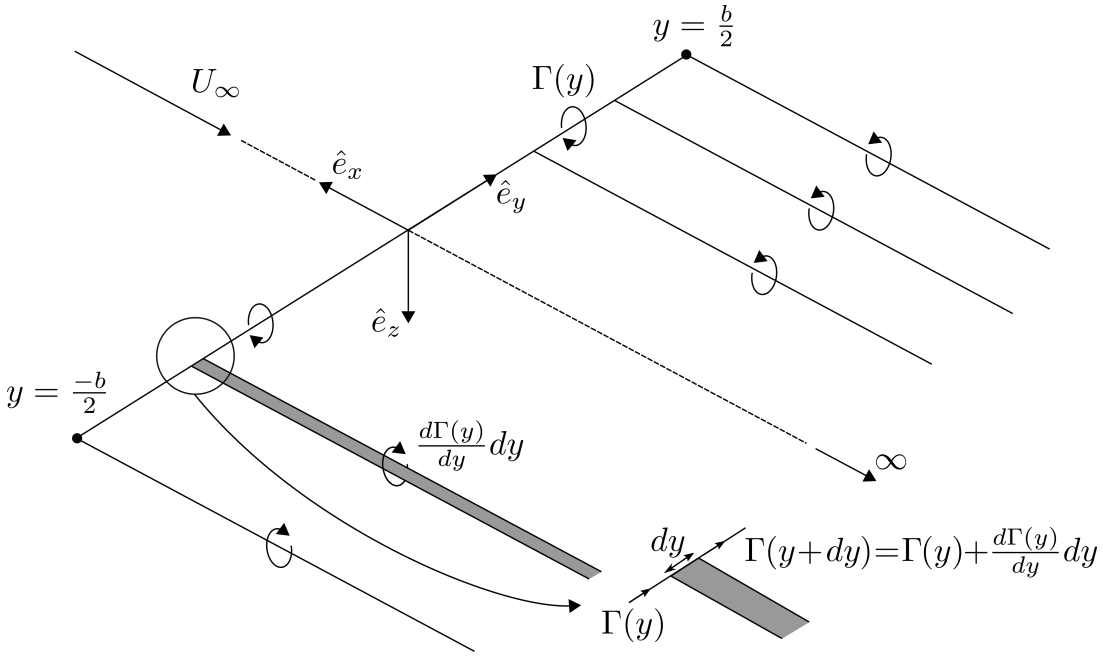


Figure 2.4: Illustrations of Prandtl's lifting-line theory

Where C_L and C_D are aerodynamic coefficients called *lift coefficient* and *drag coefficient*. They depend of the angle of attack (α) and the shape of the airfoil as well as of the Mach and Reynolds numbers. In SITL, the lift coefficient is computed as explained in section 4.2.2 of [1]. The influences of the Mach and Reynolds numbers are neglected. To take into account the effect of the stall, a linear function $C_L = C_{L0} + C_{L\alpha}\alpha$, which is valid for small angles of attack (far away from the stalling points of the aircraft), is blended with the expression of the lift coefficient of a flat plate, given by: $C_{L,flat\ plate} = 2\ sign(\alpha)\sin^2(\alpha)\cos(\alpha)$. This second expression is valid far beyond stalling points. In between, a sigmoid function with cutoff at $\pm\alpha_0$ and transition rate M is used to blend the two functions together. The expression of the lift coefficient becomes then:

$$C_L(\alpha) = (1 - \sigma(\alpha))[C_{L0} + C_{L\alpha}\alpha] + \sigma(\alpha)\left[2\,sign(\alpha)\sin^2(\alpha)\cos(\alpha)\right]$$

With $\sigma(\alpha)$ being the sigmoid function defined by:

$$\sigma(\alpha) = \frac{1 + e^{-M(\alpha-\alpha_0)} + e^{M(\alpha+\alpha_0)}}{\left(1 + e^{-M(\alpha-\alpha_0)}\right)\left(1 + e^{M(\alpha+\alpha_0)}\right)}$$

This blended function is illustrated in figure 2.5.

The drag coefficient is computed as a sum of two terms [6]: the induced drag and the parasitic drag. The induced drag is due to all surfaces that generate lift (illustrated in figure 2.6) and can be expressed as $C_{D_i} = \frac{C_L^2}{\pi A}$. The parasitic drag is generated by the other components of the drone (fuselage, tail, ...) who don't participate to the lift but create drag nonetheless. This drag can be treated as a constant, that we will name C_{D_0}. Only taking into account the linear part of the lift coefficient into account for the expression of the induced drag, the full expression of the drag coefficient becomes:

$$C_D(\alpha) = C_{D_0} + \frac{(C_{L0} + C_{L\alpha})^2}{\pi\, e\, AR}$$

Where e is Oswald's efficiency and AR the aspect ratio of the wing.

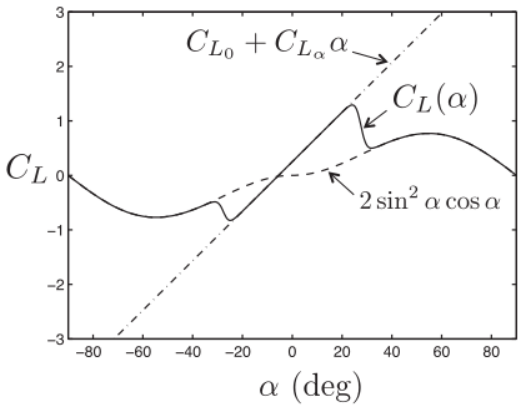


Figure 2.5: Blended lift coefficient function (source [1])

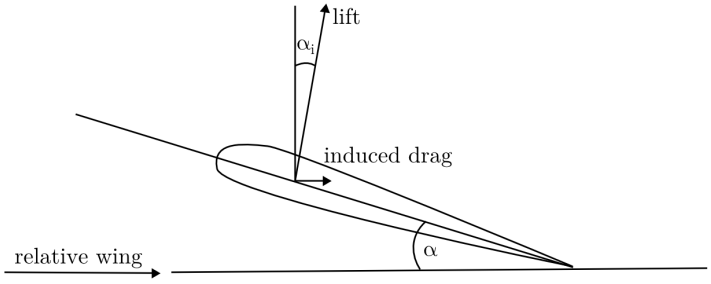


Figure 2.6: Lift-induced drag

Once these coefficients are defined, the only missing link is to compute the angle of attack α. In the code, the algorithm start by computing the angle of attack based on the airspeed of the drone via $\alpha = \arctan\left(\frac{U_{\infty_z}}{U_{\infty_x}}\right)$. Then, the coefficients of drag and lift are computed as explained here above and finally the lift and drag forces are computed.

2.2.2 Wake physics added in the simulator

To prevent any slow-down in the simulator, it is important to add the wake physics all the while maintaining the code as intact as possible. Keeping this in mind and recalling that the influence of the wake generates a local change of the angle of attack, the natural conclusion is that we can compute the additional term leading to the effective angle of attack $\alpha_{eff} = \alpha - \left(\frac{-\overline{w_u}}{U_\infty}\right)$ and then replace the angle of attack in the code of the simulator by this effective angle of attack, therefore not interfering with the computation of aerodynamic coefficients and forces already in place. The important point is then to compute the term $\overline{w_u}$, the average upwash. It is obtained by averaging the perpendicular to the wing component of the speed induced on the wing by the wake:

$$\overline{w_u} = \frac{1}{b}\int_{-b/2}^{b/2} u_\theta(y)\cos(\gamma(y))dy$$

With γ being the angle between the velocity of the vortex (u_θ) and the perpendicular to the follower's wing (see figure 2.7). Because there are two vortices (one for each wingtip of the leader), the previous expression needs to be duplicated and re-written as:

$$\overline{w_u} = \frac{1}{b}\int_{-b/2}^{b/2} u_{\theta 1}(y)\cos(\gamma_1(y)) + u_{\theta 2}(y)\cos(\gamma_2(y))\ dy$$

Where the indices 1 and 2 correspond to each of the two vortices.

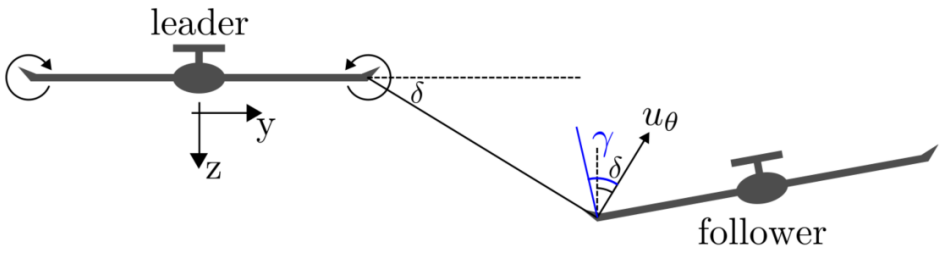


Figure 2.7: Orientation of u_θ and definition of γ

Concretely, in the code, at each time step of the simulator, the angle of attack will now be computed as $\alpha = \tan^{-1}\left(\frac{U_{\infty z}}{U_{\infty x}}\right) - \frac{\overline{w_u}}{U_\infty}$. This obviously means that the integral leading to $\overline{w_u}$ must be computed at each time step. To do this, the method of trapezoidal numerical integration is used. For each dy of the integration along the airfoil of the follower, multiple steps have to be fulfilled in order to compute $u(\theta)$ and $\cos(\gamma)$:

1. Using a transformation matrix representing the roll, pitch and yaw rotations as well as the translation from the follower frame to the leader frame, the position of the current integration point is transformed from the follower frame to the leader frame. The details of the calculation of the matrix and its usage is given in Annex A.

2. The distance between the integration point and the two vortices generated by the leader can then be computed ($r_1(y)$ and $r_2(y)$).

3. With these distances, one can the compute the norm of the two speed vectors experienced at the integration point. This is done by replacing the distances in the expression of the vortex (see section 2.1). This therefore gives the $u_{\theta 1}(y)$ $u_{\theta 2}(y)$ needed in the integral.

4. Next, the angles δ_1 and δ_2 (see figure 2.7) between the two speed vectors and the z vector of the leader's frame are computed via the geometrical identity $\delta = \arctan(\frac{\Delta z}{\Delta y})$ with Δz and Δy being the distances along axes z and y of the leader frame between the integration points and the center of the vortices.

5. The previous point allow the computation of normalized speed vectors in the frame of the leader: $\hat{u}(\theta) = (0, \sin\delta, \cos\delta)^T$.

6. These vectors can then be used to compute their scalar product with the normalized vertical vector of the follower's frame, which is computed by using the transformation matrix on the vector $(0, 0, -1)^T$. The scalar products (also called "dot product") that are computed are the equivalent of $\cos(\gamma)$ and are noted dot_1 and dot_2.

7. Finally, the local upwash at the current integration point is computed as $u = u_{\theta 1}\, dot_1 + u_{\theta 2}\, dot_2$

The results of this algorithm are illustrated in figure 2.8 with a plot of the induced change in angle of attack in function of the position in the wake. The position of the leader drone is showed in the center of the figure in black.

It is interesting to note that since computing γ is not as straightforward as one might think, but computing δ (the angle between the speed vector and the vertical axis of the leader) is easy, a little change was done between the theory and the numerical computation: With the angle δ computed, unitary vectors aligned with the speed vectors generated by each vortex at each

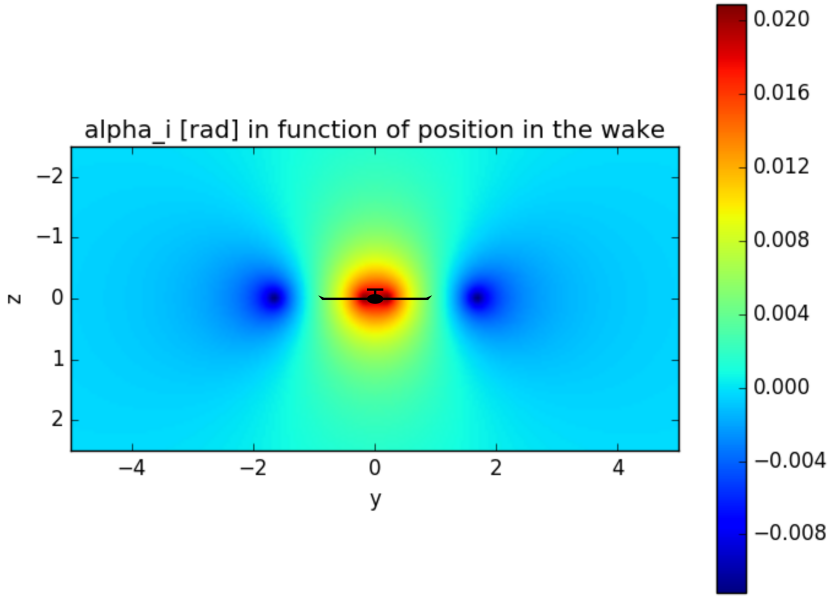


Figure 2.8: Change of angle of attack ($\frac{\overline{w_u}}{U_\infty}$) induced in function of position on the wake

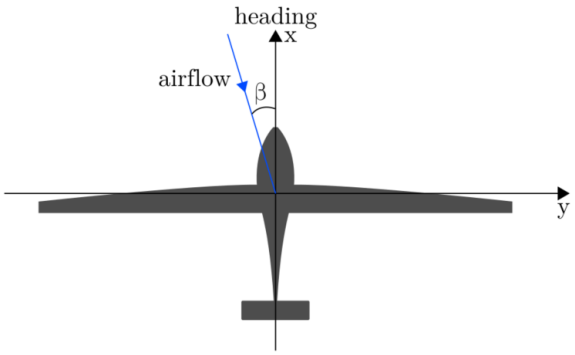


Figure 2.9: Sideslip angle β

integration point are easily computed, and can then be used to compute the scalar product of these vectors with a unitary vertical axis vector of the follower. This gives of course the same value as $\cos(\gamma)$, but with simpler numerical computations.

2.3 Slip

2.3.1 Wake-less physics

When flying in an airflow, it happens that the heading of the drone is not perfectly aligned with the airflow. When this happens, we say that the aircraft *slips*. Mathematically, the slip is quantified via the *sideslip angle*, noted β (see figure 2.9). In the code of the simulator, the definition of this angle is similar to the definition to the angle of attack [1]:

$$\beta = \arctan \frac{U_{\infty_y}}{U_{\infty_x}} \tag{2.7}$$

2.3.2 Wake physics added in the simulator

Again, to prevent any slow-down of the simulator by adding the wake physics, it is fundamental to add as little code as possible to implement the effect of the wake on the sideslip angle. In the code, the sideslip angle is directly computed from the x and y components of the airflow, using eq 2.7. Therefore, it is straightforward to see that the effect of the wake can easily be added by computing the x and y components of the radial speed u_θ and adding them to U_{∞_x} and U_{∞_x}. Eq 2.7 becomes then:

$$\beta = \arctan\left(\frac{U_{\infty_y} + u_{\theta_y}}{U_{\infty_x} + u_{\theta_x}}\right)$$

To compute the extra terms that are added in the expression of β already in the code, a similar method as for the angle of attack is used. The only difference is that there will be no integration as we consider that the sideslip will be mainly generated by the rudder and the fuselage, therefore limiting the influence to only the center of gravity of the drone:

1. Using the same transformation matrix as in section 2.2, compute the position of the center of the follower drone expressed in leader frame.
2. The distance between the center point and the two vortices generated by the leader can then be computed ($r_1(y)$ and $r_2(y)$).
3. With these distances, compute the norm of the two vortex speed vectors by replacing the distances in the expression of the vortex (see section 2.1). This gives $u_{\theta 1}$ $u_{\theta 2}$.
4. Next, the angles δ_1 and δ_2 (see figure 2.7) between the speed vectors and the z vector of the leader's frame are computed (δ).
5. The previous point allows for the computation of normalized speed vectors in the frame of the leader: $\hat{u}(\theta) = (0, \sin\delta, \cos\delta)^T$.
6. These vectors can then be used to compute the scalar product with a the normalized x and y vectors of the follower's frame (called dot_{1x}, dot_{1y} and dot_{2x}, dot_{2y}).
7. Finally, $U_{\infty_x} = u_{\theta 1}\, dot_{1x} + u_{\theta 2}\, dot_{2x}$ and $U_{\infty_y} = u_{\theta 1}\, dot_{1y} + u_{\theta 2}\, dot_{2y}$ can be computed and added to the calculation of β already in place.

The results obtained with this algorithm are showed in figure 2.11. It shows the value of the sideslip angle induced by the wake in function of the position of the follower in the wake. The leader drone is represented in the center in black.

One might argue (and this is correct) that the influence of an incoming wake on the sideslip angle is a small effect in the globality of the simulator, and is not fundamental to add in order to start exploring the use of AI to detect where the drone is located in the wake of the leader. But despite the very low impact on the physics of the drone, the sideslip angle has one very important purpose: it prevents symmetry in the effects of the wake along the vertical axis. As illustrated in figure 2.10, if two airplanes are flying in the same wake, with their position only changed by symmetry along the y axis of the leader frame, the effects of the wake on the lift and the rolling moment (see section 2.4) are identical given the fact that the projection of the vortex speeds on the vertical axis of the follower frame are what causes these influences of the wake. On the other hand, as illustrated by the blue arrows in figure 2.10, the direction of the slip changes between one position and the other. Therefore, the influence of the slip is fundamental for the AI to be able to distinguish the two otherwise symmetrical positions.

