UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

REPORT

Airfoil Optimisation with Genetic Algorithm

Master Thesis

Master's Degree in Aerospace Enginering

ESEIAAT - Universitat Poilitècnica de Catalunya

Student

Pere Camps Castellanos

Director

Manel Soria

Terrassa, May 23, 2021

Acknowledgments

I wish to express my sincere gratitude to Lluís Alsedà and Albert Ruiz, professors of the Universitat Autònoma de Barcelona, who taught me all the things I know about genetic algorithms, without whom this work would have not been possible. And also to Manel Soria, my thesis Director, to gave me the freedom of work on the topic I wanted.

I am also extremely thankful to UPC Venturi for allowing me to share some results of their work with the V17, the best aircraft the team has ever designed.

I express my deepest thanks to my colleagues from UPC, for sharing the long working hours during the COVID-19 pandemic. Whose suggestions and constructive criticism have helped me to develop the current thesis.

Finally, I would like to hugely thank my family and friends for the support given. Thank you for everything.

Abstract

In this Master Thesis, an airfoil optimisation using a Genetic Algorithm is developed. This project has been proposed by myself and done with the guidance and supervision of professor Manel Soria.

The main idea of the project is to develop from scratch an algorithm capable of finding the optimal airfoil for specific flow conditions, such as the angle of attack, the Reynolds number, and the Mach number. The objective is to create a useful tool for aerospace engineering students so they can use it on their projects and designs during the college years.

The work has a first theoretical part about Genetic Algorithms, in which the basic concepts needed to understand the current project are explained. Then, the implementation of the algorithm is fully expounded and all the intern processes of the genetic algorithm can be consulted. Several validations of the code have also been made.

The Genetic Algorithm created uses crossovers and mutations. The airfoil parametrisation used has been the PARSEC parametrisation and the computation of the aerodynamic coefficients is done with XFOIL. The whole code is written in C language and the results analysis and graphs are done with MATLAB and XFLR5.

Finally, the algorithm is tested with two real design cases, an airfoil for a heavy lifter aircraft that participated in the Air Cargo Challenge 2017 in Stuttgart, and an airfoil for a glider that flew in the Paper Air Challenge 2015 in ESEIAAT, Terrasa. The results and improvements offered by the algorithm are compared with the results that the designers of these aircraft obtained manually during the design process.

Contents

List of Figures

List of Tables

Glossary

Symbols

C_l	Lift Coefficient.	[-]
Cd	Drag coefficient.	[-]
α	Angle of attack.	[º]
f_a	Aerodynamic force.	$[N]$
$m_{c/a}$	Aerodynamic moment	$[N \cdot m]$
L	Lift. Projection of the aerodynamic force to the vertical axis.	$[N]$
D	Pressure Drag. Projection of the aerodynamic force to the horizontal axis.	$[N]$
c	Chord. Length of a given airfoil.	$[m]$
u_∞	Upstream air speed.	$[m/s]$
P_∞	Upstream air pressure.	$[Pa]$
P	Local air pressure.	$[Pa]$
ρ_∞	Upstream air density.	$[Kg \cdot m^3]$
μ_∞	Upstream air viscosity.	$[kg/(m \cdot s)]$
a_∞	Upstream sound speed.	$[m/s]$
Re	Reynolds number.	[-]
Ma	Mach number.	[-]
γ	Heat capacity ratio.	[-]
R	Specific gas constant for dry air.	$[J/(Kg \cdot K)]$
T	Temperature.	$[K]$
E_{ff}	Airfoil Efficiency.	[-]
r_{LE}	Radius of an airfoil leading edge.	$[m]$
r_{LElo}	Radius of an airfoil leading edge in the lower surface.	$[m]$
X_{up}	x coordinate of maximum position of the upper surface.	$[m]$
Z_{up}	z coordinate of maximum position of the upper surface.	$[m]$
X_{lo}	x coordinate of minimum position of the lower surface.	$[m]$
Z_{lo}	z coordinate of minimum position of the lower surface.	$[m]$
Z_{xxup}	Curvature of the position of maximum thickness of the upper surface.	$[m]$
Z_{xxlo}	Curvature of the position of maximum thickness of the lower surface.	$[m]$
α_{TE}	Angle of inclination of the mean chamber line in the trailing edge.	[º]
β_{TE}	Angle of separation of upper and lower surface in the trailing edge.	[º]
Z_{TE}	z coordinate of the trailing edge.	$[m]$
ΔZ_{TE}	Separation between the upper and lower surface in the trailing edge.	$[m]$
a_i	Intermediate parameters of the PARSEC parametrisation.	[-]
b_i	Intermediate parameters of the PARSEC parametrisation.	[-]
W	Weight.	[Kg]
γ_d	Angle of descent.	[º]

Acronyms

GA	Genetic Algorithm.
CFD	Computational Fluid Dynamics.
NACA	National Advisory Committee for Aeronautics.
CST	Class Function Transformation.
SVD	Singlular Value Decomposition.
RBF	Radial Basis Function.
MinGW	Minimalist GNU for Windows.
GNU	Operative system. Recursive acronym of "*GNU's Not Unix!*".
ACC	Air Cargo Challenge.
PAC	Paper Air Challenge.
APAE	Asociação Portuguesa de Aeronáutica e Espaçao
EUROAVIA	European Assocication of Aerospace Students.
OEW	Operational Empty Weight.
MTOW	Maximum Take Off Weight.
GUI	Graphical User interface.
ESEIAAT	Escola Superior d'Enginyeria Industrial, Aeronàutica i Audiovisual.

Software

XFOIL	Is an interactive program for the design and analysis of subsonic isolated airfoils. It consists of a collection of menu-driven routines which perform various useful functions such as: viscous (or inviscid) analysis of an existing airfoil, airfoil design and redesign by interactive modification of surface speed distributions, airfoil redesign by interactive modification of geometric parameters, Blending of airfoils and plotting of geometry, pressure distributions, and multiple polars. Written in FORTRAN. [4]
*XFLR*5	XFLR5 is an analysis tool for airfoils, wings and planes operating at low Reynolds Numbers. It includes: XFoil's Direct and Inverse analysis capabilities and also wing design and analysis capabilities based on the Lifting Line Theory, on the Vortex Lattice Method, and on a 3D Panel Method. User Friendly version of XFOIL. Uses the same 2D solver written in C++. [5]
MATLAB	MAtrix LABoratory. Is a multi-paradigm numerical computing environment. A proprietary programming language developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python. [6]

Chapter 1

Introduction

This work aims to develop a code to optimise an airfoil shape given specific flow conditions with a Genetic Algorithm. The main idea is to generate an easy-to-use code capable to define an optimal airfoil that excels in the task it is designed for.

The main inputs of the code are, on the one hand, the flow conditions: the angle of attack, the Reynolds number and the Mach number. On the other hand, the user can modify some aspects of the computation in order to save time or on the contrary, obtaining more precise results sacrificing time computation. The user can also select the optimisation parameter, depending on what finds more useful for the aircraft design.

The work is focused on the development of the Genetic Algorithm and the precise selection of its parameters. To achieve the goal the following steps are required:

- Definition of the optimisation problem including objective functions, constraints and optimisation variables.
- Selection of the airfoil parametrisation.
- Development of the genetic algorithm
- Computation of the fitness function. In this work, the aerodynamic calculus are done externally via XFOIL.
- Code Validation
- Code testing and obtention of results for specific applications

The work is divided into three parts. First, there is a brief explanation of the concepts needed to understand the development of this specific genetic algorithm. This part is not intended to be a complete guide of theoretical aspects of genetic algorithm. Then, the implementation of the algorithm in C language using the concepts previously explained can be seen, including several validations. Finally, the algorithm is tested with two real design cases, an airfoil for a heavy lifter aircraft that participated in the Air Cargo Challenge 2017 in Stuttgart, and an airfoil for a glider that flew in the Paper Air Challenge 2015 in ES-EIAAT, Terrasa. The results and improvements offered by the algorithm are compared with the results that the designers of these aircraft obtained manually in the design process.