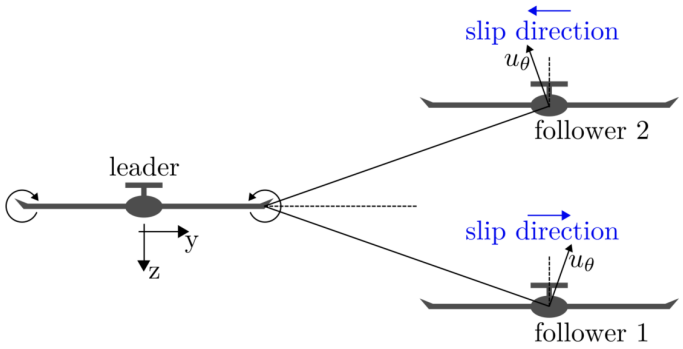


Figure 2.10: Vertical asymmetry of sideslip

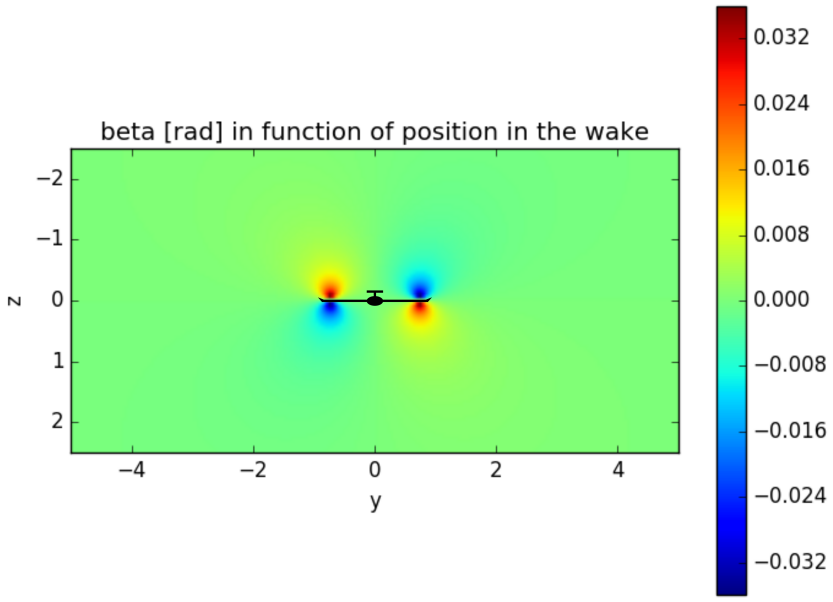


Figure 2.11: Sideslip angle in function of position on the wake

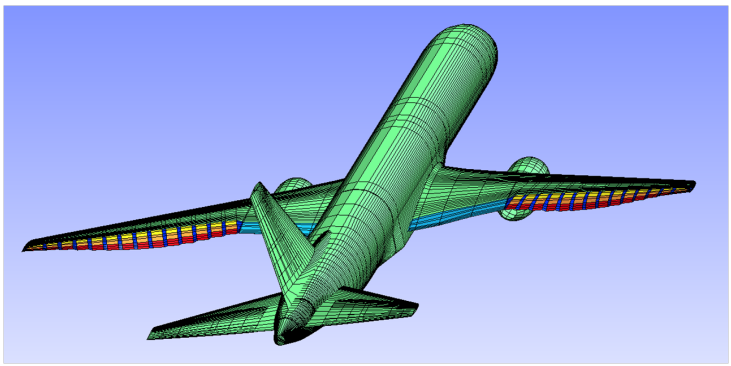


Figure 2.12: Variable Camber Continuous Trailing Edge Flap (VCCTEF) (Source: Boeing, NASA)

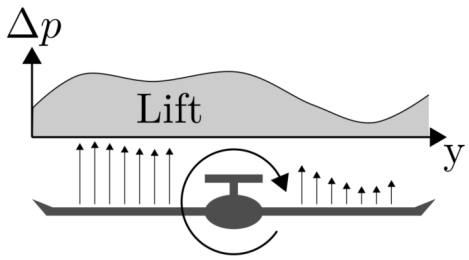


Figure 2.13: Example of rolling moment due to asymmetrical pressure distribution

2.4 Rolling moment

As explained in section 2.1, the influence of the wake on a wing modifies locally its angle of attack. Therefore, one can see a regular wing evolving in a wake as a wing with *twist*. A wing with twist is a wing for which the profile's orientation changes longitudinally, therefore changing the angle of attack locally. For an illustrative example, see figure 2.12: Boeing and NASA are working on a new type of aircraft control based on twistable wings instead of typical ailerons called Variable Camber Continuous Trailing Edge Flap (VCCTEF) [18]. This difference in α along the airfoil changes the pressure distribution, which is not necessarily symmetrical with respect to the fuselage, therefore generating a *rolling moment*. Indeed, the lift force generated by the wing is actually due to this gradient of pressure between the top and bottom of the wing. If, then, the pressure varies along the wing (as illustrated in figure 2.13), the lift force generated will be superior on one side of the airplane, generating a rolling moment.

2.4.1 Wake-less physics

In the simulator's computations, since the pressure distribution along the wing is considered symmetrical, the only rolling moment is generated by the controls of the drone, namely the ailerons and the rudder. The ailerons act by changing the pressure distributions on the wings: On one wing, the aileron will augment the lift, while the aileron on the opposite wing decreases the amount of lift on that wing. This difference in lift from left to right therefore generates a rolling moment. Mathematically, this is modeled in SITL using a linear aerodynamics model from Randal Beard's *Small Unmanned Aircraft: Theory and Practice* [1]. For a standard aircraft configuration, like the *ArduPlane* vehicle, the control surfaces are ailerons, a rudder and an elevator. The amount of deviation between their resting position and their current position is called a deflection, and is respectively noted δ_a, δ_r and δ_e (see figure 2.14). Since the ailerons go by pair, the aileron deflection is actually defined by $\delta_a = \frac{1}{2}(\delta_{a,\,left} - \delta_{a,\,right})$

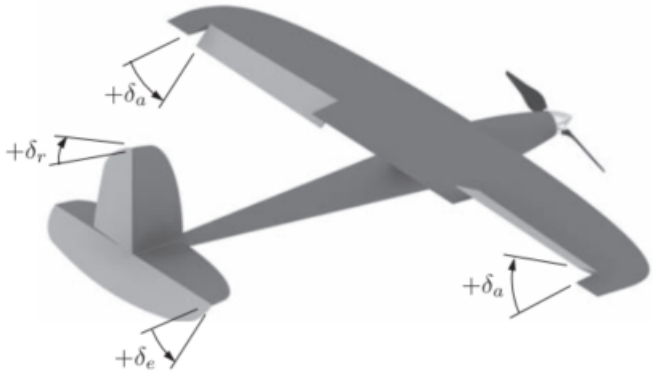


Figure 2.14: Standard aircraft configuration of control surfaces and deflections (source [1])

Just as the lift and drag can be expressed with the equations 2.5 and 2.6, a moment can be expressed as:

$$m = \frac{1}{2}\rho U_\infty^2 S \bar{c} C_m$$

With $\bar{c}$ being the mean chord of the aircraft and C_m an aerodynamic coefficient. According to [1], the rolling and yawing moments are influenced mainly by the sideslip angle β, the roll rate p, the yaw rate r and the deflections from the rudder and the ailerons. The rudder and the aileron deflection both generate moments in yaw and roll. Indeed, as any (even novice) pilot will have experienced, using the rudder alone generates a rolling moment reinforcing the yaw movement. On the other hand, using the ailerons alone generate what is called the *adverse yaw*, a yaw movement which tends to counteract the effect of the roll. With all the previously cited dependencies, the rolling moment can be expressed as:

$$l = \frac{1}{2}\rho U_\infty^2 S b C_l(\beta, p, r, \delta_a, \delta_r)$$

With C_l being a non-dimensional aerodynamic coefficient which is nonlinear with respect to its parameters. This nonlinear relationship being complex to define, we use a linear model, which yields usually acceptable results. By using a first order Taylor series approximation and some nondimensionalization, we obtain the following linear relationship:

$$l = \frac{1}{2}\rho U_\infty^2 S b \left[C_{l_0} + C_{l_\beta}\beta + C_{l_p}\frac{b}{2U_\infty}p + C_{l_r}\frac{b}{2U_\infty}r + C_{l_{\delta_a}}\delta_a + C_{l_{\delta_r}}\delta_r\right]$$

With C_{l_0} being the bias (or parasitic) rolling moment (the rolling moment when all control surfaces deflection are zero), and all other C_{l_i} being the rolling moment coefficients in relation with the parameter replacing i.

2.4.2 Wake physics added in the simulator

Unfortunately, due to the way the rolling moment is computed in the code, there is no direct way to enhance the already in place code with the physics of the wake. Therefore, the only solution is to go back to the roots, and compute the integral of the circulation that corresponds to the rolling moment it generates. Given that the lift force is computed with the integral:

$$L = \rho U_\infty \int_{-\frac{b}{2}}^{\frac{b}{2}} \Gamma(y)\, dy$$

And by recalling that a moment is a force multiplied by the moment arm (here the distance with the center of gravity, considered to be the geometrical center), the moment can be computed via

the integral:

$$M = \rho U_\infty \int_{-\frac{b}{2}}^{\frac{b}{2}} y\,\Gamma(y)\,dy \tag{2.8}$$

It is thus clear that to compute the rolling moment, we must solve numerically the equation 2.4 from section 2.1.2 in order to obtain the B_n terms to compute the circulation Γ needed in the integral. For a wing with twist, which is physically similar to a wing in a wake, we decompose $B_n = b_n\alpha - D_n$. This way, the compatibility equation can be decomposed in a system of two separate equations:

$$\begin{cases} \sum_n b_n q_n(\theta) = r(\theta) \\ \sum_n D_n q_n(\theta) = r(\theta)\alpha_v(\theta) \end{cases}$$

Where the following definitions hold:

$$\begin{aligned} q_n(\theta) &= \sin(n\theta) + \frac{1}{4}a_0\frac{c(\theta)}{b}n\frac{\sin(n\theta)}{\sin(\theta)} \\ r(\theta) &= \frac{1}{2}a_0\frac{c(\theta)}{b} \\ \alpha_v(\theta) &= \frac{u_v(\theta)}{U_\infty} \end{aligned}$$

Therefore, since the first equation of the previous system only depends on variables that are not expected to change during the simulation, it can be solved before the simulation begins to save computational power during the simulation. On the other hand, since the second equation of the system depends on α_v, which in turn depends on the perpendicular to the airfoil upwash speed u_v, it needs to be solved at each time step.

To solve the system of equations, we use the least-squares numerical method [5]. For the first equation of the system, we have:

$$\begin{aligned} \mathcal{H}_1 &= \int_0^\pi \left(\sum_n b_n q_n(\theta) - r(\theta)\right)^2 d\theta \\ \Rightarrow \frac{\partial \mathcal{H}_1}{\partial b_m} &= 2\int_0^\pi \left(\sum_n b_n q_n(\theta) - r(\theta)\right)^2 q_m(\theta)d\theta = 0 \\ \Rightarrow \sum_n \left(\int_0^\pi q_m(\theta)q_n(\theta)\right) b_n &= \int_0^\pi q_m(\theta)r(\theta)d\theta \end{aligned}$$

Which can then in turn be re-written as a linear system with matrix notations:

$$\begin{aligned} \sum_n A_{mn}b_n &= c_m \\ \Rightarrow \mathbf{Ab} &= \mathbf{c} \end{aligned}$$

As said previously, this needs to be solved only once for the whole simulation. For the second equation, the same method yields:

$$\begin{aligned} \mathcal{H}_2 &= \int_0^\pi \left(\sum_n D_n q_n(\theta) - r(\theta)\alpha_v(\theta)\right)^2 d\theta \\ \Rightarrow \frac{\partial \mathcal{H}_2}{\partial D_m} &= 2\int_0^\pi \left(\sum_n D_n q_n(\theta) - r(\theta)\alpha_v(\theta)\right)^2 q_m(\theta)d\theta = 0 \\ \Rightarrow \sum_n \left(\int_0^\pi q_m(\theta)q_n(\theta)\right) D_n &= \int_0^\pi q_m(\theta)r(\theta)\alpha_v(\theta)d\theta \end{aligned}$$

Which can then in turn be re-written as a linear system with matrix notations:

$$\sum_n A_{mn} D_n = c'_m$$
$$\Rightarrow \mathbf{Ad} = \mathbf{c}'$$

One could notice that we find the same matrix $\mathbf{A}$ than for the first system, so this matrix shall be stored in memory after solving the first system to be reused here. At every time step, only the vector $\mathbf{c}'$ must be computed, to then solve the system in order to finally obtain the D_n coefficients.

The algorithm implemented in the code is the following:

- Before the simulation:
  - Generate $\mathbf{A}$ and $\mathbf{c}$.
  - Use the *Eigen* library from C++ to solve the system and find $\mathbf{b}$.
  - Store $\mathbf{A}$ and $\mathbf{b}$.
- At each time step:
  1. Fill the $\mathbf{c}'$ vector. This requires to integrate over (amongst others) $\alpha_v(\theta)$. For every integration point θ, use the same algorithm as in section 2.2.2.
  2. Use the stored $\mathbf{A}$ and the computed $\mathbf{c}'$ to solve the system with *Eigen* and obtain $\mathbf{d}$.
  3. Compute the integral of equation 2.8 to obtain the rolling moment induced by the wake.
  4. Add the computed rolling moment to the one already computed in the wake-less code.

The result of this algorithm as a function of the position in the wake is showed in figure 2.15. Unfortunately, as it becomes clear from the algorithm, the amount of calculation to compute $\alpha_v(\theta)$, at each time step, along the whole airfoil and then to solve the system, is significant. It does slow down the simulator by a factor 3.

2.5 Energy savings

Regarding in-flight energy consumption, we can neglect the energy needed for moving the control surfaces. Then, all the energy used during the flight is due to the engine(s) of the aircraft. As explained previously in section 1.1, when an aircraft is cruising, the only force the engine must work against is drag. Therefore, the location(s) in the wake that induce the greatest reduction in drag are the ones of optimal energy consumption. These locations are directly linked with where the influence of α_i is the biggest. If we go back to figure 2.8, we see that there are two positions of minimal drag: the two dark blue spots to the left and the right of the leader. These points are located in $(y, z) = (\pm 1.69, 0.0) = (\pm 0.9b, 0.0)$, as illustrated in figure 2.16. In ArduPilot, there is a "throttle" parameter that can be read by Dronekit, so one could implement a minimum energy point seeking algorithm. However, ArduPilot is not extremely reactive and allows a small error in the desired flying speed before applying corrections. Therefore, it is still best to perform the wake sensing task, as we would then be able to know how to move the drone to reach the known minimum energy point.

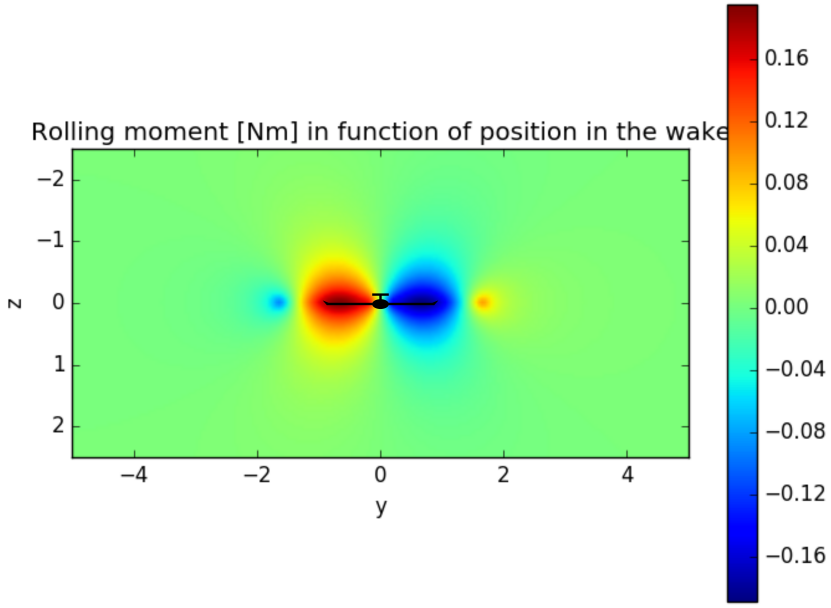


Figure 2.15: Rolling moment induced by the wake in function of position on the wake

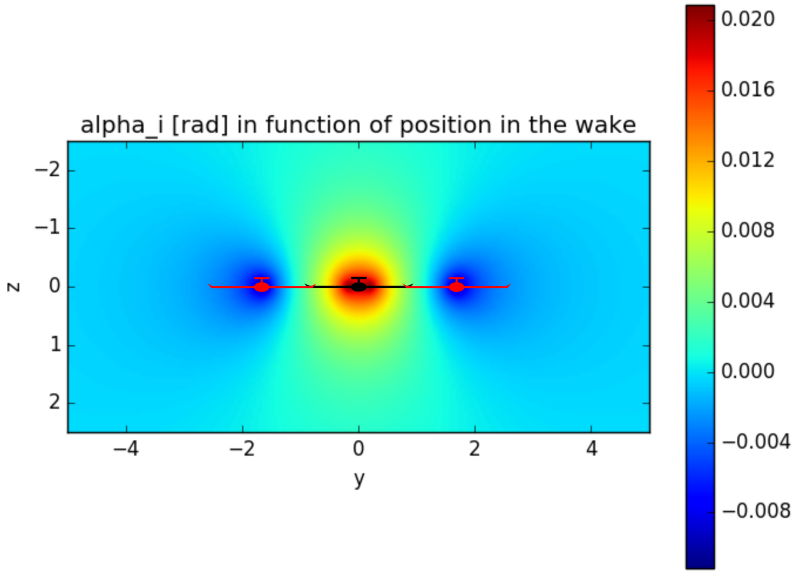


Figure 2.16: Ideal positions of follower drones for minimal energy consumption (illustrated on the plot of figure 2.8, follower in red, leader in black)

2.6 Other modifications to the simulation environment

For more details about the general organization of the code and the modifications that were done to the code of the simulation environment, please refer to annex D. This section also describes other modifications that were made to the simulator for reasons not directly related to wake physics.

Chapter 3

Choose an Artificial Intelligence Algorithm

This chapter's goal will be to choose an artificial intelligence algorithm to perform wake sensing without additional sensors. Therefore, the approach will be to use as inputs the correction commands applied by the autopilot to compensate the effects of the wake. In other words, the only data that will be used as the input of the algorithm will be the deflections of the control surfaces. The output will be the (y, z) coordinates corresponding to the position of the follower in the wake.

Besides the selection of an algorithm, this chapter will also serve as a quick introduction to the vast subject of artificial intelligence. Indeed, today "Artificial intelligence" and "Machine learning" are selling buzzwords, frequently used indiscriminately, without a clear distinction between both terms. Machine Learning is actually but a very small part of what is today called Artificial Intelligence. Before entering the heart of the subject of wake sensing using AI, it seems therefore necessary to recall what artificial intelligence is about. This chapter will start by introducing the general concept and definition of AI. Next, a short overview of popular sub-concepts of AI that are less relevant for the wake sensing problem will be presented. Finally, machine learning will be introduced. In this final part, the most important algorithms of machine learning will be presented and confronted with the wake sensing problem to choose the most adapted. The content of this chapter is mainly inspired by "Artificial Intelligence, a modern approach" by Stuart Russel and Peter Norvig [24], and "Machine Learning" by Tom M. Mitchell [25].

3.1 Definition of AI

It is a rather complex task to give a precise definition of what artificial intelligence is. Doing some research in the scientific literature gives almost as many different definitions to AI as there are authors. Some examples of definitions found in [24] are listed below:

- The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning,... (Bellman, 1978)
- The art of creating machines that perform functions that require intelligence when performed by people (Kurzweil, 1990)
- The study of how to make computers do things at which, at the moment, people are better (Rich and Knight, 1991)
- The study of mental faculties through the use of computational models (Charniak and McDermott, 1985)

- The study of the computations that make it possible to perceive, reason, and act (Winston, 1992)
- AI is concerned with intelligent behaviour in artifacts (Nilsson, 1998)

From these examples, it is clear enough that the opinions of these authors differ on the term "intelligence". This is however not a particularity of the AI researchers, as the definition of the word "intelligence" tends to give headaches to scientist in many different subjects, namely zoology, philosophy, medicine, psychology, ... Concerning AI in particular, there are four main tendencies to be noticed. First, the distinction between *human* behaviour and *rational* behaviour. While the term "human" behaviour is probably easy to understand for the reader, the term "rational" behaviour might require some explanation. Rationality is about behaving in such way that the agent will always make the choice that leads to the best outcome (and if uncertainty is involved, the best expected outcome). The second distinction is made between *thinking* and *acting*. The fields of research that focus on thinking try to mathematically and algorithmically describe the thinking process, while the focus on acting concerns itself not about the process of thinking, but solely on the output. In this work, the focus will be made on the field of *acting rationally*.

3.2 Overview of AI's sub-concepts

Artificial intelligence is a vast subject with multiple sub-concepts aimed at tackling different kinds of problems. Wake sensing requires to learn a function mapping the space of the physical effect of the wake to the (y, z) position of the follower in the wake. This is a particular kind of problem in which Machine Learning will be the most effective approach, since it is aimed at making a computer learn something a human can not describe in a straightforward manner. This section will shortly introduce the other main sub-concepts of AI and the kind of problems they are ideal for.

It is first useful to introduce some vocabulary proper to AI:

- **Representation**: Chosen way of representing the problem. For a chess game, the representation could be a $8 * 8$ matrix of numbers that each represent the current occupation of each tile (0 if empty, 1 if pawn, ...).
- **State**: One particular value of the representation of the problem. The initial state is the one in which the problem starts.
- **Agent**: Evolves in the environment of the problem. It senses the problem through sensors, and can interact with the problem through actions.
- **Action**: What an agent can perform to modify the current state of the problem. For example, in a chess game, an action could be $move\_pawn(x, y)$, which would move the pawn in position (x, y) 1 tile forward.
- **Transition model**: Each action has a transition model, which describes how the state is modified by it. The transition model of action *move_pawn* could for example be to modify the matrix by setting the value of the current tile to 0 and the tile in front of the pawn from 0 to 1 (after checking the value of the tile in the front to check if the action can be performed).
- **Goal**: A goal is a particular state the agent aims to reach. There can be multiple different goals. For example, in chess, there are many different states that correspond to a won game.
- **Path**: List of states from the starting state of the problem to the current state.

3.2.1 Search

The first sub-concept of AI that will be explored here is the *search* algorithm. This algorithm is called search because it searches the space of reachable states to find a path to reach a goal state. The principle of search is rather simple: starting from the current state, the agent first generates all the possible successors states, *i.e.* all the reachable states when applying one of the possible actions on the current state. All these successors can then have successors of themselves, therefore generating a tree. If every generated successor is checked to see if it has appeared in the tree before, and deleted if it did, then the agent generates a graph instead of a tree. The *depth* of the tree corresponds to how many "levels" of successors are generated. The nodes of the tree represent different states of the problem, while the branches represent actions that the agent can perform on the given states. When, while generating the successors, the agent finds a goal state, he travels back up the tree or graph to compute the path he should follow to reach the state. For more details about the search algorithm and details about the different forms of searches, please refer to annex B.1. Search algorithms are used widely in motion planning problems. One of the most used motion planning algorithms in robotics, called A^*, is actually a search algorithm.

3.2.2 Constraint satisfaction

Very different from search methods, the constraint satisfaction problem (CSP) approach does not represent the problem as a number of defined states that are reachable by the agent. Instead of that, it uses a *factored representation*, *i.e.* the states are described by a number of variables. Each of these variables have a defined domain, representing all the values that this variable can take. The problem is solved when each variable satisfies a set of constraints. Of course, the problems that are ideally solved with a CSP-approach are ones which are easily represented with a limited number of variables and constraints. A good example of a problem ideally modeled as a CSP is the well-known Sudoku game. The rules of Sudoku themselves are indeed constraints (for example: not twice the same number on the same line, ...). For an example of a constraint representation of the Sudoku game, see annex B.2.

When all the variable satisfy all the constraints, the algorithm terminates. To solve the problem, CSP algorithms can decide either to search (*i.e.* choose a new assignment for the variables), or can *propagate constraints*. This means that the algorithm will use the constraints to narrow down the domains of the variables. For example, if in the starting grid of the Sudoku, there is a 3 in the $(1, 1)$ cell, constraint propagation can remove the value 3 from the domains of all the cells that are in the first column, the first line, and the first sub-grid. This constraint propagation might delete the penultimate value in one of the domains, therefore fixing the value of that cell, which can in turn be used to propagate constraints, and so on. The advantage is that the constraint propagation algorithms are general and can be used for any set of variables, domains and constraints. Therefore, no adaptation of the algorithm is necessary when switching from one problem to another; one simply needs to describe the new problem as a set of variables, domains and constraints.

3.2.3 Propositional Logic

The main idea behind logical agents is to make agents that have an internal representation of knowledge. In the previous concepts, the agents have no real knowledge about the environment in which they operate. If for example a robot is moving in an environment using informed search, it might compute, through the value of the heuristic function, that by moving to the left it is getting closer to its goal, but it does not have an internal form of knowledge on which he would base this observation. If a human were in the same situation, he would reason in a way comparable to this: "I know that the goal is to my left. I know that the action `move_left`

makes me move to the left. Therefore, I know that executing the action `move_left` brings me closer to the goal". Making an agent that has the same kind of knowledge and reasoning as what is illustrated by this example is exactly what propositional logic (PL) is about. This kind of approach is of course best applied to problems for which human logic would be powerful. Another interesting feature of this method is that, once the algorithm has found a goal state, a human can *understand* the thinking process of the algorithm. Again, for the wake sensing problem, PL is not an interesting method since human knowledge representation will not help in finding a function mapping from one space to another. So, the working process of PL is out of context for this work. Nonetheless, the interested reader can refer to annex B.3 for more details.

3.2.4 Planning

This concept is not fundamentally different to the ones presented before and its description will therefore not be as complete. The main difference with the previous concepts is the introduction of a new syntax particular to planning problems. The goal of this syntax is to take time and resources into account. Each action is therefore described not only as its result, but also as how much times it consumes and how many resources it mobilizes. Planning problems therefore also concern themselves with resource management and the fact that the resources are usually limited. An example of a problem that should be described as a planning problem is a car workshop, with resources being the limited number of workers and machinery while the time constraint is a full day of work during which all intended maintenance and reparations have to be performed.

3.2.5 Probabilistic reasoning

Not fundamentally different to propositional logic, the difference is that instead of considering that some knowledge is true or false, it will be given a certain degree of certainty. So, this is a type of logic that takes probabilities into account, and can be used in situations where the outcome of an action if not explicitly known.

3.3 Machine learning

The previous section gave an introduction about how to write algorithms to make a machine behave intelligently. Machine learning is a very different concept, that means to develop machines that can *learn* how to behave instead of explicitly being told so. The main importance of learning is that for complex problems, it is time-consuming (if it is even possible) for the programmers to trace all the possible situations that could occur during the evolution of an agent in its problem environment. Also, sometimes the programmer himself does not know how to program a certain task. Vision is certainly one of the best examples of these un-programmable situations. While it can be possible to write a program that recognizes a black square on a white paper, how do you write a program that recognizes an animal? Certain very complex algorithms have been written (and work) for vision and object recognition, but it is a very complex task, that machine learning algorithms have also been able to tackle.

There are multiple approaches to machine learning. As this is merely an overview of the subject of AI, only a few of them (chosen arbitrarily but taking into account their popularity) will be presented.

3.3.1 Feedback of the learning process

One way to classify the different machine learning approaches is with the type of feedback that is given to the machine during the learning process:

- *Supervised learning:* In this approach, the learning is done by supplying example data (training set) to the machine in which both the input and the output are given. The machine will train itself to fit as well as possible the training set.
- *Unsupervised learning:* Here, on the other hand, the training set is composed of raw data only, without the solution of what the output should be for a given training example. The computer must therefore figure something out of this data by itself. This concepts sound vague, but is better understood with an example: Imagine that you knew the principle of seasons, but that you were never explicitly taught that the days are shorter during the winter. By experience, you would, after some time, start to identify a pattern by associating shorter days with the cold temperatures and therefore with the winter. This association is likely to happen naturally, in a very instinctive process. Implementing this type of behaviour into a machine is the goal of unsupervised learning.
- *Reinforcement learning:* Is done with no training data at all. The principle is let the machine act on the problem environment. When the machine executes an action, it gets a reward or a punishment. For this method, a human-designed *interpreter* observes the output of the agent, and gives rewards accordingly. The machine trains itself in order to maximize the reward.

In the context of the wake sensing problem, since we are working with a simulator, we know the input (the physical effects of the wake) and the corresponding output ((y, z) position) of the machine learning algorithm. Therefore, supervised learning is the most adapted solution. If, however, the learning was performed directly on a flying drone, for which we therefore do not know the output, reinforcement learning based on a positive reward for a smaller drag would be the way to go. In the next sections, a few machine learning algorithms will be introduced and confronted with the wake sensing problem.

3.3.2 Q-learning

The Q-learning algorithm is part of the reinforcement learning class. The basic principle of this algorithm is to learn to evaluate the outcome of a certain action on a certain state. To do this, it will learn an evaluation function over the [state, action] couples encountered while evolving in the problem's environment by observing the result of the output of that action on the state. This evaluation function is called the *Q-function*, noted $Q(s, a)$. Its value is given by the sum of the reward received after applying action a on state s (noted $r(s, a)$) and the value of following the optimal set of actions from the newly attained state to a goal state (noted V^*):

$$Q(s, a) = r(s, a) + \gamma V^* = r(s, a) + \gamma \max_{a'}[Q(\delta(s, a), a')]$$

Where γ is a discount factor and $\delta(s, a)$ is the state attained by applying action a on state s. In practice, the algorithm works by initializing all values of $Q(s, a)$ to zero. Then, it starts from the initial state designed by s, and loops forever on the following steps:

- Select an action a and execute it
- Observe obtained reward r and the new state s'
- Update the value $Q(s, a) \leftarrow r + \gamma \max_{a'}[Q(s', a')]$
- Update state $s \leftarrow s'$

The main problem with this algorithm is that it will therefore simply learn to evaluate every couple [state, action], which is not only impractical when a large number of these couples exist, but also impossible when faced with non-discrete states and/or actions (as for the wake sensing

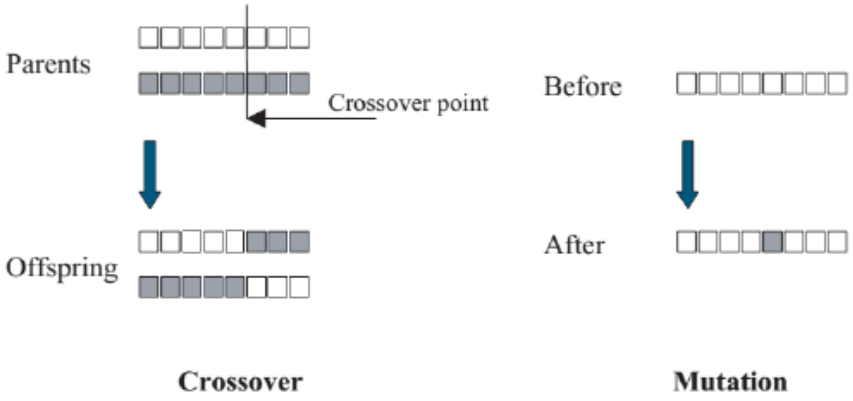


Figure 3.1: Illustration of crossover and mutation [15]

problem). Moreover, its approach is entirely based on testing every action to compute the value, there is no added "cleverness" by, for example, developing an estimate of the reward that will be obtained with (s, a). Also (but this is general remark about reinforcement learning), since the algorithm will in fact learn to optimize its reward, and will therefore learn that reward function, this function must be chosen wisely so that it indeed corresponds to the most effective way of reaching a goal state.

3.3.3 Genetic algorithms

The principle of genetic learning algorithms is inspired by the biological mutations of DNA that generate natural selection and evolution. The idea [27] is to start with a population of possible solutions to a problem, and to perform on them several genetic modifications to alter them in order to find the optimal solution. The idea of the algorithm is that while a certain stopping condition (a certain quality of the new population) is not met, the population is modified in four steps:

- *Selection:* select individual solutions that are "worth" reproducing because they fulfill certain objectives (they have a certain asset).
- *Crossover:* (see figure 3.1) merge the genetics of the selected individuals two by two (create children from them, hoping that the children will combine both assets from their parents).
- *Mutation:* (see figure 3.1) change randomly one (or a few) of the "genes" of the solutions hoping that these random modifications will improve their behaviour. This is important as it preserves genetic diversity.
- *Sampling:* create a new population based on the previous population and their offspring (the new solutions generated by crossover and mutation).

In some way, genetic algorithm can be seen as a form of search algorithm, as it does not explicitly "learns" anything, but rather cleverly selects the individuals with the most potential amongst the population it generates at random. Again, genetic algorithms are not particularly adapted to non-discrete problems, and therefore not ideal for the wake sensing problem.

3.3.4 Artificial Neural Networks (ANN)

This approach consists in implementing a network of nodes comparable to how neurons are connected in the human brain. It is best explained as in [25] with the help of a schematic, in figure 3.2. It represents the typical organization of an artificial neural network. Each node represents a *network unit* also called a *neuron*, and the arrows represent their inputs/outputs.

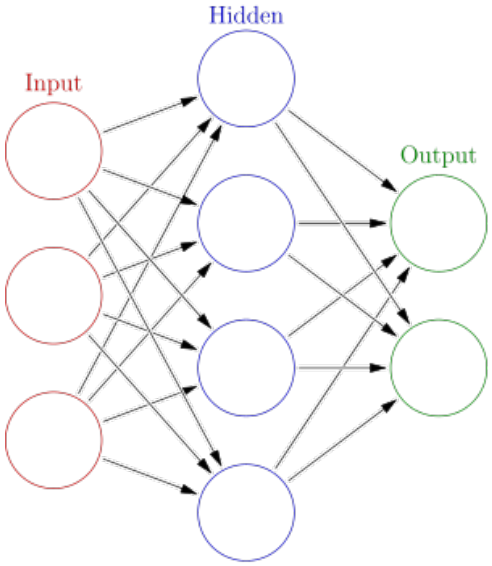


Figure 3.2: Typical organization of an artificial neural network, source: wikipedia

The "hidden" neurons are called so because neither their input nor their output is connected outside of the neural network. Sometimes, there are multiple layers of hidden neurons. Each neuron in the network will compute an output value based on its inputs. The process of learning will dynamically attribute weights to each of these inputs to compute a correct output.

ANNs are particularly suited for problems in which the training data correspond to sensor data that are complex to describe mathematically, such as cameras or microphones. Moreover, they are very powerful for problems that consist in learning a function mapping from one space to another. Finally, neural networks are able to deal with both discrete and continuous input and output spaces. All this makes them a good choice for the wake sensing problem. In the next chapters, more details about NNs will be given, and the algorithm will be tested on a simplified wake sensing problem.

Chapter 4

ANN for Wake Sensing

The conclusion of chapter 3 was that artificial neural networks were a good candidate for wake sensing. In this chapter, we will first present the framework of the simplified problem on which the neural network will be tested. Then, we will go in more details in the structure of ANNs and how to make choices for the parameters on which can be played.

4.1 Simplified Problem

Because the simulator is too slow, performing training directly on the simulator will be slow too. Moreover, the simulator takes into account complex situations, with vehicle dynamics and noisy sensors. While ANN are inherently powerful at working with noisy data, this will also slow down the training quite a bit. Finally, Dronekit presents some problems to keep a steady altitude due a serious bias in its altitude sensor when up high, and the solution to this problem was not found yet. All this is why in the context of this work, the ANN will be used on a simplified problem, to provide a proof of concept.

This simplified problem neglects vehicle dynamics. The drones are considered perfectly steady, and the follower "jumps" from one position in the wake to the next. It also considers situations where both the leader and the follower have the same attitude. Also, just like the physics added in the simulator, no deformation of the wake is taken into account. In this context of perfectly steady flight where the control surfaces counterbalance the effects of the wake perfectly, there is a direct mapping between these effects and the deflection of the control surfaces. Therefore, the inputs of the neural network will simply be the three effects described in chapter 2 (β, α_i, rolling moment), and the outputs will be the (y, z) position in the wake.

4.2 Types of neurons

An artificial neural network consists, as explained in chapter 3, a series of numerical units called *neurons*. These neurons can be of many different types, depending on the type of inputs/outputs of the ANN, and the kind of function that the ANN will be representing. All existing types of neurons will not be described here, only the most frequent in the literature.