Chapter 2

Identification of the problem

When designing a plane for academic purposes or for student competitions, the step of analysing and deciding the airfoils used for the aircraft is complex and long, because the tools and knowledge that students have are limited. The typical steps are to do a little research about what kind of airfoils are used for the purpose of the aircraft. For instance, it is common to search for glider airfoils, high lift airfoils for cargo plane or reflex airfoils for flying wings. Then, after the list of possible airfoils is done, several analyses are performed to check or identify which is the airfoil that most adapts to the desired behaviour. This process, apart of being long, because the analyses are done one by one, usually ends up with a compromise solution because no airfoil fits exactly in the expected behaviour.

All in all, it usually takes several hours of work to find a couple of airfoils that can actually be used for the plane design. The main idea of this project is to automatise this process by creating a tool capable of find airfoils that are optimum for the desired flow conditions, with the aim of saving time for the designer and make the design process easier. With this objective, all this work is uploaded in the GitHub repository `https://github.com/perecamps/Airfoil_Optimisation_GA`, so it is accessible to anybody interested in it.

It must be said that the process of designing a plane is iterative and usually it is needed to do some steps backward and repeat some analysis. For instance, once the airfoil is chosen, it can happen that it does not behave well with the 3D geometry of the aircraft and the 2D analysis of airfoils must be repeated. Regarding this aspect, this work aims to reduce the time spent in these 2D analyses that must be done anyway.

Usually, students use XFOIL or XFLR5 software to compute aerodynamic coefficients for his airfoils. These software use methods based on the lifting line theory, the vortex lattice and 3D panels, which offer fair enough results for the purpose needed. Compared with CDF software, they are simpler to use and the results are obtained faster [4]. In this work, XFOIL is used to compute the aerodynamic coefficients for the so said reasons: it is easier to use, the results are obtained faster and they are good enough for the studied cases. However, use XFOIL in a C routine is not as easy and unfortunately, it takes some time to call XFOIL from a C execution during several iterations. These issues are treated in section 4.6.

This work does not aim to be a complete guide of the Genetic Algorithm or aerodynamics, it is assumed that the reader is familiarised with the topics of this project. However, a brief introduction to airfoil geometry and aerodynamic coefficients is done.

Figure 2.1 shows the most important aspects of an airfoil. The leading edge (*Borde de ataque*) is the front part of the airfoil and is the division point between the upper side (*Extradós*) and the lower side (*Intradós*). The radius of the leading edge has huge effects on the pressure distribution of the airfoil. The trailing edge (*Borde de Salida*) is the last part of the airfoil and its shape should generate a gentle separation of the streamline. The chord (*Cuerda*) is the straight line between the leading and the trailing edge. The curvature

(*Curvatura*) is the separation between the chord and the mean line between the upper and the lower side (*Línea de Curvatura*). Airfoils with zero curvature are symmetric airfoils and do not generate lift with α=0º. On the other hand, airfoils with more curvature tend to generate more lift, but also more drag and momentum. The thickness (*Espesor*) is the distance between the upper and the lower side.

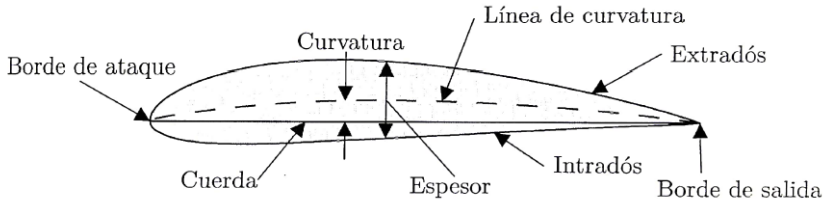


Figure 2.1: Airfoil Geometry. Image from [1].

Figure 2.2 shows the aerodynamic force f_a that an airfoil generates, and how it can be decomposed into the horizontal and the vertical axis as the lift and the pressure drag. Regarding the $m_{c/4}$, in every airfoil exist a point that if it is taken as momentum center, the momentum is approximately constant for every angle of attack. This point is called the aerodynamic center and in most of the subsonic airfoils, it is located near $c/4$ [1].

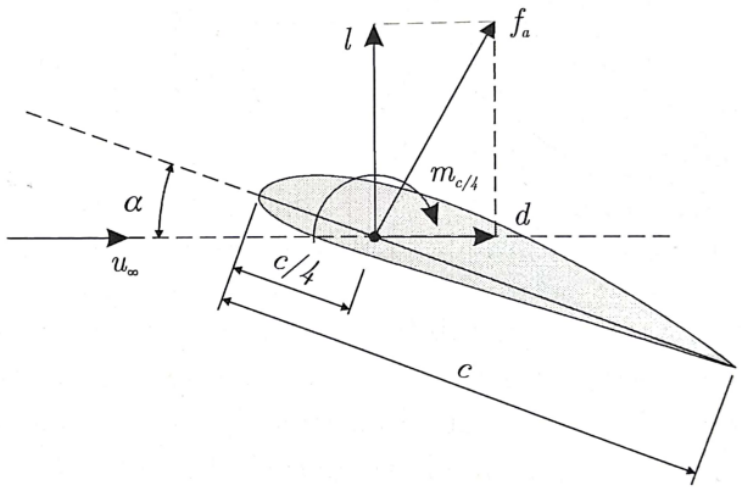


Figure 2.2: Forces on an airfoil. Image from [1].

The explanation of how the force f_a is created is more understandable looking at figure 2.3. Figure 2.3 shows the pressure distribution and the streamlines around a symmetric airfoil. When the angle of attack is 0 (left image), since the airfoil is symmetric the pressure distribution around it is also symmetric. In the right image, the airfoil is rotated 5º and as a consequence, the pressure distribution is not uniform anymore. On the upper side of the airfoil, the pressure is lower and on the lower side the pressure is higher. This pressure difference generates a suction, an aerodynamic force that can be decomposed as lift and drag. Airfoils are shapes designed to achieve a specific pressure distribution around them that generates the desired lift and drag. It must be said that, apart from the pressure drag, there is the friction or viscous drag, which is generated due to the friction of the air with the airfoil surface.

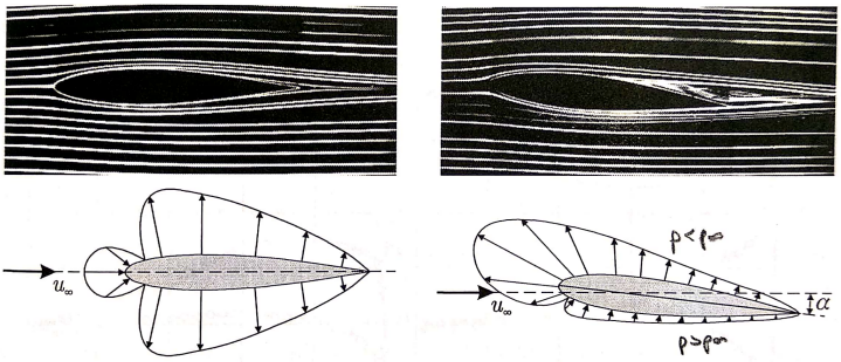


Figure 2.3: Flow lines and pressure distribution. Image from [1].