4.2.1 Perceptron

The very first artificial neuron was proposed by Warron McCulloch and Walter Pitts [16], and was first called a *Threshold Logic Unit*, later renamed as perceptron. Its structure is illustrated in figure 4.1. Every input of the neuron is linked with a weight. These weights are being tuned when the neural network learns. To generate an output, the perceptron first computes the weighted

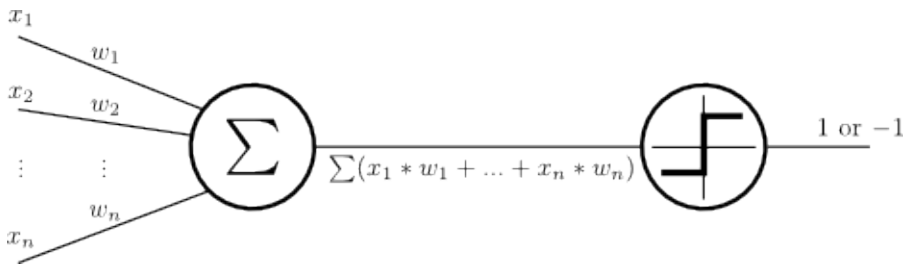


Figure 4.1: Perceptron

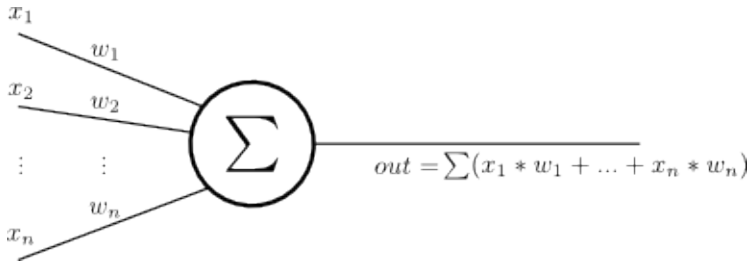


Figure 4.2: Linear neuron

sum of the inputs, and then passes through a *sign*() function, which outputs 1 or −1 if the weighted sum is greater or smaller than zero.

Because the output of the neuron is binary, it is particularly adapted to classification problems, where the output sought for is black or white. In the case of wake sensing, the output of the ANN should ideally be a set of two continuous values, so this type of neuron is not adapted. However, one could use a binary representation for the output and use perceptrons, but this would imply having a big number of output neurons to ensure a minimal precision, making it impractical.

4.2.2 Linear neuron

A linear neuron is simply a perceptron where the second part (the *sign*() function) of the neuron has been removed (see figure 4.2). Therefore, the output is directly the weighted sum. This is already more adapted to the wake sensing problem since the output is continuous, but has one main drawback: if an ANN is made of linear neurons only, the network's input-output relationship will always be a linear function. The problem presented by the wake sensing being non-linear, an ANN with linear neurons only is not adapted. However, we can take advantage of their unbounded output to use them as output layer of an ANN with other types of neurons in the input and/or hidden layers.

4.2.3 Sigmoid neuron

In a sigmoid neuron, the *sign*() function of the perceptron has been replaced by a sigmoid function (fig 4.3), defined by:

$$\sigma(x) = \frac{1}{1 - e^{-x}}$$

The output of this sigmoid function is bounded between 0 and 1 while being continuous and non-linear. This is therefore more interesting for the wake sensing problem. Moreover, it is a differentiable function, an interesting feature for the *back-propagation* training algorithm described later in section 4.4.

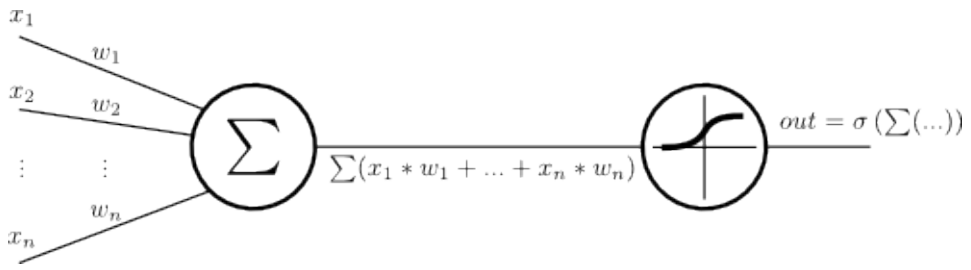


Figure 4.3: Sigmoid neuron

4.2.4 Hyperbolic tangent neuron

The hyperbolic tangent neuron is very similar to the sigmoid neuron, to the sense that the tanh function replaces the sigmoid function in a sigmoid neuron. The tanh is actually a stretched version of the sigmoid function, and has outputs that vary between -1 and 1. It is also differentiable, and therefore adapted to backpropagation.

4.2.5 Internal NN organization

To chose the internal representation of the Artificial Neural Network, or, in other words, which type of neurons will populate each of the layers of the neurons, one need to look at which problem the ANN will have to tackle. Sigmoid and tanh layers have the important advantage of being able to deal with non-linear problems. Therefore, if working with a non-linear problem, as wake sensing for example, at least one layer of the neural network should be equipped with nonlinear neurons. When facing highly non-linear problems, multiple layers of nonlinear neurons might help.

The type of input and output the ANN will be submitted to is also important. When the input or output (or both) are continuous unbounded values, the input or output layers are best equipped with linear neurons, as they will be able to treat these inputs and outputs. On the other hand, when facing classification problems, which should have binary outputs, the output layer should be made of one of the bounded-valued neurons. A very used type of neuron for classification problem is the *softmax*, which is not covered here because it is out of context.

From this section, one could conclude that a good starting point for the wake sensing problem would be a neural network with linear input and output layers, to satisfy the type of data, and a sigmoid or tanh hidden layer to satisfy the non-linearity. It might also be interesting to explore multiple nonlinear layers to see if they might bring more precision (in other words, to check if the wake sensing problem is highly nonlinear).

4.3 Training set representation

In neural networks, and in AI problems in general, it is important to chose the data representation wisely. In AI, this can lead to very big reductions in necessary computing power. In neural networks, it might simply lead to a neural network being able to learn, or not. In the case of wake sensing, there is little play in the representation of the data. The straightforward representation would be to use both continuous inputs and outputs on the neural network (and this is what we will do in this work). But other representations could be possible. For example, the (y, z) position in the wake could be represented in polar coordinates instead of Cartesian. Another idea could be to represent the output as the two distances between the follower and the two energetically ideal positions in the wake.

Another very important point about data set representation in neural networks, is called *normalization*. Normalizing data means to ensure that their ranges are comparable. As an example, consider a neural network used to find the relationship between temperature and material dilation. If the inputs are a temperature sensor in $°C$ and a deflection sensor in km, one sensor will send readings in the range of tens of degrees, and the other in $1^{-6}km$. With such a difference in range between both sensors, the neural network will have the tendency to neglect the km sensor. Normalizing the input data could be done by remapping the deflection sensor from km to mm, which would prevent the NN to focus on the temperature input.

The inputs to which the neural network of the wake sensing problem will be confronted have ranges of comparable scales, and should therefore not need remapping. However, if changing the output layer from linear neurons to non-linear layers, the output data of the training set will need to be remapped to $[-1, 1]$ for tanh neurons and $[0, 1]$ for sigmoid neurons. Otherwise, the neural network simply will not be able to learn from the training data as he will be unable to even output the training values.

4.4 Training rule: Backpropagation

Training a neural network consists in tuning the weights of the inputs of every neuron in the network. The objective of this process is to minimize the error between the output of the neural network and the training data it trains on. For multi-layer neural network aimed at learning nonlinear functions, backpropagation is the most and massively used training algorithm. Although it is not guaranteed to converge to the global minimum of the error space, it produces usually in practice very good results [25]. It is the rule that will be used to train the neural networks of this work later in chapter 5. The backpropagation algorithm can be seen as a generalization to multi-layer networks of the gradient descent rule; a rule applied to a single neuron.

This section uses the notations and definitions of [25].

4.4.1 Gradient descent

This section will not be a thorough description of the gradient descent rule as it is a very classic minimum-seeking method. The focus will be made on applying the rule on a neuron (a linear one for this example taken from [25]). As a reminder, gradient descent consists in computing the gradient of the error in function of the parameters of the error, to find the steepest slope. Then, tune the parameters to follow that slope until reaching a local minimum. For a linear neuron, the output is given by:

$$o(\vec{x}) = \vec{w} \circ \vec{x}$$

Where $\vec{x}$ is the vector of the inputs of the neuron and $\vec{w}$ the vectors of weights associated with the inputs. The gradient descent rule adapts the weight according to:

$$\vec{w} \leftarrow \vec{w} - \eta \nabla E(\vec{w})$$

Or, for each single weight:

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}$$

Where η is a positive constant called the *learning rate*, and $E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$ is the *training error*, with t_d, o_d being the target value and real output of the neural network for training example d in training set D. The gradient descent algorithm works as follows:

1. Generate all outputs for each training example

2. Compute the derivatives of the training error with respect to each input
3. Update the input weights according to the rule above

4.4.2 Backpropagation

To generalize the gradient descent rule to a multi-layer network, one first needs to redefine the training error to take into account the multiple outputs of the neural network:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

Where D is the set of all training examples and *outputs* the set of all output units of the network. Backpropagation has one main difference with gradient descent: instead of computing the error over all training examples to then adapt the weights, backpropagation adapts the weights after each training example. So, coming from the gradient descent rule, every weight in the neural network will be adapted via:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E_d}{\partial w_{ij}}$$

Where w_{ij} designates the i^{th} weight of neuron j and E_d is the training error of one single training example d:

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

To derive the backpropagation rule, a difference must be made between output neurons and hidden neurons. This is because output neurons influence the output directly, whereas hidden neurons only have influence on the part of the neural network situation downstream to them. For each training example, each neuron (hidden or output) will be adapted via:

$$w_{ij} \leftarrow w_{ij} + \eta \delta_j x_{ij}$$

Where δ_j is called the *error term* associated with neuron j. It is this term that will be different for output and hidden neurons. For sigmoid output neurons, we will have :

$$\delta_j = (t_j - o_j) o_j (1 - o_j)$$

And for sigmoid hidden neurons, we have:

$$\delta_j = o_j (1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

With $Downstream(j)$ being the set of units immediately downstream of neuron j. The details of the calculus of these error terms are given in annex C. One important point to notice is that in the calculations of the δ, the derivative of the neuron's function appears. This is why nonlinear neurons are usually chosen with a sigmoid or tanh output function: they are easily differentiable and therefore very practical for a use in the backpropagation algorithm.

4.5 Number of hidden neurons and layers

The goal of this section is to explore ways of determining the ideal number of hidden layers and the number of hidden neurons in an artificial neural network to reach the best possible accuracy while avoiding *overfitting* the training data. Overfitting means that the neural network will memorize the training data. When this happens, the NN will be accurate when tested with the training data, while performing poorly on other data. This is why it is important to work with a training set and a *validation* set. The latter is used as a test to evaluate the error on data on which the NN did not train. No overfitting has appeared if the errors for the training and validation set are comparable.

4.5.1 Hidden layers

The number of hidden layers that a neural network should use depends mainly on the type of problem. Linear problems often do not require a hidden layer. Non-linear problems, like the wake sensing problem, require at least one hidden layer with non-linear neurons. Only when the problem is highly non-linear will multiple hidden layers prove useful. For the wake sensing problem, as we will see in chapter 5, a single hidden layer will be enough.

4.5.2 Hidden neurons

The choice of the number of hidden neurons, however, is not as straightforward. Indeed, there is no universally adopted method giving the best possible results for each problem. An overview of many proposed solutions has been made in a literature review by Gnana Sheela and Deepa in [23]. Most often, in the reviewed articles, the proposed solution gives the best results on the problem covered by the article, but is not necessarily the best for other types of problems. An example of this, is the rule of thumb claimed by Jeff Heaton (Ph.D., data scientist at Sever Institute, Washington University) to be a good starting point for neural network. Heaton claims that the number of hidden neurons should be a number between the number of input neurons and the number of output neurons.

Although this might be a good rule of thumb for many problems, for the wake sensing problem, this would mean that the number of hidden neurons should be in between 2 or 3. And via tests, it was made clear that this is largely insufficient, as the minimal number of hidden neurons for a correct accuracy was found to be around 50. The best results were obtained with 100 neurons. All the tests done in chapter 5 are done with 100 hidden neurons. In practice, to chose the ideal number of hidden neurons, NN designers will often start from a rule of thumb of their liking, and then rely on iterative methods. They change the number of hidden neurons, sometimes even dynamically, until obtaining the best accuracy while preventing overfitting [9].

4.6 Number of epochs

An *epoch* is the term used by ANN scientists to indicate one complete training iteration over the complete training set. Again, there are multiple methods to choose when to stop training a neural network. One straightforward choice is to stop when the neural network has attained a certain determined accuracy [25]. This approach, however, can lead to potentially infinite training times when the designed neural network can not reach the required accuracy. This could happen for example with training algorithms that do not implement mechanisms to prevent being stuck in local minima. Another approach, which is the one used for the wake sensing problem in this work, is to stop when the training error converges and displays only minimal improvements from one iteration to the next [9]. For all tests in chapter 5, the number of iteration was fixed at 1000 epochs, as it was observed by testing that for any variation of the neural network (number of hidden neurons, type of neurons, number of inputs, ...), the convergence was attained well before 1000 epochs.

Chapter 5

Results

This chapter presents the results obtained by training neural networks on the simplified wake sensing problem. Different network configurations are tested, and improvement clues are tested by adding a simulated sensor. The details of the programming of the simplified problem and the neural networks are given in annex D.

5.1 PyBrain

These results were obtained by using the PyBrain Python library [22]. This library is the result of a collaboration between a team from the university of Lugano and a team from the Technische Universität München. It is an easy to use machine learning library which implements, amongst others, multidimensional neural networks.

5.2 Results

This section will present the results obtained with the PyBrain library on the simplified problem described in section 4.1. By default, a neural network created with PyBrain has linear neurons in the input and output layers, and a single hidden layer with sigmoid neurons. To analyze the efficiency of the different tested neural networks, we will analyze the error between the output of the neural network (where the follower drone thinks he is in the wake) and the target output of the data set (where the drone actually is). This error will be analyzed via classical statistical tools (mean, standard deviation, ...) and via a heat map plots, to observe in which zones of the wake the neural network is (or is not) accurate. The used data sets all have a range extending from $(y, z) = (-5m, -2.5m) = (-2.65b, -1.33b)$ to $(y, z) = (5m, 2.5m)$ with a discretization of $0.1m$. This was chosen arbitrarily to encapsulate both energetically ideal positions (see section 2.5) and the significant effects of the wake.

For more details about the organization of the Python programs that were developed to generate the results, please refer to annex D.

5.2.1 Default PyBrain NN

In this first part, we will observe the behaviour of the default PyBrain NN with 100 hidden neurons. We will also use the straightforward data representation: since the output layer is linear, it should not be necessary to normalize it.

With 1000 epochs, we can observe in figure 5.1 the mean error of the neural network on the data set. The red line approximates the evolution of the mean error. After 1000 iterations, the NN keeps improving, but the evolution is very slow, and is not significant anymore when

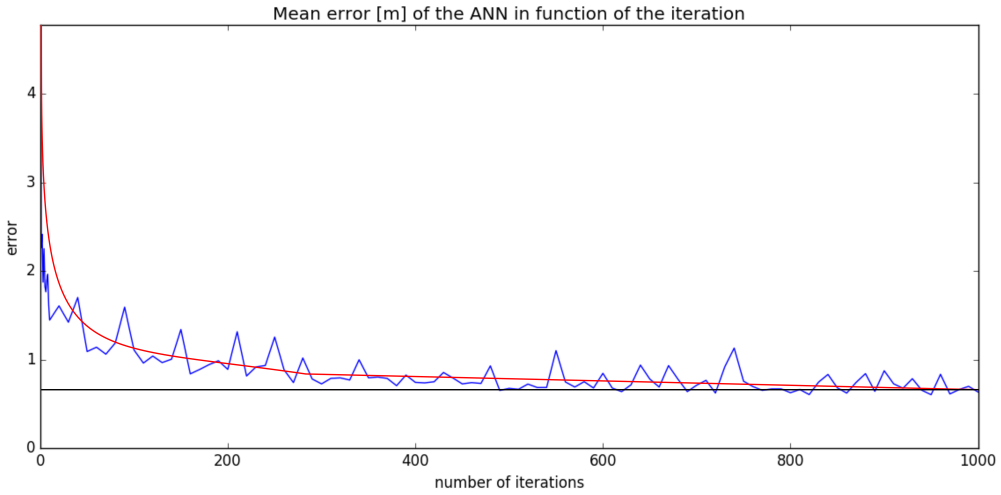


Figure 5.1: Mean error over 1000 epochs

compared to the computing power required (one epoch takes approximately 8 seconds on an intel core i3-2310M laptop). The most accurate neural network was obtained at iteration 950. The error heatmap of this network is shown in figure 5.2. The average error of this neural network is of $0.6m$ with a std deviation of 0.76 and a variance of 0.58.

On the heat map, interesting shapes can be observed. These are linked to the fact that in the zones in which the error is the worst (green, red and white zones), the three input data (α_i in figure 2.8, β in figure 2.11 and the rolling moment in figure 2.15) are symmetric. Indeed, in these zones, the three inputs are close to zero, making it nearly impossible for the neural network to discriminate one corner from the other. Interestingly, the visual look of this error plot is comparable to what was obtained in [8] by Levi DeVries and Derek Paley.

In their article, they analyzed the observability of the wake using the method of wing-distributed pressure sensors for wake sensing. One of their results is illustrated in figure 5.3, and shows the same kind of pattern in the corners of the figure than with the neural network. This suggests that without additional sensors, the neural network will have a hard time discriminating one corner from the other.

In a more realistic environment, such as the enhanced SITL, turbulences and other parasitic atmospheric effects would be blended with the effects of the wake. The symmetry seen in the tests on the simplified problem would then worsened, and the zones of lacking accuracy would increase in size.

The learning process is illustrated in figure 5.4. In the very beginning, the neural network starts (fig 5.4a) with randomly assigned weights. Therefore, the output of the neural network is completely wrong, showing a white picture, synonym of errors above $3m$ almost everywhere in the wake. in figure 5.4b, we see the result of the first training epoch: the error plot starts to show the characteristic shape of the trained network. After 10 iterations (fig 5.4c), the corners are starting to become symmetrical, and some zones of the wake start to show reasonable accuracy. Finally, after 300 iterations (fig 5.4d), the neural network is already accurate enough, with some zones (the green zone on the right) still displaying a lack of accuracy.