In order to be able to predict the loads around a given airfoil, the non-dimensional numbers are very helpful. In general, it could be said that the lift, drag and m_{ca} depend on 6 variables: ρ_∞ ($[Kg \cdot m^3]$ air density), μ_∞ ($[N \cdot s \cdot m^{-2}]$ air viscosity) , a_∞ ($[m/s]$ sound speed) , c ($[n]$ airfoil length) , u_∞ ($[Km/s]$ airspeed) and α. With non-dimensional analysis, it is possible to reduce this amount of variables if the following coefficients are defined.

$$c_l = \frac{l}{\frac{1}{2}\rho_\infty u_\infty^2 c} = f_l(\alpha, Re, M_\infty) \tag{2.1}$$

$$c_d = \frac{d}{\frac{1}{2}\rho_\infty u_\infty^2 c} = f_d(\alpha, Re, M_\infty) \tag{2.2}$$

$$c_{m,ca} = \frac{m_{ca}}{\frac{1}{2}\rho_\infty u_\infty^2 c^2} = f_m(\alpha, Re, M_\infty) \tag{2.3}$$

As a recall, the Reynolds number is a non-dimensional number that indicates the relation between the convective forces and the viscous forces. The relation of these forces is what determines if a flow is laminar or turbulent. Low Reynolds number imply laminar flows and high Reynolds number imply turbulent flows [7]. The Reynolds number is defined as:

$$Re = \frac{convective\ forces}{viscous\ forces} = \frac{\rho u c}{\mu} \tag{2.4}$$

The Mach number is also a non-dimensional number that indicates the relation between the fluid velocity and the sound speed of this same fluid. The sound speed depends on the heat capacity ratio (non-dimensional), the normalised gas constant $[J/(Kg \cdot K)]$ and the temperature $[K]$.

$$a = \sqrt{\gamma R T} \tag{2.5}$$

$$M_a = \frac{fluid\ velocity}{sound\ speed} = \frac{u}{a} \tag{2.6}$$

As shown, the variables are reduced from 6 dimensional variables to 3 non-dimensional variables. With these reductions, the tests in wind tunnels are significantly easier to perform and obtain data for these coefficients in function of the angle of attack, Reynolds number and Mach number. Once this data is computed experimentally or with specialised software nowadays, the aerodynamic forces in an airfoil are easily computed as:

$$L = \frac{1}{2}\rho_\infty u_\infty^2 c_l c \tag{2.7}$$

$$D = \frac{1}{2}\rho_\infty u_\infty^2 c_d c \tag{2.8}$$

$$M_{ca} = \frac{1}{2}\rho_\infty u_\infty^2 c_m c^2 \tag{2.9}$$

In this work, these coefficients are computed with XFOIL, as it has been said in the previous chapter. This software only needs the airfoil geometry, the Mach and Reynolds number and the angle of attack to generate the three coefficients. The coefficients are very precise when compared to experimental results especially with low Reynolds number, Mach number and angle of attack.

All in all, with the developed code, it is expected to find airfoil shapes that maximises these aerodynamic coefficients or some relations between them. For instance, the Efficiency (Eff) is the relation between the Cl and the Cd and is a very useful parameter to take into account when designing a plane or an airfoil.

$$E_{ff} = \frac{C_l}{C_d} \tag{2.10}$$

Chapter 3

Genetic Algorithm

3.1 Theoretical approach to Optimisation Methods

The main goal is to perform an airfoil optimisation. Broadly, two types of optimisation methods can be defined [8]:

- Deterministic Method: It solves the optimisation problem working in the gradient of the function to be optimsed, the objective function. The deterministic method is the classical method, in which the minimum or maximum is found using mathematical tools based on the analysis of the function. It is widely used in linear problems.
- Stochastic Method: It solves the optimisation problem including some random aspects in the search. Usually, these random elements are inspired by natural processes. Some of the most common stochastic methods are:
  - Evolutionary algorithms. Inspired by Darwin's theory of natural selection. The Genetic algorithm is often considered a subcategory of evolutionary algorithms. The main difference is that Evolutionary Algorithms rely mainly on mutation to achieve the evolution of the individuals, while Genetic Algorithms use crossovers and mutations [9].
  - Game Theory-Based algorithms: Based on the Nash game theory [10].
  - Particle Swarm algorithms: Inspired by the behaviour of flocking birds.

3.2 The Genetic Algorithm

In general, the Genetic Algorithm uses the three aspects of Darwin's theory: Selection, Reproduction and Mutation. The selection, as a concept of the survival of the individuals who are more adapted to their world. Thus, the characteristics of these individuals, with the best adaptation, is transferred from one generation to the other one. The reproduction, with the objective of generating new individuals with the best characteristics of its predecessors. Finally, the mutation, to explore other solutions that may not be contemplated without it and, eventually, generate a better individual than the rest [11].

In a biological system, an individual is composed of various chromosomes, and each of the chromosomes is composed of several genes. In the genetic algorithm code, usually an individual is composed of only one chromosome, and this chromosome can be formed by various genes. In this algorithm, the most important task is to identify which parameters will be the genes of the individual, in order to perform the encoding from a physical or tangible concept, to the series of numbers that are the genes [9].

In the case of study, in which the objective is to optimise an airfoil, it is pretty clear that the individual is the airfoil. However, it is still needed to find how to represent this individual as a series of numbers (genes). The solution is the airfoil parametrisation.

The shape parametrisation is a way to represent the airfoil shape via certain parameters. The objective is to be able to define a large range of airfoils with the lowest number of parameters. Two shape parametrisation methods can be found, the constructive and the deformative. The constructive methods generate the full airfoil shape with its variables, while the deformative methods modify the shape of an actual airfoil [12].

Among the constructive methods, the following can highlight:

- NACA [13].
- Class Function transformation (CST) [12].
- SVD Method [12].
- B-Spinlines [12, 14].
- PARSEC [12, 14, 15].
- Joukouwsky transformation [14].

While, among the deformatives methods, the following are the most important:

- Hicks-Henne Bump Functions [12, 14, 16].
- Radial Basis Function Domain Element (RBF) [12].
- Bèzier Surfaces [12].

All these methods have its pros and cons. In the cited references they are compared, considering the number of variables needed to define the shape and its effectiveness in optimisation methods.

Usually, the Evolutionary Algorithms applied to airfoils use deformative methods, in order to improve the performance of a given airfoil, while Genetic Algorithms use constructive methods, to find the optimal solution from scratch.

Once the parametrisation has been chosen, the main idea is to generate an initial population of N individuals and then evaluate their fitness in the system. The fitness function is a complex issue that is further explained in section 3.2.5 and implemented in section 4.6. For now, let's say that fitness is a value that expresses how good is the individual for surviving in its world, the lower is the fitness, the higher are its probabilities to survive and be chosen for reproduction. Once the fitness of all individuals are computed, a random process selects two individuals for reproduction. As said before, individuals with lower fitness have more chances to be chosen. These two individuals generate two descendants via crossover and mutation, explained in section 3.2.3 and 3.2.4. The reproduction continues until a new population of N individuals is achieved. Then, the process is repeated and the fitness of each individual is computed, the best ones are selected for reproduction, etc. The process stops when certain conditions are met, for example, the process reaches the maximum number of generations allowed, or the fitness of the best individual does not improve for a given number of generations.

3.2.1 PARSEC parametrisation

For this work, the PARSEC parametrisation has been selected, mainly because its very intuitive. Since each of the parameters have a physical meaning, it can generate several airfoil shapes and it has been widely used in airfoil optimisation problems [10, 17–20]. This method was presented by Sobiecsky [15], and it uses 11 parameters in its original form to represent the airfoil, although modified versions use 12.

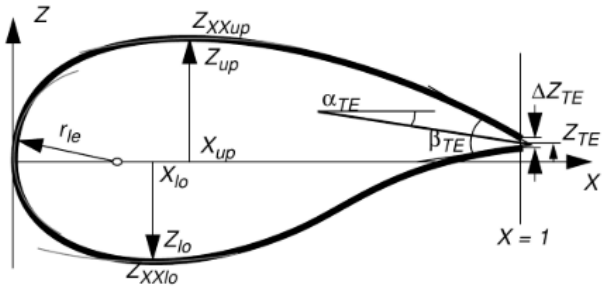


Figure 3.1: Representation of PARSEC parameters.

Figure 3.1 shows the 11 original parameters of the PARSEC representation, which are the following:

- r_{LE}: radius of the leading edge
- X_{up}: x coordinate of the maximum position of the upper surface
- Z_{up}: z coordinate of the maximum position of the upper surface
- X_{lo}: x coordinate of the minimum position of the lower surface
- Z_{lo}: z coordinate of the minimum position of the lower surface
- Z_{xxup}: curvature in the position of maximum thickness of the upper surface
- Z_{xxlo}: curvature in the position of maximum thickness of the lower surface
- α_{TE}: Angle of the inclination of the mean chamber line in the tailing edge
- β_{TE}: Angle of separation of the upper and lower surface
- Z_{TE}: z coordinate of the trailing edge
- ΔZ_{TE}: Separation between the upper and lower surface in the trailing edge

A modified version, used in this work, includes also the radius of the leading edge in the lower surface, r_{LElo}.

With these 12 parameters, it is needed to solve two systems of equations in order to be able to represent the upper and lower side of the airfoil. Equation (3.1) shows the system of the upper side, while equation (3.2) shows the one of the lower side. They are very similar and with them, 12 new values are obtained: a_i and b_i.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ X_{up}^{1/2} & X_{up}^{3/2} & X_{up}^{5/2} & X_{up}^{7/2} & X_{up}^{9/2} & X_{up}^{11/2} \\ 1/2 & 3/2 & 5/2 & 7/2 & 9/2 & 11/2 \\ \frac{1}{2}X_{up}^{-1/2} & \frac{3}{2}X_{up}^{1/2} & \frac{5}{2}X_{up}^{3/2} & \frac{7}{2}X_{up}^{5/2} & \frac{9}{2}X_{up}^{7/2} & \frac{11}{2}X_{up}^{9/2} \\ -\frac{1}{4}X_{up}^{-3/2} & \frac{3}{4}X_{up}^{-1/2} & \frac{15}{4}X_{up}^{1/2} & \frac{35}{4}X_{up}^{3/2} & \frac{63}{4}X_{up}^{5/2} & \frac{99}{4}X_{up}^{7/2} \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} Z_{te} + \frac{1}{2}\Delta Z_{te} \\ Z_{up} \\ tan(\alpha_{te} - \beta_{te}/2) \\ 0 \\ Z_{xxup} \\ \sqrt{r_{leup}} \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ X_{lo}^{1/2} & X_{lo}^{3/2} & X_{lo}^{5/2} & X_{lo}^{7/2} & X_{lo}^{9/2} & X_{lo}^{11/2} \\ 1/2 & 3/2 & 5/2 & 7/2 & 9/2 & 11/2 \\ \frac{1}{2}X_{lo}^{-1/2} & \frac{3}{2}X_{lo}^{1/2} & \frac{5}{2}X_{lo}^{3/2} & \frac{7}{2}X_{lo}^{5/2} & \frac{9}{2}X_{lo}^{7/2} & \frac{11}{2}X_{lo}^{9/2} \\ -\frac{1}{4}X_{lo}^{-3/2} & \frac{3}{4}X_{lo}^{-1/2} & \frac{15}{4}X_{lo}^{1/2} & \frac{35}{4}X_{lo}^{3/2} & \frac{63}{4}X_{lo}^{5/2} & \frac{99}{4}X_{lo}^{7/2} \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix} = \begin{bmatrix} Z_{te} - \frac{1}{2}\Delta Z_{te} \\ Z_{lo} \\ tan(\alpha_{te} + \beta_{te}/2) \\ 0 \\ Z_{xxlo} \\ -\sqrt{r_{lelo}} \end{bmatrix} \tag{3.2}$$

Finally, the coordinates of the airfoil are obtained with expression (3.3) and (3.4), where x is the vector of coordinates in the x-axis, usually normalised from 0 to 1.

$$Z_{upper} = \sum_{n=1}^{6} a_n x^{\frac{n-1}{2}} \tag{3.3}$$

$$Z_{lower} = \sum_{n=1}^{6} b_n x^{\frac{n-1}{2}} \tag{3.4}$$

Thus, we can represent an airfoil with 12 intuitive parameters, such as the ones from table 3.1. Note that the values of α and β are shown in radians, and in the example are -10 and 10 degrees respectively.

r_{LEup}	r_{LElo}	X_{up}	Z_{up}	X_{lo}	Z_{lo}	Z_{xxup}	Z_{xxlo}	Z_{TE}	ΔZ_{TE}	α_{TE}	β_{TE}
0.02	0.005	0.43	0.12	0.23	-0.018	-0.8	0.35	-0.01	0	-0.17	0.17

Table 3.1: Example of Parsec parameters

If systems (3.1) and (3.2) are solved with parameters from table 3.1, the following values are obtained.

a_1	a_2	a_3	a_3	a_5	a_6	b_1	b_2	b_3	b_3	b_5	b_6
0.141	0.838	-3.392	5.759	-5.136	1.790	-0.071	-0.336	-1.612	4.403	-4.880	1.822

Table 3.2: Intermediate values of Parsec parametrisation

With these values, the coordinates of an airfoil can be computed with expressions (3.3) and (3.4). Figure 3.2 is the result of computing the upper and lower side with an x vector of 100 points with a cosine distribution, which generates more points near the trailing-edge and the leading-edge and, as a consequence, more precision.

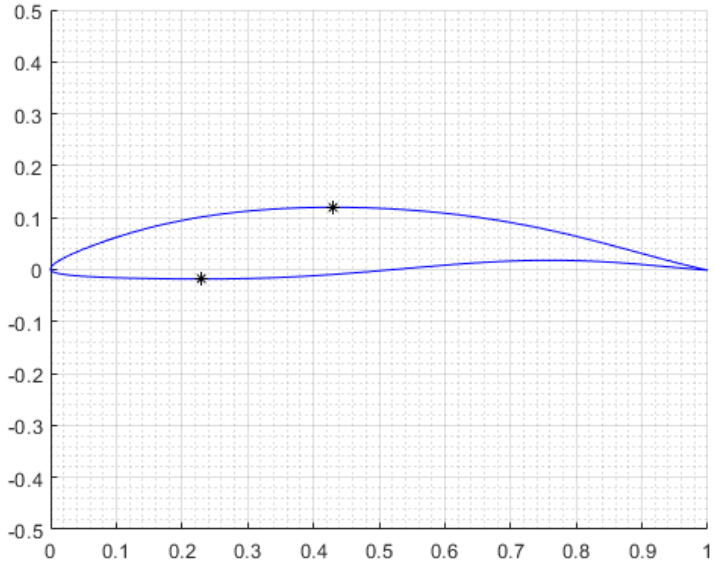


Figure 3.2: Airfoil representation with values from table 3.1.

3.2.2 Encoding

In this section, the transformation of the parameters from table 3.1 to a chromosome expressed in binary numbers is explained.

As explained before, at the beginning a first population of N individuals is generated randomly. In order to do so, it is needed to know which rang of values can take each parameter.

A MATLAB code has been created to test the PARSEC parameters limits and define the range. These parameters are modifiable and can be seen in the code, although the alteration of them can affect the final performance of the code.