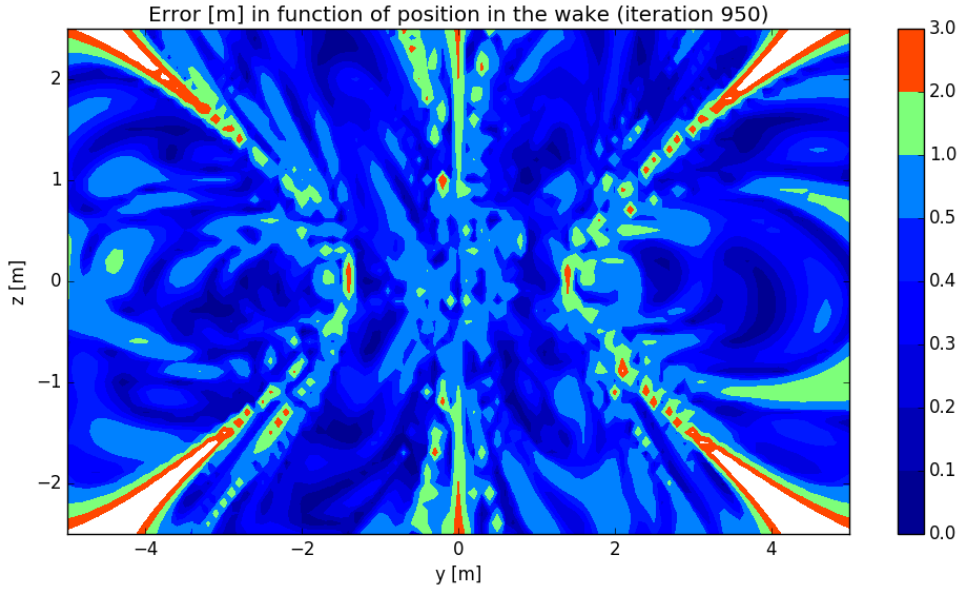


Figure 5.2: Error heatmap of the default PyBrain ANN after 950 iterations on the straightforward data set (white zones present an error higher than the maximum of the color scale)

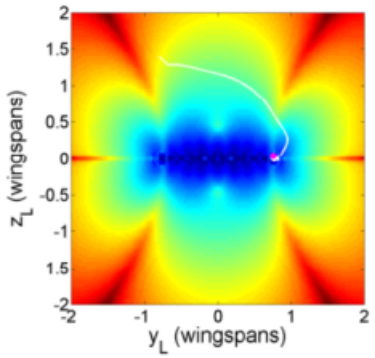
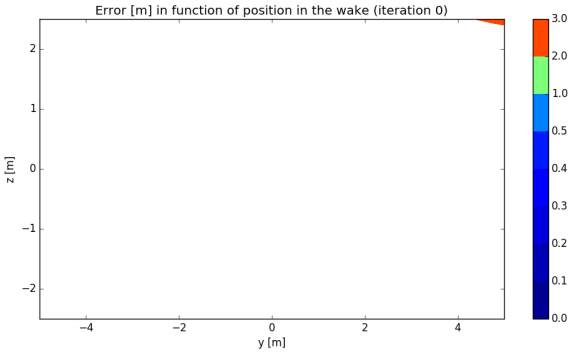
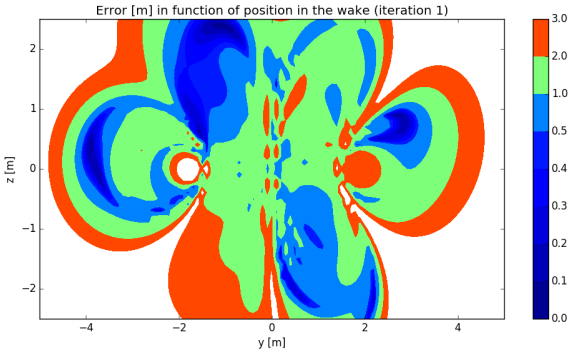


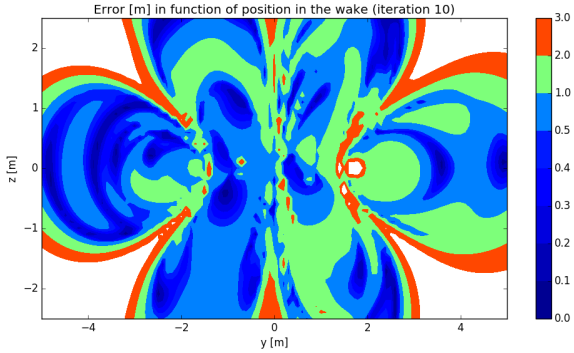
Figure 5.3: Unobservability index of wake sensing using wing-distributed pressure sensors [8]



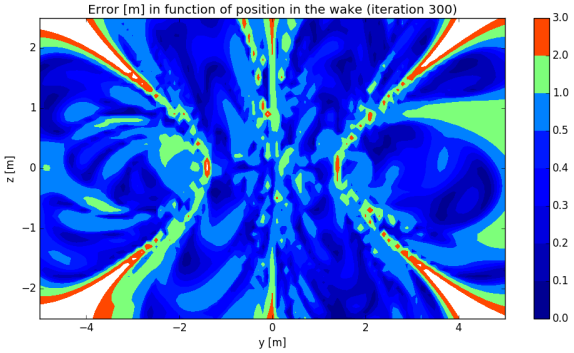
(a) Iteration 0



(b) Iteration 1



(c) Iteration 10



(d) Iteration 300

Figure 5.4: Evolution of the neural network's error heatmap during the learning process

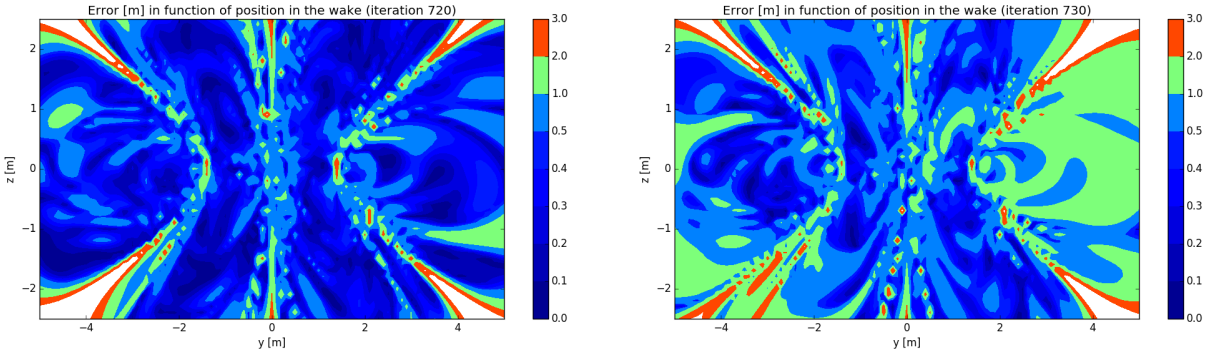
An interesting observation to make when looking back to figure 5.1 and the peaks that appear, is that the iterations sometimes make the neural network considerably worse than it was before. This happens for example between iterations 720 and 730, as shown in figure 5.5. The reason for this is not confirmed, but might be due to the neural network going back when it started to overfit the training data.

5.2.2 Tanh hidden layer

Using tanh neurons on the hidden layer does not require to normalize the training set, since the input and output layers are still linear. In practice, the difference with the sigmoid hidden layer is not significant. The only difference being that for a same number of epochs, the tanh neural network usually presents worse error values than the sigmoid NN. After running 1000 iterations on the tanh net, the best result was obtained at the 980^{th} iteration (see figure 5.6), with a mean error of 0.92, a standard deviation of 0.82 and a variance of 0.68.

5.2.3 Tanh hidden and outer layer

By using tanh neurons on the outer layer, one limits the outputs of the ANN to bounded values between -1 and 1. Therefore, using the straightforward data set will not be possible as it contains outputs varying from $(y, z) = (-5, -2.5)$ to $(y, z) = (5, 2.5)$. The data will thus require normalization. Here, it was done by linearly reducing the output data's range to a range extending from $(y, z) = (-1, -0.5)$ to $(y, z) = (1, 0.5)$. No normalization is performed on the inputs of the neural network, since the input layer is still linear. With this normalization, over 1000 iteration, the best neural network was attained at the 960^{th} iteration (see figure 5.7).



(a) Iteration 720

(b) Iteration 730

Figure 5.5: Illustration of 10 iterations worsening the neural network's accuracy

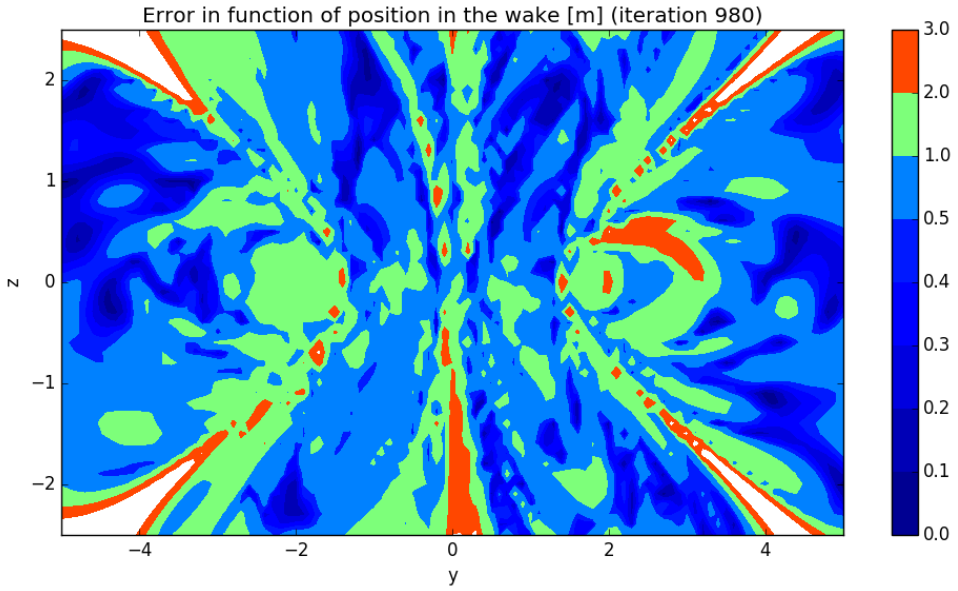


Figure 5.6: Error heatmap of the PyBrain NN with hidden tanh layer after 980 iterations

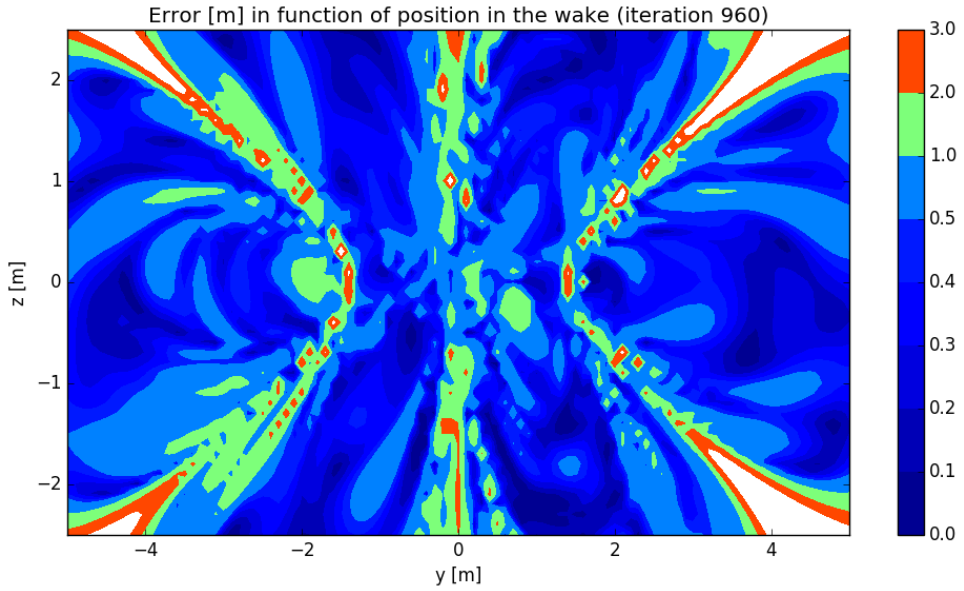


Figure 5.7: Error heatmap of the PyBrain NN with hidden and output tanh layers after 960 iterations

The interest of having two layers of nonlinear neurons is that more complex and highly nonlinear problems can be approached with this type of network. The results found with this double layer of tanh neurons shows very comparable accuracy to the default PyBrain network, with a mean error or 0.71, a standard deviation of 0.9 and a variance of 0.81. This means that, in the context of this problem, a single layer of nonlinear neuron is sufficient. Having multiple nonlinear layers will just increase the demand in computational power. Indeed, one epoch on this double nonlinear layer ANN took 12 seconds in average on the same computer that took 8 seconds in average for a single nonlinear layer.

5.2.4 Polar data representation

Since the wake physics depend on the distance between the follower and the centers of the two vortices generated by the leader, a polar representation for the output of the neural network could prove interesting. This representation was tested on the default PyBrain neural network during 1000 epochs. The best NN was obtained at iteration 950 (see figure 5.8) with a mean error of 1.57, a standard deviation of 1.77 and a variance of 3.15. This is obviously much worse than the best NN obtained with Cartesian representation. However, the same geometric pattern appears, showing that the difficulty the neural network faces in the corners of the wake is not related to the Cartesian representation, but rather to the wake's properties.

Interestingly, another zone of very low accuracy (along the y axis, to the left of the center of the figure) appears when using polar coordinates. This zone is absent of the right part of the wake, indicating that this is due to the fact that around the left part of the y axis, the angular coordinate of the polar representation switches from $-\pi$ to π. Along this part of the y axis, there is a discontinuity in the function the neural network is trying to learn. Standard neural networks are unable to learn discontinuous functions, it is therefore not surprising to observe the results obtained here. There are however methods allowing neural networks to work with discontinuous training data [14], but these are far beyond the scope and interest of this work.

For the sake of completeness, another test was made with a double sigmoid layer to check if the difficulty that faced the NN did not come from the fact that polar coordinates make the

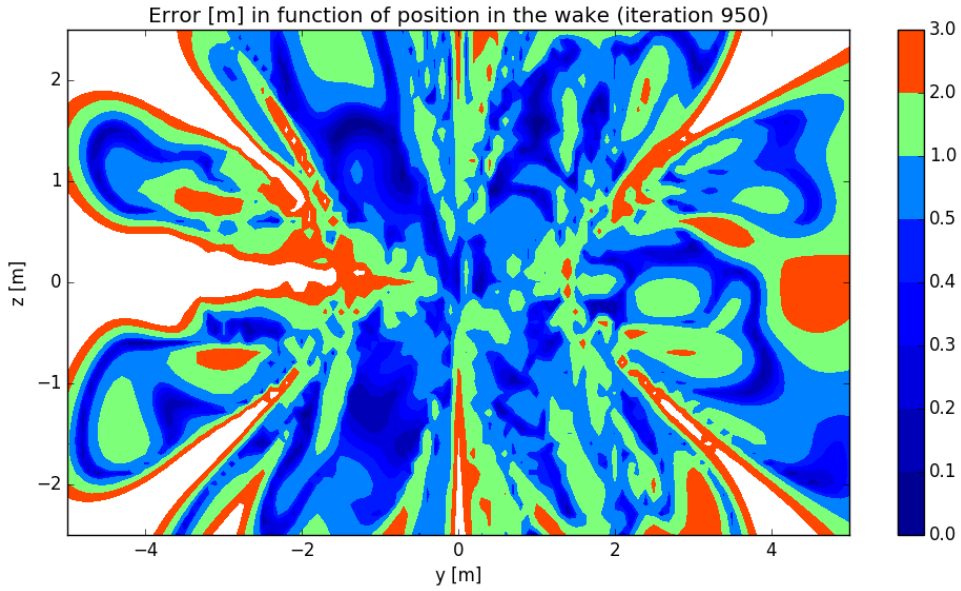


Figure 5.8: Error heatmap of the best NN obtained with polar coordinates (iteration 950)

problem even less linear, but the results obtained with this new NN were considerably worse than with the standard NN, so they are not shown here.

5.2.5 Adding a rough camera sensor

The neural networks shown in the previous sections are not able to discriminate on their own one corner of the wake from the other. Moreover, they present weak spots close to the energetically ideal locations (as illustrated in figure 5.9). These local weak spots are due to the fact that the neural network does not manage to discriminate them. When observing the output data, one can notice that in these zones, the neural networks thinks he is on the other side of the leader. This is again due to symmetrical zones in the three input data. Therefore, adding a sensor that does not present these symmetricality issues might be a good idea to improve the precision of the neural network.

In this section, the influence of adding a camera sensor will be tested. This influence shall be tested by simulating a very bad camera sensor that will only allow the follower to know in which quadrant of the wake he is situated. The roughness of the simulated camera is necessary to be able to observe if it allows the NN to discriminate the corners, while having a minimal influence on the rest of the learning. Indeed, by simulating a very good camera sensor (that is, a sensor which gives the real position of the leader), the NN will very likely base his learning solely on the input of the camera. In such a situation, all other inputs, which are the main interest in this work, will be neglected. In practice, we add two inputs to the neural network; one of these inputs will take the value -1 if the follower is on the left of the leader and 1 if he is on the right. The other input will take the value -1 if the follower is below the leader, and 1 if he is above. With this quadrant sensor added, the error obtained with the default PyBrain neural network after 100 iterations is shown in figure 5.10. Clearly, something is not working.