Once the limits are set, it is needed to choose which data type (`char`, `int`...) will represent the genes of the chromosome. To do so, two aspects must be taken into account: the memory saving and the precision wanted in each gene. For instance, if a chromosome is conformed by gens that can take integer values from 0 to 10, since the precision needed is low, the smallest data type in terms of memory should be selected, which is the `char`, or also the `unsigned char`, which can store values from 0 to 255 and the storage size is 1 byte (8 bits). The problem arises when it is needed to store decimal variables, such is this case, because its representation in binary is complicated to handle. If so, it is needed to do line recombination and transform the original decimal value into an integer value that a given data type can store. The higher data type in terms of memory is the `long` or the `unsigned long`, which can store numbers from 0 to 18446744073709551615 (in the case of the `unsigned long`), with a storage size of 8 bytes (64-bits) [21]. However, selecting this data type may be unnecessary and inefficient because is too expensive in terms of memory. Thus, the objective is to find the lower data type capable of being precise enough for the problem.

In the case of study, the `unsigned short` has been selected, which has a storage size of 2 bytes (16 bits) and can store numbers from 0 to 65535. To sum up, in the current problem, each PARSEC parameter is transformed into an integer number linearising between the limits of each parameter and the limits of the `unsigned short`.

For instance, if the limits of the radius of the leading edge are [a,b] = [0.005,0.09], and the current radius is x = 0.02, the integer linearised value n between 0 and $2^k - 1$=65535 is (where k is the bit size of the data type):

$$n = \frac{x-a}{b-a}(2^k - 1) = \frac{0.02 - 0.005}{0.09 - 0.005}(2^{16} - 1) = 11565 \tag{3.5}$$

The inverse process is expressed by:

$$x = a + \frac{n}{2^k - 1}(b - a) = 0.005 + \frac{11565}{2^{16} - 1}(0.09 - 0.005) = 0.02 \tag{3.6}$$

Now that the decimal value is transformed into an integer value, it is possible to have a binary representation of it. Since the chosen data type is a `unsigned short` of 16 bits, the numbers will be 16 digits long, as it can be seen in table 3.3.

index	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*
2^n	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
binary number	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 3.3: Maximum number stored by an unsigned 16 bit data type.

The number 1111111111111111 in binary is transformed into decimal with the expression $2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 + 2^{10} + 2^{11} + 2^{12} + 2^{13} + 2^{14} + 2^{15} = 65535$.

For instance, the radius of the leading edge, r = 0.02, which is 11565 in binary, is expressed as:

index	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*
2^n	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
binary number	0	0	1	0	1	1	0	1	0	0	1	0	1	1	0	1

Table 3.4: 0.02 in binary representation.

Since $2^{13} + 2^{11} + 2^{10} + 2^8 + 2^5 + 2^3 + 2^2 + 2^0 = 11565$. Thus, 0.02 is 10110100101101 in this specific line recombination.

All in all, if we set limits for all the PARSEC values, the airfoil can be expressed as integer numbers from 0 to 65535 and then as a series of genes expressed in binary. The PARSEC limit values can be, for example, the ones from table 3.5. It is important to know that when executing the genetic algorithm code, these limits would determine the exploratory behaviour of the algorithm, because the offspring (new generations) would be always in these limits, so it is important to set them well.

	r_{LEup}	r_{LElo}	X_{up}	Z_{up}	X_{lo}	Z_{up}	Z_{xxup}	Z_{xxlo}	Z_{TE}	ΔZ_{TE}	α_{TE}	β_{TE}
lower limit	0.005	0.002	0.3	0.08	0.15	-0.07	-1.7	0.04	-0.02	0	-32^o	1^o
upper limit	0.09	0.0055	0.65	0.30	0.55	-0.018	-0.4	0.9	0.02	0.005	10^o	25^o

Table 3.5: Example of PARSEC limits.

Table 3.6 shows the airfoil from figure 3.2 expressed in integer values computed with limits from table 3.5.

r_{LEup}	r_{LElo}	X_{up}	Z_{up}	X_{lo}	Z_{lo}	Z_{xxup}	Z_{xxlo}	Z_{TE}	ΔZ_{TE}	α_{TE}	β_{TE}
11565	56172	24341	11915	13107	65535	45370	23623	16383	0	34327	24575

Table 3.6: Linearised Expressions of values from table 3.1.

Finally, the binary representation of this airfoil is computed using equations (3.5) and (3.6), and it is shown below.
0010110100101101 1101101101101100 0101111100010101 0010111010001011 0011001100110011 1111111111111111 1011000100111010 0101110001000111 0011111111111111 0000000000000000 1000011000010111 0101111111111111

Which is a chromosome of 12 genes of 16 bits, making a total of 192 bits for only one airfoil. Thus, the binary encoding of the airfoil has been completed and now it is possible to make the crossovers between the individuals.

3.2.3 Crossover

There are several types of crossovers: one-point crossover, k-point crossover, shuffle crossover, uniform, discrete, flat... [22, 23], yet this work does not intend to test them all or explain all of them. The purpose of this work is to develop a well functioning genetic algorithm and explain how it has been done. Thus, for its simplicity and utility, the one-point crossover has been the chosen one for this work.

The one point crossover consists of the fragmentation of the parents in one point to create two children combining the offspring of the fragmentation. The fragmentation point is created with a mask, which is a binary chain used to operate and modify binary numbers. Table 3.7 shows this process. The partition point is chosen randomly, the first child gets the first part of the parent 1 gene and the second part of the parent 2 gene, while the second child obtains the opposite. In the example below, the crossover is between 0.120 (11915) and 0.234 (45875) with limits (0.08,0.3). The first child is 11059, which is 0.117 and the second one 46731, which is 0.237 once the conversion is done.

Parent 1	0	0	1	0	1	1	1	0	1	0	0	0	1	0	1	1
Parent 2	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Mask	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
Offspring 1	0	0	1	0	1	0	1	1	0	0	1	1	0	0	1	1
Offspring 2	1	0	1	1	0	1	1	0	1	0	0	0	1	0	1	1

Table 3.7: Example of One-Point-Crossover

As it has been seen, the crossover has to be done gene by gene, so 12 crossovers must be done to obtain 2 children.

Figure 3.3 shows the result of a "forced" crossover in which the fragmentation point is the same for all genes of the chromosome. Normally, a random process would make it different for each of them. As can be seen, in this case, the changes are really minor and difficult to see. Since the mask makes a partition right in the middle, one of the children is very similar to parent 1 and the other to parent 2, although not equal.

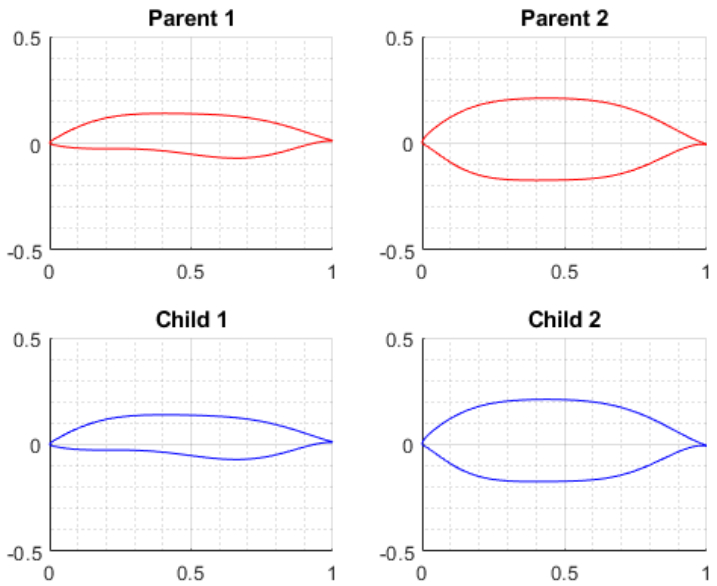


Figure 3.3: Crossover with mask = 1111111100000000 for each of the genes.

3.2.4 Mutation

The optimisation solvers can be either exploratory or exploitatory. Usually, they have a combination of both, determined by some parameter or process in the algorithm. In this case, the crossover takes the responsibility of the exploitatory part. The crossover between two individuals has the objective of extracting the best of them and find a better version of the parents. On the other hand, the mutation has an exploratory behaviour, since it introduces changes in the genetic code that modify the current solution to explore other ones. It helps the algorithm to avoid local minimums, by introducing new genetic information to explore more possible solutions [24].

In this work, 3 different mutations have been tested, 1-bit mutation, 2-bit mutation and heavy mutation.

1-bit mutation

The logic behind this mutation is very simple. If activated, it consists of changing one random bit of the children. This change can have a major or minor impact depending on the position of the mutated bit. The more to the left, the major impact it will have. The idea in every mutation is that it can happen with a certain probability, to be set by the programmer. Once the crossover is performed, a random number determines if the mutation will happen or not. If so, another random number determines the position in which the mutation will take place and the bit there is going to change from 1 to 0 or from 0 to 1.

The mutation "lottery" happens for each of the genes of the chromosome. In a real example two genes, 0.017 and 0.436, would have the following children using the mask

1110000000000000: 0.08295 and 0.4073. However, including a 1-bit mutation, the result changes more or less depending on the position of the mutation. In the example from table 3.8 the results are 0.08294 for the first child (a very minor change) and 0.3549 for the second child, a more important change since the mutated bit is the third one starting from the left:

Parent 1	0	0	1	0	1	1	1	0	0	1	1	1	1	0	1	1	11899	0.017
Parent 2	1	1	1	0	1	0	0	0	0	1	0	0	1	0	1	1	59467	0.397
Mask	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0		
Child 1	0	0	1	0	1	0	0	0	0	1	0	0	1	0	**1**	1	10315	0.08295
with Mutation	0	0	1	0	1	0	0	0	0	1	0	0	1	0	**0**	1	10313	0.08294
Child 2	1	1	**1**	0	1	1	1	0	0	1	1	1	1	0	1	1	61051	0.4073
with Mutation	1	1	**0**	0	1	1	1	0	0	1	1	1	1	0	1	1	52859	0.3549

Table 3.8: Example of crossover with 1-bit mutation

Figure 3.4 shows the result of 1-bit mutation for 3 different mutation probabilities, 30%, 60% and 90%. With a higher probability, more genes of the chromosome will mutate, however, it does not imply big changes necessarily. As has been seen, sometimes the changes produced due to a mutation are minor. The bit mutated plays an important role here, and the selection of this bit is completely random. Thus, the outcomes of a mutation are very unpredictable, although it is clear that with a higher probability, the changes, in general, would be more important.

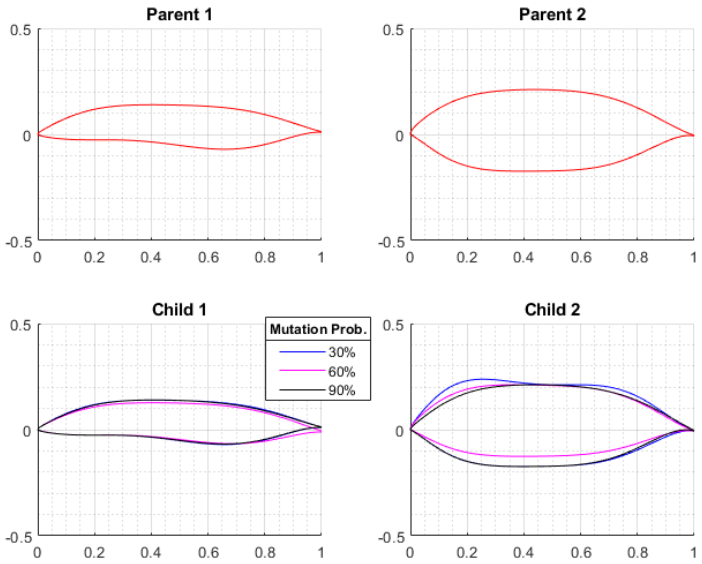


Figure 3.4: 1-bit mutation results.

2bit mutation

It is similar to 1-bit mutation, but in this case, the mutation happens in 2 different bits, and the position of these bits is also chosen randomly. It is a little bit more aggressive since the left bits have the double of the probability of being chosen and generate a major impact. Figure 3.5 shows the results of 2-bit mutation for 3 different probabilities as well

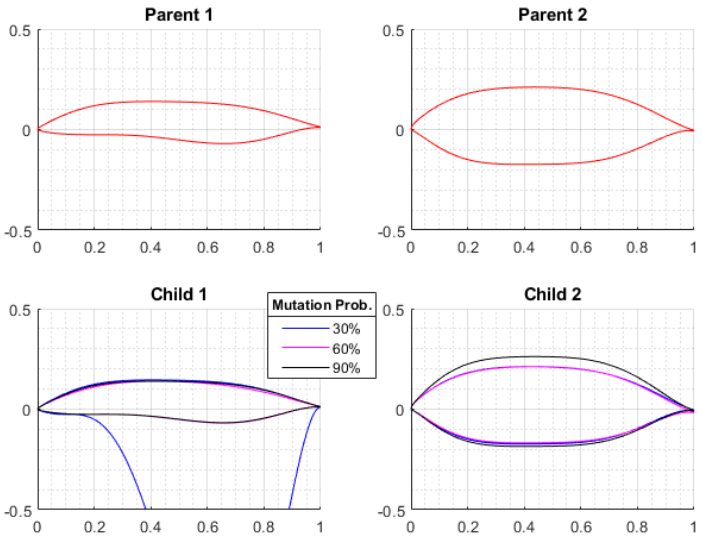


Figure 3.5: 2-bit mutation results.

Heavy mutation

This is the strongest mutation. It consists of mutating the bits located in even position and keep the odd ones intact. The mutation mask would be 0101010101010101, and the bits located in the same position than the 1's would change. Figure 3.5 shows the result of this mutation in three different probabilities. Compared to the previous mutations, it can be seen how it is hugely more aggressive and the changes produced are major.

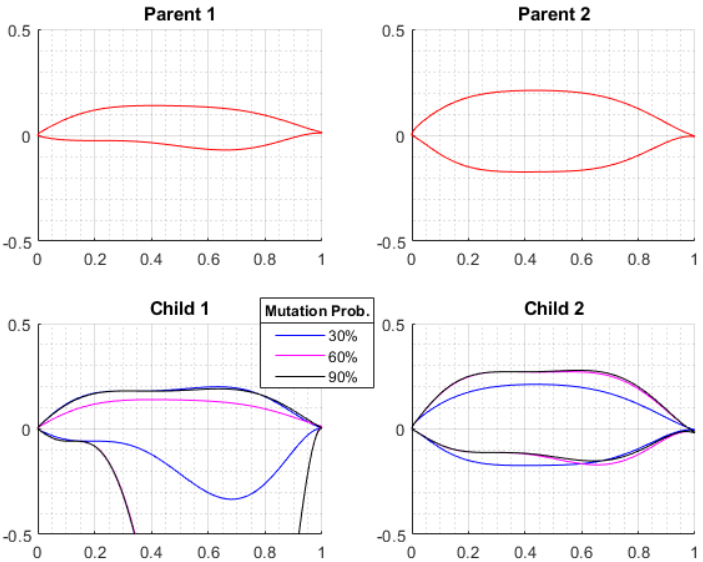


Figure 3.6: Heavy mutation results.