By looking into the inputs of the training data, we can notice (going back to figures 2.8, 2.11 and 2.15) that the wake inputs have numerical values ranging from 0.16 to -0.16. So, even if the input layer is made of linear neurons, the difference in range between the camera inputs and wake inputs will work in favor of a learning based mainly on the camera. The latter being very imprecise, it is not possible to determine the position solely from it. To allow the neural network

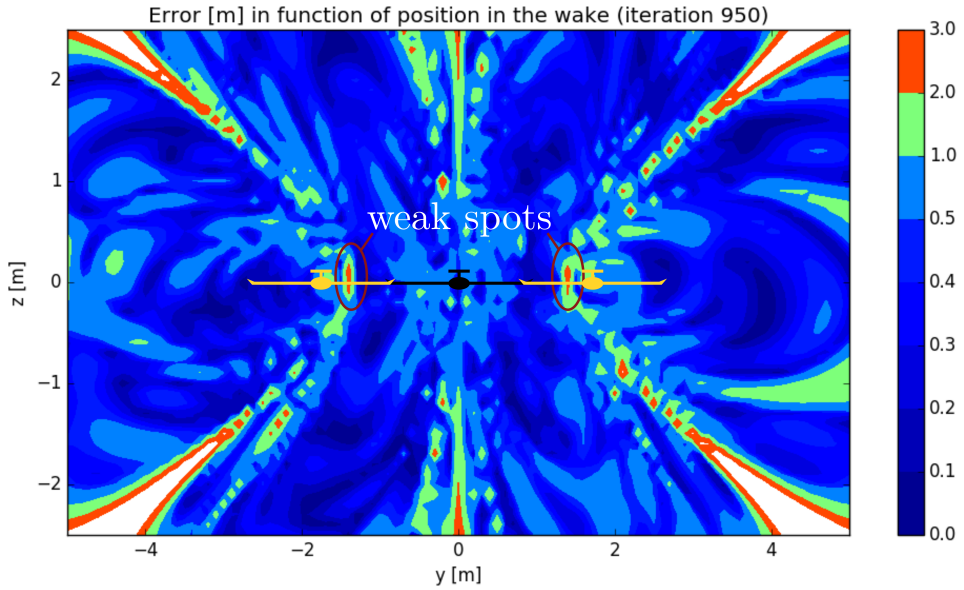


Figure 5.9: Illustration of the weak spots near the energetically ideal locations (yellow drones) (illustrated on the plot of figure 5.2)

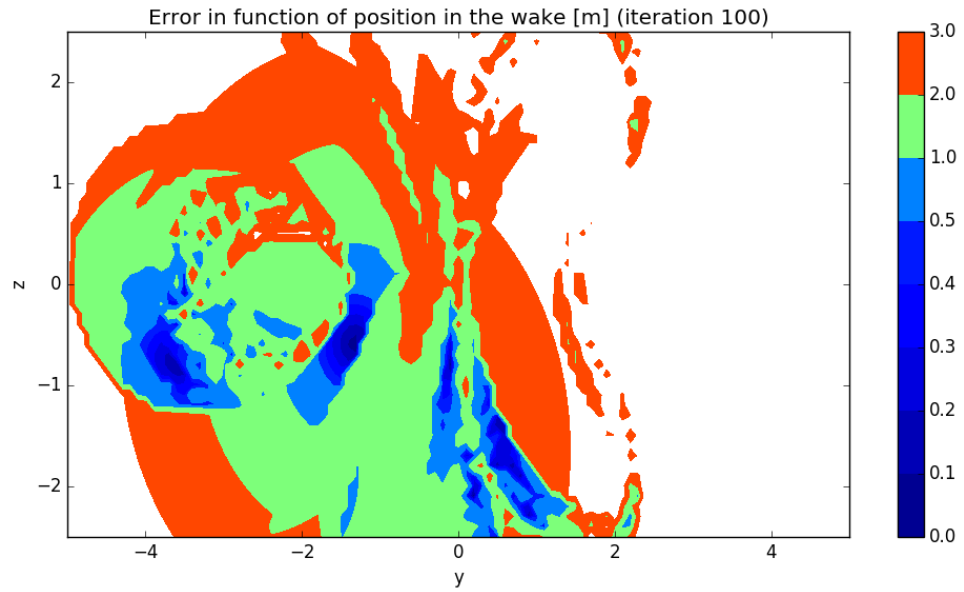


Figure 5.10: Error heatmap of the default PyBrain NN with rough camera input after 100 iterations

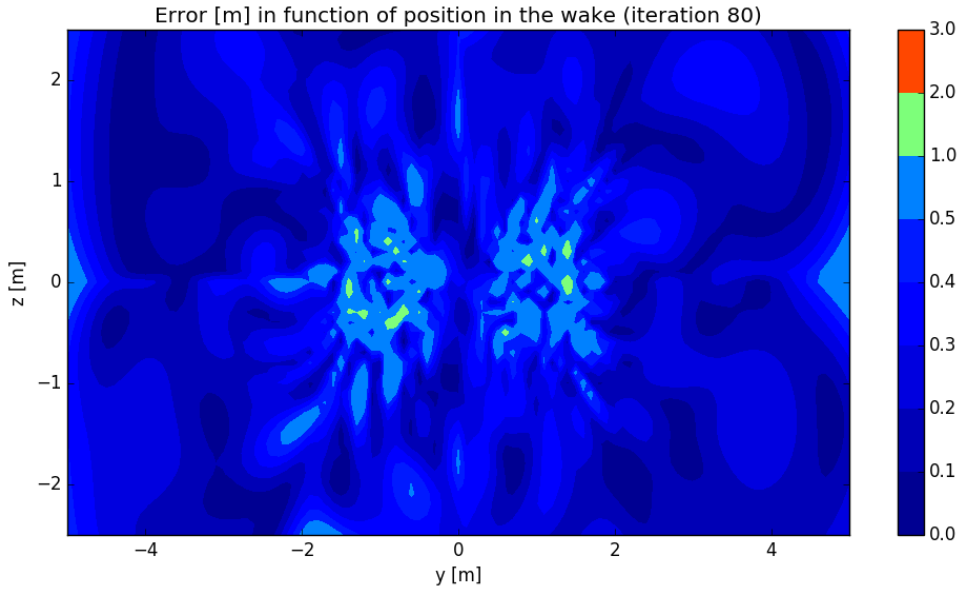


Figure 5.11: Error heatmap of the default PyBrain NN with rough camera input after 80 iterations with input normalization

to work with all its input datas, it is necessary to normalize them. This was done by linearly remapping the ranges of the wake inputs to [−1,1].

The results obtained with the normalization are shown in figure 5.11 after 80 iterations. This time, the learning is done correctly and the results show significant improvements with respect to the camera-less neural network. With the camera, the neural network shows a mean error of 0.24 with a standard deviation of 0.16 and a variance of 0.027. Moreover, the zones in which the standalone neural network showed accuracy problems are not present. This result shows that by combining a neural network that learns from the influence of the wake with an external sensor (as a camera, a GPS position sensor, ...) giving a hint about the follower's position with respect to the leader, we can hope to obtain good results for wake sensing. And the icing on the cake: neural networks are particularly powerful at working with noisy data. Thus, if the added sensor is imprecise and gives now and then wrong information, a correctly implemented neural network should eliminate the noisiness and work similarly to a situation with perfect data [21].

5.2.6 Camera sensor with shifted wake

In real conditions, the wake of the leader can be deformed by atmospheric conditions (see section 1.2). One of the possible deformation is a shifted wake, that could for example be the result of a crosswind, as illustrated in figure 5.12. If this deformation alone were to happen, the standalone wake sensing neural network would perform identically. However, the camera input (and therefore the separation in quadrants) would be shifted (illustration figure 5.13). From experiences, a shift smaller than $\pm b_0$ in y has no influence on the learning abilities of the neural network without camera shift. A shift in z will always influence the learning, although the effect is only significant for shifts above approximately $\pm b_0/2$. When the shift is significant enough, it generates a discontinuity in the function to be learned, exactly at the axis corresponding to the change of quadrant. This generates a region of poor accuracy, as shown in figure 5.14, with a shift of $+1.5m$ in y or figure 5.15 with a shift of $-1m$ in z.

These observations raise another question: what would happen if one of the shifts was very large? In other words, what if the "camera" sensor were to only provide information about

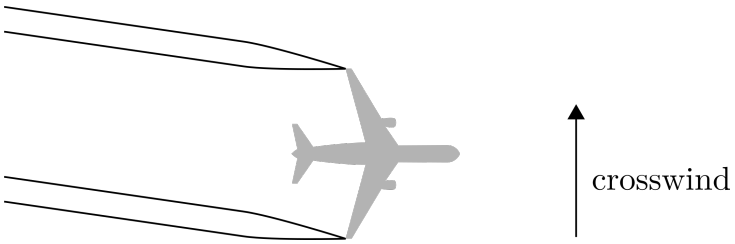


Figure 5.12: Illustration of a shifted wake due to crosswind

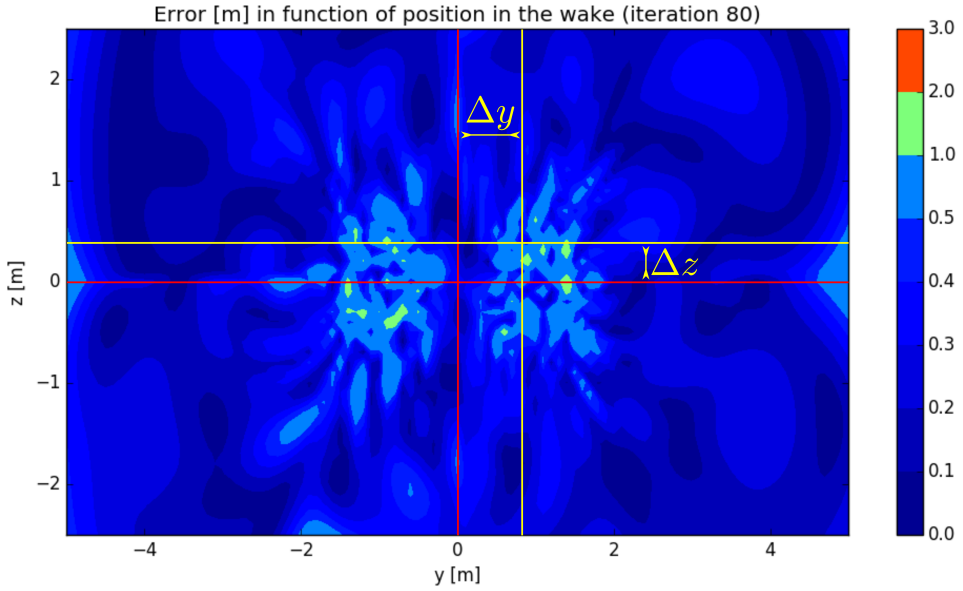


Figure 5.13: Illustration of a shifted quadrant-type sensor (in yellow) on the plot of figure 5.11

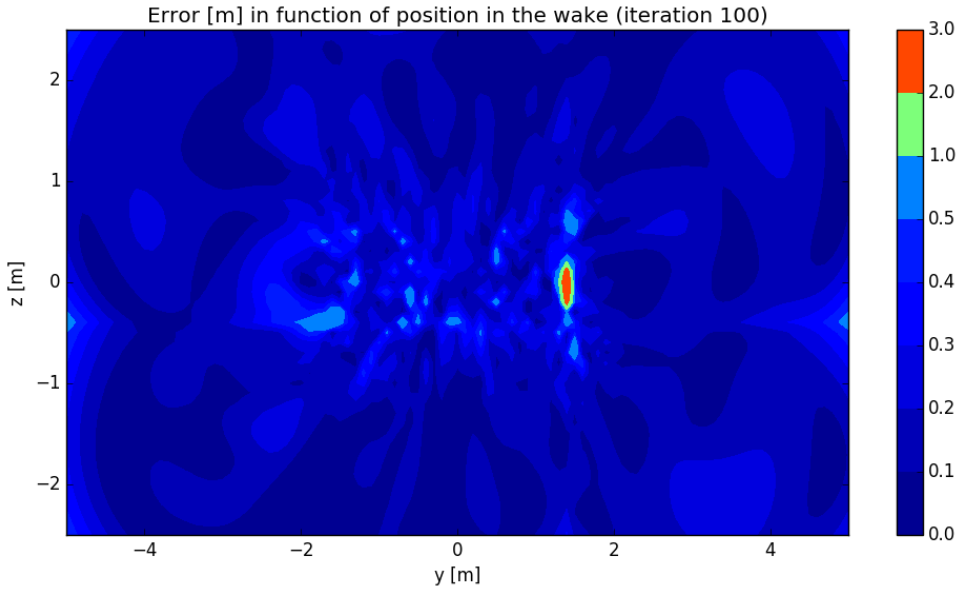


Figure 5.14: Error heatmap of the default PyBrain with $+1.5m$ shift in y of the camera sensor after 100 iterations. Weak region around $y = 1.5$.

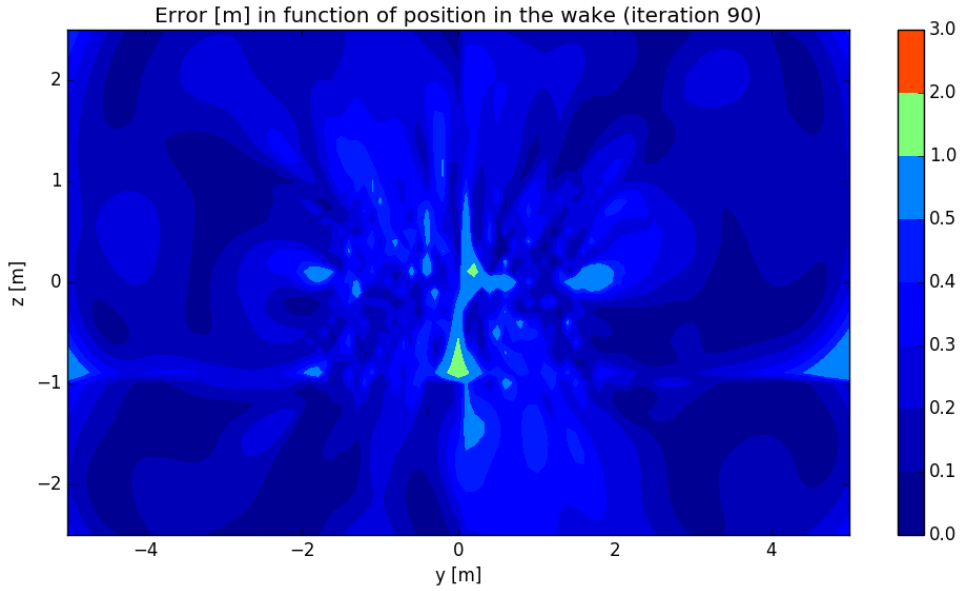


Figure 5.15: Error heatmap of the default PyBrain with $-1m$ shift in z of the camera sensor after 80 iterations. The discontinuity forms a visible line at $z = -1$.

crossing one of the axis instead of indicating quadrants?

Figure 5.16 shows the results of training a NN with a camera that would only give information about the follower's position in y. We can see that, because the neural network has the necessary information to discriminate the regions of the wake that present a symmetry around the z axis, most of the accuracy obtained with the quadrant sensor is maintained. However, on the z axis itself, the neural network faces problems to differentiate the regions above and under the y axis. Nonetheless, this type of sensor would be very useful on its own, as it helps increasing the accuracy along the y axis, where the energetically ideal positions of the follower are.

In the case of a camera sensor giving only information about the position in z, the same reduction of accuracy appears around the y axis. The most important difference with the y-only sensor is the significant reduction in accuracy in the center of the wake (see figure 5.17). It is interesting to link these results with previously shown results: note that the "weak spots" described in section 5.2.5 and illustrated in figure 5.9 are also strongly present on the results with a z-only camera sensor but less so with a y-only camera. Again, this proves that these weak spots appear because of a symmetry in the input data around the z axis.

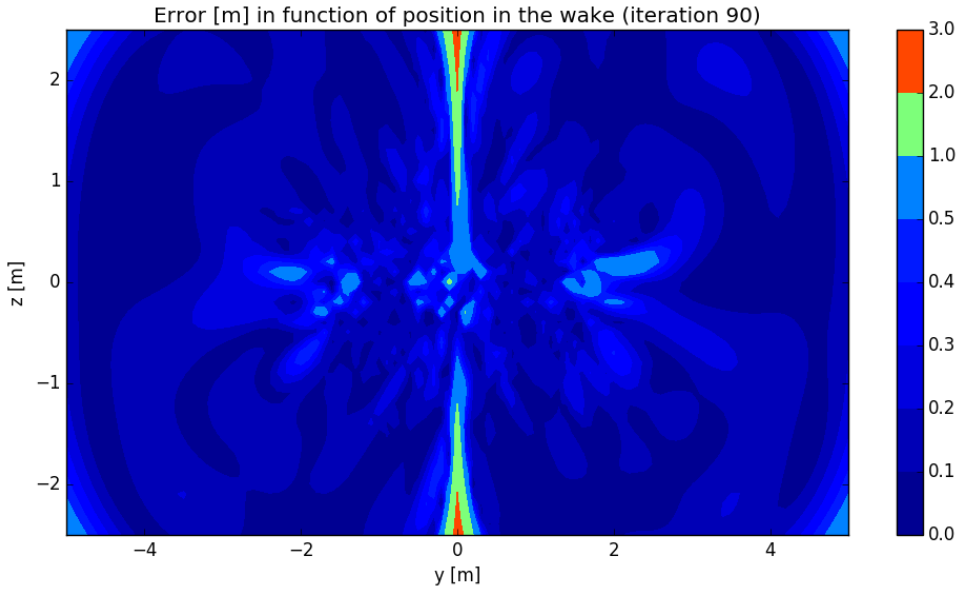


Figure 5.16: Error heatmap of the default NN with only y camera sensor

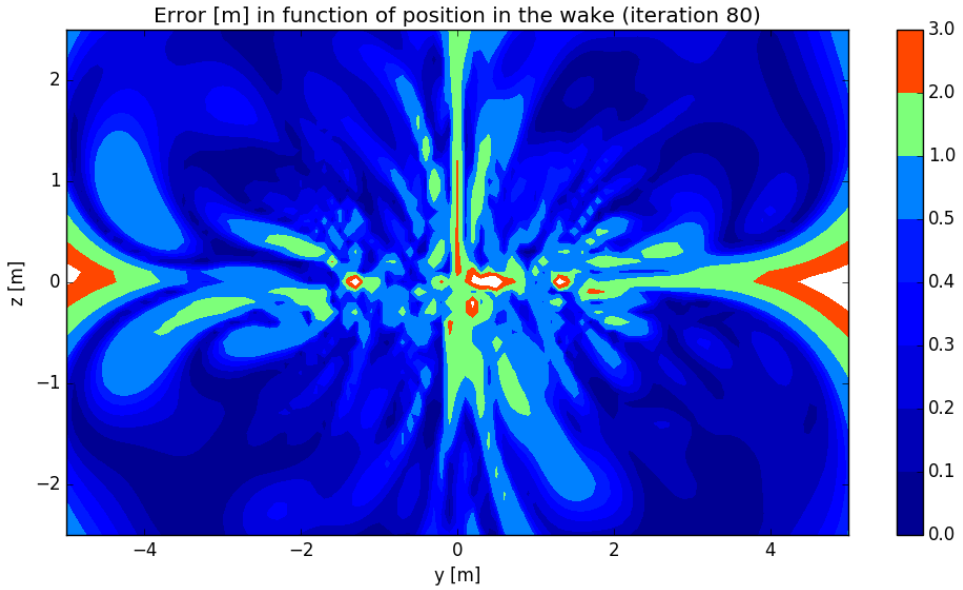


Figure 5.17: Error heatmap of the default NN with only z camera sensor

Chapter 6

Conclusion

In the first part of this master's thesis, an open-source drone simulator was enhanced by adding the physical effects induced by flying in a wake. The steps necessary to accomplish this task were to:

- Read and understand the source code organization of the simulator and the physical model of the flying drone.
- Analyse the physical effect of flying in a wake through its 3 main effects:
  - Change of effective angle of attack
  - Side-slip
  - Rolling moment
- Integrate these effect into the existing code.