3.2.5 Fitness function

Until now, the process of generating an initial population and reproduce it has been explained. However, not every individual from the population is chosen for reproduction. As happens in real environments, some of the individuals do not make it to the reproduction and die before, while some others are able to reproduce several times. This is what is called the Natural Selection [25].

In the genetic algorithm, this process is computed with the fitness function. The fitness function is the way of computing a value that will determine the fitness or viability of a given individual to the system. For instance, in a snowy environment, preys with a dark skin colour would be less likely to survive on it, while the ones with light skin tones would have more chances. In this case, the fitness function would be a function that assigns high values to preys with a dark skin tone, and low values to preys with light skin tones. In the end, each of the individuals would have a number assigned, the fit, that would be key for the selection for reproduction.

In the case of the study, the fitness function can be computed in several ways. Perhaps, the easiest and powerful way to compute the fitness is via the airfoil Efficiency. It is important to understand that the fitness function is directly determining which parameter of the problem is going to be optimised. In this example is the Efficiency, but it could be the Cl, the Cd, a combination of various coefficients...

$$fit = \frac{1}{Eff} = \frac{Cd}{Cl} \tag{3.7}$$

In this work, the problem is designed as a minimisation problem, thus, it is wanted to obtain low values of the fitness function, and this is why the fit is computed as the inverse of the Efficiency, as shows equation (3.7). Other fitness functions used in the work can be seen in section 4.6.

Once the fit of each individual is computed, it is transformed into a probability. The lower is the fit, the higher is the probability to be chosen for reproduction, while infeasible individuals with infinite fit value, have 0 probability to be chosen for reproduction. All in all, each of the feasible individuals has an assigned probability, and a random number determines which of them is selected for reproduction. From a population of N individuals, N/2 would be selected for reproduction, and as explained, each couple of parents would have 2 children and a whole new generation of N individuals would be created. In this way, the individuals with major adaptability to the system (with a lower fitness function value) would have large probabilities to be chosen, but other individuals apparently less feasible, also have the chance of being selected for reproduction and transfer its genes to the next generation. With this system, the infeasible individuals are set aside and its genome is eventually lost, while the genome of feasible individuals is transferred through the populations and modified and mutated to finally met the final and optimal solution.

3.2.6 Pseudocode, the structure of the algorithm

With all the important aspects of the algorithm explained, it only lasts to show the scheme that the algorithm follows, which is pretty simple. As can be seen below, first, the initial population is created randomly. Then, the fitness is computed for each individual, and the best of them is saved into a variable called `best_fitness`. Then, while the solution is not found or certain conditions are not met, the population is reproducing and evolving. First, a whole new generation is created via crossover and mutation. Then, the fitness is evaluated again and the best of them is saved into `new_best_fitness`. If the new fitness is better than the previous one, the `best_fitness = new_best_fitness` and a new reproduction is performed. However, if the new fitness is worst than the one from the previous generation, a counter is activated and the reproduction continues. If this counter arrives at a certain value set by the programmer, the algorithm stops because it is considered that the solution has been found and it can not be more optimised. In addition, if the whole population is infeasible, and it happens a certain number of consecutive times, it is considered that the problem has ended without a solution and the algorithm also stops. It is also possible to stop the algorithm within a certain number of generations.

```
main{
    GenerateInitialPopulation;    // With a Random process

    for( individuals ){
        fits[i] = ComputeFitness;
        if(fits[i]<best_fit){
            best_fit = fits[i];
            best_individual = current_ind;
        }
    }

    while(!Solution){
        FitsToProbability;     // Vector of fits to probabilities
        for(individuals/2){
            NaturalSelection; // Selects two individuals for reproduction
            Crossover;        // Crossover + Mutation, 2 new individuals
        }

        // A new population is created, its fitness is computed now...
        for( individuals ){
            fits[i] = ComputeFitness;
            if(fits[i]<new_best_fit){
                new_best_fit = fits[i];
                new_best_individual = current_ind;
            }
        }

        /* Stoppping Algorithm */

        if( new_best_fit == inf ){ // Infeasible individuals are the ones with
            counter_infeasible ++; // infinite fitness
            if(counter_infeasible>limit1) break;  //exit while and END
        }else{
            counter_infeasible = 0;
            if( new_best_fit >= best_fit){        // Case in which the new
                counter_NO_improve++;             // generation is worst
                if ( counter_NO_improve > limit2){
                    Solution_individual = best_individual;
                    Solution_fitness    = best_fit;
                    Solution = 1;                 //exit while and end
                }
            }else{                                // Most common case,
                best_fit = new_best_fit;          // new generation improves
                best_individual = new_best_individual;
                counter_NO_improve = 0;
            }
        }
    }
}
```

Chapter 4

Implementation

4.1 Introduction

The development of such a project needs great organisation and control, since there are several processes to take into account, and it is easy to skip or misunderstand some of them. The code structure of this algorithm can be separated into two parts, the one related to the genetic algorithm part, and the one related to the aerodynamic computation. Fortunately, I have worked previously with both parts, and it has been useful to recover some of my previous works to review some of the code structures needed for this specific project. Specifically, I have reviewed my Bachelor Final Thesis, dedicated also to a basic optimisation of a blade's airfoil [26] and a project of the Master's dedicated to the development of a Genetic Algorithm for a cancer treatment [27, 28].

The code has been developed constructing small functions with very specific objectives, to end up with a large, complex and robust code. In this chapter, it is explained the code implementation of the processes theoretically described in the previous one.

4.2 Structure and Functions

The code is written in C language and, as a consequence, the freedom in the designing of the code structure is quite limited. In this case, the followed scheme is:

1. Libraries
2. Constants and Definitions
3. Declaration of Functions
    - Errors and User Information
    - Random numbers
    - Airfoil Parametrisation and Aerodynamic Computation
    - Genetic Algorithm
    - Solver of a System of Equations
4. Main Code
5. Functions Code

Below, the structure of the main code is shown in a flow diagram. As can be seen, it is a very simple scheme in which the function `GeneticAlgorithm` is called. However, notice that there are two calls to the system at the beginning and at the end. The objective of these two instructions is to execute and kill a parallel program that prevents the XFOIL, which is the software used for aerodynamic computation, to crack and block the main program. In section 4.6, this topic is described more exhaustively.

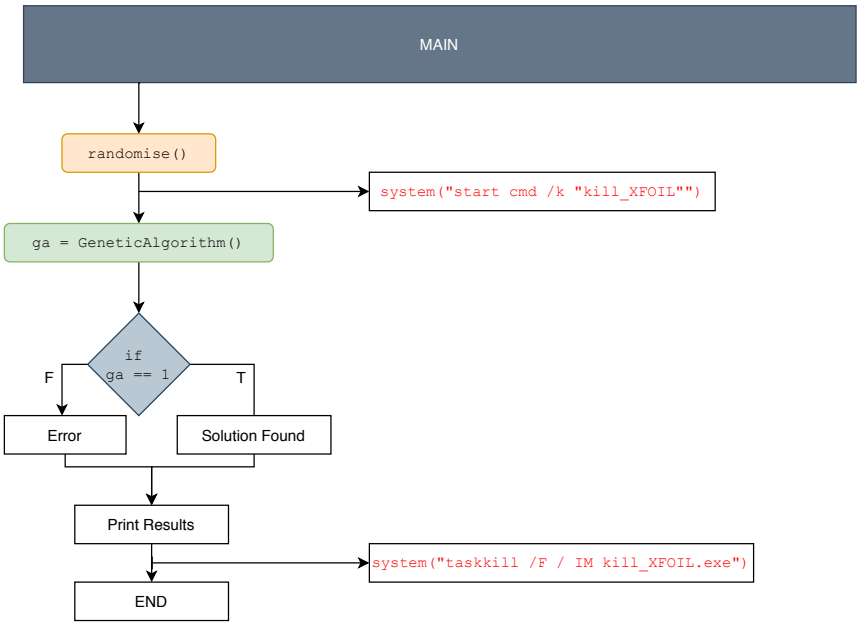


Figure 4.1: Main Flow Diagram.