In a second time, a proof of concept of utilizing a machine learning algorithm in the context of wake sensing was developed. This was done by going through the following steps:

- Define a simplified situation of the wake sensing problem
- Select a machine algorithm adapted to the problem (Artificial Neural Network)
- Select the ideal parameters for the neural network
- Try out the designed neural network on the simplified problem
- Explore techniques to improve the accuracy of the neural network

The end results showed that despite a possibility of using neural networks for wake sensing as a stand-alone solution without added sensors, some symmetries in the physics of formation flight prevent it to be optimally accurate. However, adding a camera sensor, even to use it only as a quadrant sensor, gives a very interesting increase in accuracy by removing these symmetries.

In the current state, this work provides future works in the field of formation flight and wake sensing with a simulation framework for formation flight, as well as a solution to tackle the problem of wake sensing. Despite the positive results obtained with the simplified problem, this solution would probably need adaptations to work in the complete simulation environment.

6.1 Further Thoughts

In this last section, a few problems will be listed that have been noticed during this work that could hinder future works. Also, ideas related to real-life implementation of the neural network approach will also be presented.

6.1.1 ArduPilot weakness

ArduPilot, despite being a powerful autopilot, has some limitations for formation flight; the main problem being that ArduPilot is made to reach points in space, and not to follow a leader drone. Indeed, there is a significant difference in the altitude that an ArduPilot instance thinks he is flying at, and the actual altitude the drone is flying at in SITL (of around 10m at 100m altitude). This prevents any form of precise positioning of the follower with respect to the leader drone. By finding the place in the source code that generates that bias, one should be able to fix this issue. Ideally, ArduPilot should be using the exact location of the drone in SITL. Unfortunately, this is not straightforward to do, and was not solved in this work.

Another problem faced is that because of the complexity of the computations added to model the rolling moment induced by the wake, the simulator is significantly slowed down. This means that the actual learning will take longer than in the simplified simulation. Except by using a more powerful computer or by resolving to longer training times, there is no easy solution for this problem.

6.1.2 Leader parameters dependence

One more problem, not related to the simulation environment, resides in the fact that the learning performed with the current solution depends on the characteristics of the leader and follower drones. This means that in real life situations, the follower drone having learned to perform wake sensing behind a given type of drone, will not be accurate when using the same neural network while flying behind another type of drone. Obviously, it would be impractical for every drone to learn the wake of every other drone. One improvement clue could be to go back to the physics of the wake, and try to develop another mathematical expression of the physical effects that would be independent of the leader drone's parameters. In some way, this would be a similar approach to performing a nondimensionalization of the learning inputs of the drone. In practice, the follower drone should then be able to know the parameters of the leader drone. Then, the computation would start from the deflections of the control surfaces, and transform them into these new mathematical expressions, independent of the leader's parameters. The latter would then be used as inputs of the wake sensing neural network, removing the training's dependence to the leader's parameters.

6.1.3 Supervised learning in real-life situations

In real life implementations, the main problem is that supervised learning will not be possible since there is no way to know the exact position of the follower in the wake. One way to solve that problem would be to combine the learning with the results of wake sensing with added pressure sensors (see section 1.2.1). If the wake sensing method using pressure sensors were to prove accurate in real-life situations, one could use the computations of the pressure sensors as supervised training set for the neural network. Then, once the learning has been done, the drone would be able to fly without the sensors. Another approach could be to perform reinforcement learning that would try to minimize the need for engine thrust, but this would in fact resume in a comparable approach than minimum-seeking algorithms.

Since the neural network will likely never be perfectly accurate, even with an added camera sensor, there would be some interest in combining the NN approach with a minimum-seeking approach. In such a method, the NN would compute the probable position of the follower in the leader's wake, and would bring the drone closer to the energetically ideal position. Then, a minimum-seeking algorithm could take over to fine-tune the position in order to reach a position of lower engine thrust.

6.1.4 Taking into account the leader drone's dynamics

One improvement clue could be to add inputs to the current neural network corresponding to the leader drone's movements. This would help the wake sensing algorithm to predict the deformation of the wake, and therefore keep the follower in the ideal position in the wake even when the leader performs maneuvers. Doing this might however require more non-linear layers in the neural network, since the problem could then become highly non-linear.

6.1.5 Detecting the wake deformation

Keeping in mind that some deformations of wakes are known and well-described in the scientific literature (Crow instability for example), it could be interesting to investigate methods of performing a wake sensing algorithm after having identified the type of deformation that the wake is currently undergoing. Multiple neural networks could have been trained on the different typical deformations that could be encountered, so that when the deformation type is identified, the corresponding neural network would be used.

6.1.6 Neural network dedicated hardware

Today, electronic hardware is being developed with the goal of being quicker to perform learning on a neural network. Different approaches exist: the physical neural network and the processors with AI-oriented instruction sets (explained briefly below). In real-life situations when the drone will have to perform learning, the learning process should be as quick as possible, because of obvious battery life concerns. Therefore, using dedicated hardware designed to make this process faster could prove to be an interesting improvement.

Physical neural networks This new approach relies on implementing the neural network entirely as a digital circuit, meaning that the neurons are implemented by digital electronic hardware. The computation of the output of the neural network is then almost instantaneous. The weights of the inputs of every neuron are either stored in a digital memory, or implemented through adjustable resistances.

Dedicated instruction set Another approach, implemented for example in Nvidia's new Pascal GPUs or in ARM's ML processor, is to take a "classic" processor, but to enhance its instruction set by adding hardware accelerators. These accelerators are in fact dedicated hardware to perform some instructions faster than on a conventional processor.

Bibliography

[1] Randal W. Beard and Timothy W. McLain. *Small Unmanned Aircraft, Theory and Practice*. Princeton University Press, 2012.

[2] D. Bieniek, R. Luckner, I. De Visscher, and G. Winckelmans. Simulation methods for aircraft encounters with deformed wake vortices. *Journal of Aircraft*, 53(6), 2016.

[3] KLM blog. A flight plan in 10 steps. https://blog.klm.com/flight-plan-in-10-steps/, July 2018.

[4] Marcin Brodecki and Kamesh Subbarao. Autonomous formation flight control system using in-flight sweet-spot estimation. *Journal of guidance, control and dynamics*, 38(6), June 2015.

[5] Philippe Chatelain and Grégoire Winckelmans. Aerodynamics of external flows (university lecture - lmeca2323). *Université Catholique de Louvain*, 2017-2018.

[6] L. J. Clancy. *Aerodynamics*. Pitman Publishing, 1975.

[7] Victor Colognesi. Simulation numérique de la stabilistaion d'un avion dans un sillage (unpublished master thesis). *Université Catholique de Louvain*, 2015-2016.

[8] Levi DeVries and Derek A. Paley. Wake sensing and estimation for control of autonomous aircraft in formation flight. *Journal of Guidance, Control, and Dynamics*, 39(1):32–41, 2016.

[9] Martin T. Hagan, Howard B. Demuth, Mark Hudson Beale, and Orlando De Jesús. *Neural Network Design, 2nd Edtion*. Martin Hagan, 2014.

[10] Maziar S. Hemati, Jeff D. Eldredge, and Jason L. Speyer. Wake sensing for aircraft formation flight. *AIAA Guidance, Navigation, and Control Conference*, August 2012.

[11] Frank Holzäpfel and Anton Stephan. Aircraft wake vortex evolution during approach and landing. *Innovatives Supercomputing in Deutschland*, 11(2), Autumn 2013.

[12] IATA. Iata forecasts passenger demand to double over 20 years. https://www.iata.org/pressroom/pr/Pages/2016-10-18-02.aspx, International Air Transport Association (IATA), 2016.

[13] Steven M. Lavalle. Planning algorithms. http://planning.cs.uiuc.edu/node102.html, Cambridge University Press, 2006.

[14] B. Llanas, S. Lantarón, and F. J. Sáinz. Constructive approximation of discontinuous functions by neural networks. *Neural Processing Letters*, 27(3), June 2008.

[15] Khan Tanzeem Mansoori, Amrit Suman, and Sadhana K. Mishra. Feature selection by genetic algorithm and svm classification for cancer detection. *International Journal of Advanced Research in Computer Science and Software Engineering*, 4(9), September 2014.

[16] Warren McCulloch and Walter Pitts. Aircraft wake vortex evolution during approach and landing. *Bulletin of Mathematical Biology*, 52(1/2), 1990.

[17] Corey Montella. The kalman filter and related algorithms, a literature review. *Research Gate*, May 2011.

[18] Nhan Nguyen, Sonia Lebofsky, Eric Ting, Upender Kaul, Daniel Chaparro, and James Urnes. Development of variable camber continuous trailing edge flap for performance adaptive aeroelastic wing. In *SAE Technical Paper*. SAE International, 09 2015.

[19] David Oxley. Forecasting air freight demand. https://www.iata.org/publications/economics/Reports/freigh-forecast/Forecasting-air-freight-demand.pdf, International Air Transport Association (IATA), March 2018.

[20] Lorenzo Pollini, Fabrizio Giuliettit, and Marco Innocenti. Sensorless formation flight. *AIAA Guidance, Navigation, and Control Conference and Exhibit*, August 2001.

[21] Andrzej Rusiecki, Mirosław Kordos, Tomasz Kamiński, and Krzysztof Greń. Training neural networks on noisy data. *International Conference on Artificial Intelligence and Soft Computing*, 2014.

[22] Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 11:743–746, 2010.

[23] K. Gnana Sheela and S.N. Deepa. Review on methods to fix number of hidden neurons in neural networks. *Mathematical Problems in Engineering*, 2013.

[24] Russel Stuart and Norvig Peter. *Artificial Intelligence a modern approach (3rd edition)*. Pearson, 2016.

[25] Mitchell Tom. *Machine learning*. McGraw-Hill International Editions, 1997.

[26] M. Jake Vachon, Ronald J. Ray, Kevin R. Walsh, and Kimberly Ennix. F/a-18 performance benefits measured during the autonomous formation flight project. *AIAA Atmospheric Flight Mechanics Conference*, September 2003.

[27] Pushpendra Kumar Yadav and Dr.N.L.Prajapati. An overview of genetic algorithm and modeling. *International Journal of Scientific and Research Publications*, 2(9), September 2012.

Appendix A

Transformation from follower frame to leader frame

The radial speeds generated by the wake ($u(\theta)$) are computed in the leader frame by computing the distances from the desired point to the centers of the two wakes. But this desired point is expressed in the follower frame, since it is on that follower drone that the influence of the wake needs to be computed. Therefore, a transformation matrix needs to be computed to be able to easily transform a point in the follower frame into the leader frame:

$$A * p_{foll} = p_{lead} \tag{A.1}$$

Where A is the said matrix, and p_{foll}, p_{lead} is the point expressed in, respectively, the follower and leader frames (as $(x, y, z)^T$ vectors).

In aviation, the orientation of an airplane with respect to the world frame is generally expressed with yaw (α), pitch (β) and roll (γ) angles that are defined as showed in figure A.1. To represent these three rotations, one single 3D rotation matrix can be used, computed by multiplying 3 rotation matrices each representing one of the yaw, pitch and roll angles [13]:

$$R(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}$$

$$R(\gamma) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{pmatrix}$$

The order of the multiplication of the matrices is very important as matrix multiplication is not a commutative operator. The defined order of multiplication is: $yaw * pitch * roll$, and this gives the following full rotation matrix:

$$R(\alpha, \beta, \gamma) = \begin{pmatrix} c(\alpha)c(\beta) & -s(\alpha)c(\gamma) + c(\alpha)s(\beta)s(\gamma) & s(\alpha)s(\gamma) + c(\alpha)s(\beta)c(\gamma) \\ s(\alpha)c(\beta) & c(\alpha)c(\gamma) + s(\alpha)s(\beta)s(\gamma) & -c(\alpha)s(\gamma) + s(\alpha)s(\beta)c(\gamma) \\ -s(\beta) & c(\beta)s(\gamma) & c(\beta)c(\gamma) \end{pmatrix}$$

Next to the rotations, there is also a translation that must be applied between the centers of the follower frame and the leader frame. This can be added in the transformation matrix by adding an extra row and line, which then becomes a 4x4 affine transformation matrix :

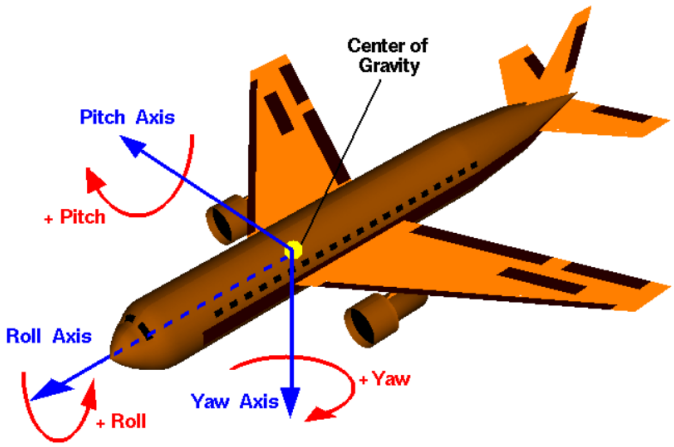


Figure A.1: Roll, pitch, yaw conventions (Source: NASA)

$$T = \begin{pmatrix} c(\alpha)c(\beta) & -s(\alpha)c(\gamma) + c(\alpha)s(\beta)s(\gamma) & s(\alpha)s(\gamma) + c(\alpha)s(\beta)c(\gamma) & dx \\ s(\alpha)c(\beta) & c(\alpha)c(\gamma) + s(\alpha)s(\beta)s(\gamma) & -c(\alpha)s(\gamma) + s(\alpha)s(\beta)c(\gamma) & dy \\ -s(\beta) & c(\beta)s(\gamma) & c(\beta)c(\gamma) & dz \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Where (dx, dy, dz) is the translation vector between the centers of the two frames. To use this type of full transformation matrix, one needs to add a 1 at the end of the vector representing the position of the point. Therefore, equation A.1 becomes:

$$T * \begin{pmatrix} p_{foll} \\ 1 \end{pmatrix} = \begin{pmatrix} p_{lead} \\ 1 \end{pmatrix}$$

Appendix B

More details on AI's main sub-concepts

B.1 Search

There are many different forms of tree searches, but the simplest and most common of them are *breadth-first search* (BFS) and *depth-first search* (DFS). The difference between them is that BFS searches the tree depth by depth. It goal-tests all nodes from a certain depth, and once this is done, expands the next depth and then searches this new depth. DFS on the other hand will search the tree by looking at the first successor of each node, going faster to the bottom of the tree. Once a node has no successors to be generated, the search goes up one level (it goes back to the parent node) and searches the other children of the parent node. Both algorithms terminate when they goal-test a node that is one of the goals of the problem. The two algorithm are illustrated in figure B.1. Choosing one search algorithm over the other is mainly a matter of speed and memory usage. But there can be some other considerations. For example, DFS has the advantage of requiring less memory and less computation time than BFS, but it does not necessarily give an optimal solution as BFS does. Moreover, if the tree is potentially infinite, the DFS algorithm might never terminate, where BFS will.

Another class of search algorithms is called *informed* search. To the contrary of *uninformed* search algorithms, they have more information about the problem than just its definition. They are indeed provided with some information that can direct the search to a solution. They use an *evaluation function*, that gives each node a cost corresponding to the cost of the actions required to reach this node from the current state of the problem. The node with the lowest cost is then the one to be expanded first. Another useful informed tool is the *heuristic function*, which is an estimation of the cost left to reach the goal from the currently analyzed node. To illustrate these tools, imagine for example a path-planning problem in which a robot must reach a known destination while avoiding known obstacles on a given map. The uninformed search algorithm will look in all directions, and will likely cover the whole map until it finds the goal and computed the path. On the other hand, the informed search will prefer looking in the direction that will bring the robot closer to the destination. Some examples of informed search algorithms are: greedy best-first search, A*search, memory-bounded heuristic search,...

Finally, there are many more search algorithms to be presented and that cover, for example, actions with unsure outcome, taking into account the potential movements of the adversary in multi-player games, searching in continuous spaces, ... But the world of search algorithms is vast and could not be described here without becoming a dissertation on its own.

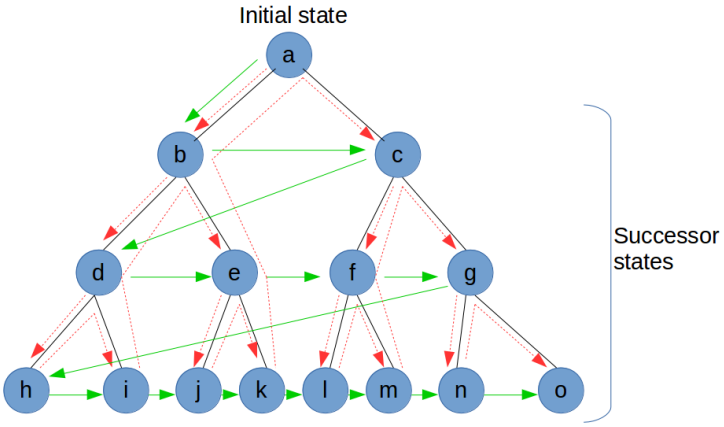


Figure B.1: BFS (plain green) and DFS (dashed red) example

B.2 Sudoku example for CSP

As a reminder, the principle of Sudoku is that the player is given a 9*9 grid divided in 9 3*3 sub-grids. Each of the cells of the grid must be given a value from 1 to 9. The constraints are that each of the values of each line and column must be different (so in each line/column, the player will find each of the 9 possible values). The values in each sub-grid must also be different. Let us define $X = \{X_1, X_2, ..., X_n\}$ as a set of variables, $D = \{D_1, D_2, ..., D_n\}$ a set of domains for each of these variables, and $C = \{C_1, C_2, ..., C_m\}$ a set of constraints. For the Sudoku game, these sets could be written as:

- **X:**$\{(1,1),(1,2),...,(9,9)\}$, the ensemble of the $9*9$ cells of the game's grid, represented by their (x, y) on the grid.
- **D:**$\{1,2,3,4,5,6,7,8,9\}$, the ensemble of all values that each cell can take.
- **C:**$\{Alldiff\,((1,:)),...,Alldiff((:,1)),...,Alldiff(subgrid_1),...\}$, the ensemble of all constraints defined by the rules of the Sudoku problem. The $Alldiff()$ constraint is a classical constraint that simply states that all the values as argument of the constraint must be different. The notation $(1,:)$ is used here to represent all cells with $x = 1$.

The game is won (*i.e.* a goal state is found) when all cells are assigned to a value and all these values satisfy all the constraints. Please note that if this example presents a discrete domain for the variables, CSP's can be used also with continuous domains (and even infinite domains). In that case, the constraints can be written mathematically, as for example $X_1 > X_2$.

B.3 Propositional Logic

The general principle of PL is to have a *knowledge base* (KB) which contains a set of *sentences*. These sentences describe some knowledge about the environment, and are written using a *knowledge representation language*. In classical logic, the knowledge is said to be *definite*, *i.e.* it is either true or false. At the beginning, the KB contains some general knowledge about the problem, and as the agent interacts with the environment, the agent will interact with the database to make it grow, and to consult it in order to know what to do next. A reasoning pattern unfolds as follows:

1. The agent `tells` the KB what is perceives.
2. The agent `asks` the KB what it should do next.

3. The agent `tells` the KB what action it performed, and it performs the action.

The fact of telling the KB about something is actually done by of constructing a sentence corresponding to what has to be told to the KB (sensor readings, action performed, ...) and by adding this sentence to the KB. Asking the KB on the other hand, is where the reasoning about the current state of the problem and the possible actions is done. Both of these steps might involve *inference* (the fact of deriving new knowledge from what is known right now, similarly to constraint propagation in CSP).

All of these concepts about logic are very abstract, but how is this being translated into something a computer can understand? This is where the knowledge representation language is proven useful. The principle of this language lies in the use of *proposition symbols* and *logical connectives*. The proposition symbols are simply statements about the environment that can be true or false. Imagine a robot that explores a building in search of a tag that emits light and noise, examples of these statements could be:

- L_{r1} :"There is light in room 1"
- H_{r2} :"Room 2 has a hole in the floor"
- N_{h3} :"there is a noise in hallway 3"

These proposition symbols can be sentences on their own, if the robot went into room 1 and detected light, it would `tell` the KB the sentence "L_{r1}". But these proposition symbols can also be assembled with *logical connectives* (for example, $\neg(not), \vee(or), \wedge(and), \Rightarrow (implies), ...$). Imagine in the previous example that if the robot detects a hole in a room, it means that the tag is not in that room. If a robot visits room 2 in which he detects a hole, he `tells` the KB the sentence $\mathbf{S_1} : \{H_{r2}\}$. The initial knowledge of the database contains the sentence $\mathbf{S_2} : \{H_{r2} \Rightarrow \neg T_{r2}\}$ (where T_{r2} designates the symbol "the tag is in room 2") because of the initial knowledge of the problem, namely the rule that if there is a hole, the tag is not there. So when the robot tells the KB about the hole in room 2, the KB can infer a new sentence $\mathbf{S_3} : \{\neg T_{r2}\}$

First-Order Logic (FOL) extends the idea of PL by adding the opportunity of designating multiple objects that share a property all at once. For the "hole in the room" example, instead of writing in the initial KB a sentence saying "if there is a hole there is no tag" for every possible room, FOL permits to use the following format:

$$\forall x : Room(x) \wedge Hole(x) \Rightarrow \neg Tag(x)$$

This sentence can be read as "for every x, if x is a room and there is a hole in x, then there is no tag in x".

Appendix C

Deriving the backpropagation rule

This section is directly taken from [25]. First, let us recall or define some notations:

- x_{ij}: i^{th} input of neurons j
- w_{ij}: weight of i^{th} in put of neuron j
- $net_j = \sum_i w_{ij}x_{ij}$ (the sum at the input of the neurons (see section 4.2.1))
- o_j: output computed by unit j
- t_j: target output for unit j (from the training example)
- σ: sigmoid function
- $outputs$: the set of neurons in the final layer of the NN
- $Downstream(j)$: set of neurons directly downstream of neuron j (one of their inputs is the output of j)

As shown in section 4.4, the weights of the neural wetwork will be adapted with the rule:

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E_d}{\partial w_{ij}}$$

With:

$$E_d = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

To compute the term $\frac{\partial E_d}{\partial w_{ij}}$, we see that the only term in E_d that is influenced by the weights w_{ij}, is the output o_k. These outputs are given by multiplying net_j by the function of the neuron (for example, the sigmoid function for the sigmoid neurons). Therefore, the weights only influence the outputs via net_j. By using the chain rule, we obtain:

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial net_j} x_{ij}$$

And this term $\frac{\partial E_d}{\partial net_j}$ depends of whether the neuron on which the calculation is made is an output neuron or a hidden neuron.

C.1 Output neurons

Since output neurons are at the output of the neural network, their weights influence directly the output of the network. And the only way for the net_j to influence E_d is through the output of the neuron. Therefore, using the chain rule again, we have:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

From the definition of the training error E_d, the first term gives:

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{2}(t_j - o_i)^2 = -(t_j - o_j)$$

For the second term, if we take the example of sigmoid neurons, we have:

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j} = o_j(1 - o_j)$$

and we then finally obtain :

$$\delta_j = (t_j - o_j)o_j(1 - o_j)$$

C.2 Hidden neurons

Hidden neurons do not have direct influence on the output of the net. Is is thus necessary to take into account the influence of the hidden neuron on the neurons situated downstream to it. If we consider that the neuron being analyzed is on the last hidden layer before the output layer, we have:

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \left(\frac{\partial E_d}{\partial net_k}\frac{\partial net_k}{\partial net_j} \right) \\
&= \sum_{k \in Downstream(j)} \left(-\delta_k \frac{\partial net_k}{\partial net_j} \right) \\
&= \sum_{k \in Downstream(j)} \left(-\delta_k \frac{\partial net_k}{\partial o_j}\frac{\partial o_j}{\partial net_j} \right) \\
&= \sum_{k \in Downstream(j)} \left(-\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \right)
\end{aligned}$$

And for sigmoid hidden neurons, we then have:

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Appendix D

More details on the developed code

This annex will give more details about the general organization of all the software used in the context of this work. It also serves as information about the location of the modifications done in SITL to enhance the simulator with the wake physics, and other modifications that were not directly related to the physics computations. Finally, it provides more insight about the organization of the code of the simplified problem.

D.1 Simulation environment

D.1.1 SITL

Implementation of the leader drone

When starting an SITL instance, the vehicle's location is available through 2 different position variables:

- A NED (north-east-down) local frame with the center of the frame being the "home" of the simulation. That is, the starting point of the vehicle.
- A global frame using the WGS84 coordinate system (longitude, latitude, altitude over mean sea level).

Since SITL can not currently simulate multiple vehicles, it is necessary to implement the leader drone in the code of the simulator. To do this, we take advantage of one of the hypothesis made on the wake in chapter 2: we neglected the dissipation of the wake behind the leader. By this hypothesis, the distance (along the x axis of the leader's frame) between the two drones has no impact on the wake physics. Therefore, in the code we make one extra simplification: we suppose that the wake is infinite both behind *and ahead* the leader drone. With this simplification, the leader's position was hard-coded in the code of SITL to be at an altitude of 100m above the "home" position. Because the wake is considered infinite in both directions, it does not matter if the follower is in front or behind the leader, as long as it is close enough in the (y, z) plane, it will feel the effects of the wake. With a parameter, the user can choose whether the leader is oriented towards the north, the east, the south or the west.

Organization of the physics code

SITL's vehicle simulation is organized in three C++ class "layers". These layers are illustrated in figure D.1. In orange is the vehicle type we are interested bu (ArduPlane; the fixed-wing drone). All files named in this section are located under the file path `/ardupilot/libraries/SITL/`.

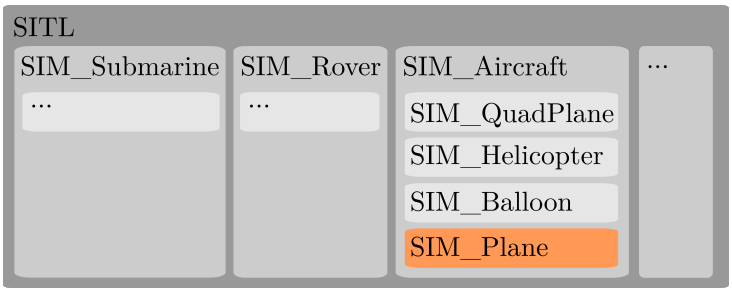


Figure D.1: Organization of the SITL C++ code to simulate vehicles. SIM_Plane corresponds to the fixed-wing drone type of vehicle used in this work.

The topmost layer is included in the `SITL.cpp` file. This implements generalities that all types of vehicles (airborne or not) will require. More specifically, this is where sensors are initialized, and where some general functions are implemented (for example, frame conversion functions). In this code, one can also find the functions used for the MAVLink communication (see section D.1.4) between SITL and other bits of code (ArduPilot, Dronekit, MAVLink console, ...).

For airborne vehicles, the second layer is found in the `SIM_Aircraft.cpp` file. This is where general functions are implemented for usage by all types of airborne vehicles (multi-rotor copters, helicopters, fixed-wing drones, ...). Concretely, this file contains more vehicle-specific functions relating to frame and altitude conversions. It also contains functions related to the simulation itself (simulation frame step time, ...) and some general physics-related functions. The latter functions implement wind and sensors, but also dynamics. These vehicle dynamic functions are used to update the attitude and position of the aircraft based on linear and angular accelerations computed by the third class layer.

The third class layer of SITL's vehicle instances is specific for each type of airborne vehicle. The file related to fixed-wing drones (the ArduPlane vehicle) is called `SIM_Plane.cpp`. This is where the physics of a flying fixed-wing aircraft is implemented. All explanations about the wake-less code in chapter 2 actually describe the internal details of this file. It is thus here also that the simulator was modified to add the wake physics described in the same chapter. Concretely, this file contains one primary function, used to update the instance of the vehicle. This function calls the `update_dynamics` and `update_position` functions from the second "layer" of code after having computed the forces applied on the vehicle. Of course, these forces are computed using many functions that implement the physics of fixed-wing flight (lift, drag, control surfaces, ...).

D.1.2 ArduPilot's autopilot

ArduPilot's autopilot is usable in different modes. Hereunder, the most important in the context of this work are listed here (there are 13 modes for ArduPlane vehicles):

- **MANUAL:** The drone's autopilot is overridden, and the control surfaces are directly controlled via the MAVLink messaging system.
- **AUTO:** The drone follows the current mission. This means that it flies in full autopilot mode to reach the next rally point in line.
- **GUIDED:** Similar to AUTO: in this mode, the drone flies to a single user-defined rally point. This is used when there is no complete mission to perform.

No modifications of the autopilot were done in the context of this work. It was controlled in AUTO mode directly from the mission planner by sending rally points to the drone. It is with

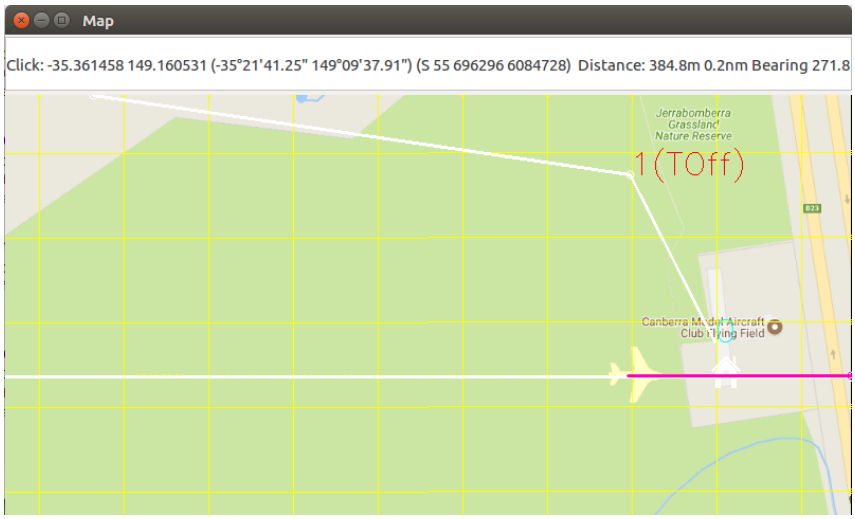


Figure D.2: Map view of a drone simulated in SITL in AUTO mode. It follows the white line that links the different mission rally points.

these rally points that the follower was placed in the leader's frame for some tests of the wake physics. An illustration of a map provided by ArduPilot's environment is given in figure D.2.

The ArduPilot environment also provides the user with a MAVLink console to send messages to the autopilot.

D.1.3 Dronekit

In the future works, the goal of using Dronekit would be to perform the NN training in Python independently of the code of the running simulation. Since the planed approach is to use supervised learning, the NN script should train on a training set with inputs and outputs. Since in a real-life situation, the neural network will not be able to measure the physical effect of the wake, the inputs of the neural network linked with SITL will be the deflections of the control surfaces. Because ArduPilot will be counter-acting the effects of the wake to keep the drone in a steady flight, these deflections will be directly related to the inputs used in the simplified problem of this work.

The output of the Dronekit network will be the (y, z) position of the follower in the wake. So for the training, Dronekit must be able to access this precise location. To generate that location, one needs to transform the position of the follower in the world frame to the position of the follower in the leader frame. This is where setting the leader's position at $(0, 0, 0)$ (the "home" position) is interesting, because it allows for easier computation without loss of generality. However, one problem with the ArduPilot environment is that the position in ArduPilot itself is issued from the simulated sensors, and does not come directly from the SITL simulator. The main problem is then that the altitude in ArduPilot presents a large error. Indeed, at an altitude of 100m (the altitude of the leader), the error is of almost 10m. Therefore, when computing the position of the follower in the leader's frame, the error will be present in this new frame. This prevents the NN to learn from the real position in the wake, and it is thus necessary to allow Dronekit to access the real simulator position.

To do this, it was necessary to add a custom MAVLink message (see section D.1.4) containing the true SITL position of the simulated vehicle (that is, the follower). This message is send automatically at each simulation step. Dronekit is then able to create a "listener" to that

message so that it updates the position of the follower. The position and heading of the leader is then hard-coded into Dronekit as it is done in SITL, so that the frame transformation can be performed, computing the follower's position in the leader's frame.

D.1.4 MAVLink communication

MAVLink is the communication protocol implemented in the ArduPilot environment. It is a standard protocol developed to communicate with a drone, and to allow on-board drone components to communicate together. In the ArduPilot environment, all communications between the running codes is done using this protocol. This allows Dronekit, the MAVLink console and the mission command center to send commands or rally point coordinates to ArduPilot. But it can also be used, when a SITL simulation is running, to obtain information about the state of the simulator (for example: the "home" location expressed in the world frame, the time of the simulation, ...).

D.2 Neural network & simplified problem

In this section, the code used to generate the results from chapter 5 is presented. Since the simplified wake sensing problem is separated from the simulation environment, all the physical effects of the wake had to be re-programmed in this new environment. The main difference being that, in this simplified environment, the leader and follower are considered as having the exact same attitude (roll, pitch, yaw), and the dynamics of both vehicles are neglected. Concretely, the code of the simplified problem was done in Python, and organized in three separate files:

Wake_generator.py This first Python script is used to generate the training set for the neural network. This means that the script generates a `.txt` file containing the inputs and outputs of the training examples for the NN. The inputs are, as explained in chapter 4, the three physical effects of the wake, and the outputs are the (x,y) position of the follower in the wake. The physical effects are generated exactly as described in section chapter 2, with the simplifications given above. The user of the script defines the parameters of the training set generation. These are: the size of the zone of the wake that will be generated, and the discretization of the generated zone. Then, when running, the script iterates from one end of the desired zone to the other, with the desired discretization parameter. For each point of the discretization, it saves an (input, output) couple in the `.txt` file.

plot_dataset.py This second Python script, as the name implies, is used to plot the training data sets generated by the previous script. It simply reads the generated `.txt` file and produces heatmaps of the three physical effects (seen in chapter 2).

ANN_pybrain.py This final script is where the PyBrain neural network is generated and trained. This script reads one of the dataset files generated with the *wake generator*, and creates a PyBrain dataset from it. Then, a PyBrain neural network is instantiated, and trained on the dataset. Each 10 epochs, an error heatmap as the ones seen in chapter 5 is generated and saved. This script also generates a `.txt` file containing the statistical values of the error (mean, standard deviation and variation) corresponding to each heatmap plot. There are a number of parameters on which the user can play:

enable_camera To add the quadrant sensor.

bias_camera To simulate a wake shift (two other parameters correspond the desired y and z shifts).

tanh To use a hidden tanh layer.

linear To use a hidden linear layer.

sigmoid To use a sigmoid output layer.

norm_out To use a normalized output dataset.

norm_in To use a normalized input dataset.

N_training Number of epochs.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**