The more important part of the algorithm is the `GeneticAlgorithm` function, and its flow diagram is shown in figure 4.2. In this flow diagram, every function used in the code can be seen. As explained before, there are 5 categories of functions, and all of them are expressed in different colours in the diagram: Blue for Errors and Information, Orange for Random Numbers, Red for aerospace-related computation, Green for Genetic functions and Purple for the System Solver.

The structure shown here is the same shown in the pseudocode from section 3.2.6. However, in this case, all the functions used in the code can be appreciated. As explained before, the first generation of random individuals is created using several random functions. The use of random functions is widely explained in section 4.2.2. Then the fitness function is computed. In order to do so, the function `GenerateAirfoil` is called. This function generates a `.txt` file with the airfoil coordinates of the current individual. To generate the airfoil, the system of equations (3.1) and (3.2) must be solved, and it is done using the *purple* functions, with the only objective of solving the system. Then the software XFOIL [4] is called using the `system` command. This solver generates an output file with the results of the current airfoil and the important data is taken and stored in the memory. This process is repeated with each individual and the best fit is stored.

Then there is the while section, in which the generations reproduce until one of the three stopping conditions are met. In here, the functions of the Genetic algorithm such as `Fits_to_Probability, Natural_Selection, Crossover` and `OnePointCrossoverMutation` are used. Notice that these functions use several random functions. The function `callXFOIl` is called again to compute the fitness of the new generation. Finally, there is the stopping algorithm part. The stopping conditions are:

1. Maximum number of generations is reached
2. Maximum number of repeated infeasible generations reached (all of them with infinite fitness)
3. Maximum number of not improving generations reached (which theoretically means that the solution can not be more optimised)

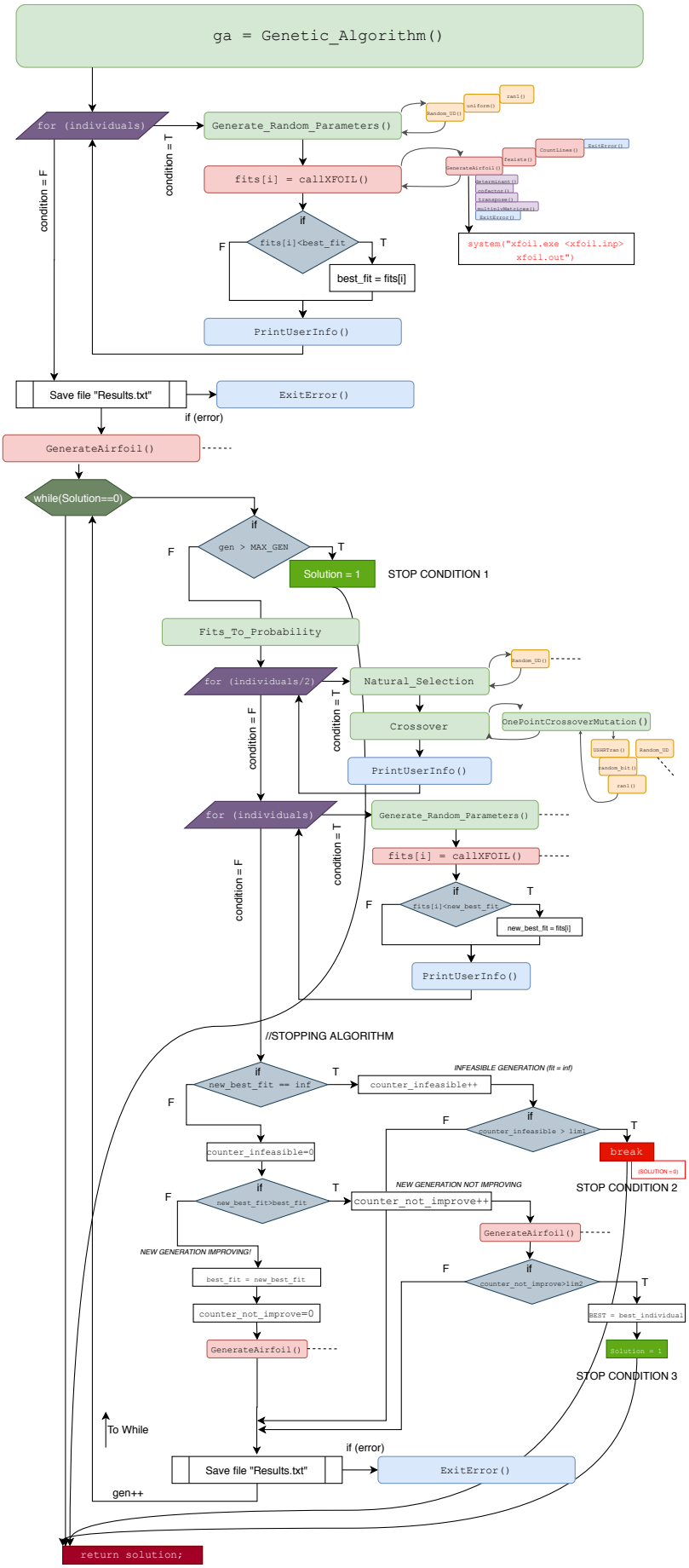


Figure 4.2: Genetic Algorithm Flow Diagram.

4.2.1 Errors and User Information

These functions are dedicated to give information of the code execution to the user.

```
void ExitError(const char *miss, int errcode);
```

This function stops the execution if there is an unexpected error and sends a message to the user indicating the type of error. The possible errors of the code are the following:

- Error 01 - the data file does not exist or cannot be opened
- Error 02 - the data file does not exist or cannot be closed
- Error 03 - Inverse of the matrix is not possible. Determinant = 0

The function takes as an input the error message and the error code.

```
void PrintUserInfo(int i, int ind);
```

This function prints the evolution of a loop execution. Since in this algorithm several loops are used, it is useful to know the evolution of them and check if the execution is stuck at some point. This function takes as an input the current index and the stopping condition of the `for` and prints the completed percentage.

4.2.2 Random numbers

These functions are dedicated to generating reliable random numbers. The random processes in the genetic algorithm are extremely important, and the use of `rand()` function from C is not an option since it is a pseudo-random generator in which the generated numbers are dependant of a seed [29]. The use of this function could generate biased answers, so this is why the random generator, `ran1()`, has been chosen from the Numerical Recipes in C [30].

```
double Random_UD(double i, double j);
```

This function returns a random double between the two inputs, i and j. In order to do so, it calls the function `uniform()`, which returns a random number between 0 and 1 with a uniform distribution and linearise the output to the input limits.

```
float uniform(void);
```

As explained, this function returns a random number in the interval (0,1) with a uniform distribution. In order to do so, it calls the function `ran1` with input `idum`, a negative integer to initialise the random generator. The initialisation is performed in the function `randomize()`.

```
float ran1(long *idum);
```

It is a function from the Numerical Recipes in C. This function generates a random number using the Minimal Standard for its purpose between 0.0 ans 1.0 excluding the endpoint value. According to references, there are no known statistical tests that `ran1` has failed, except for some specific cases with specific sequences of calls [30]. The input is an initialiser obtained from `randomize()`.

```
unsigned short USHRTran(void);
```

This function returns a random unsigned short, which means a number between 0 and 65535. It is used in the crossover and mutation, when dealing with binary operators. It uses the function `random_bit()` for its purpose.

```
unsigned char random_bit(void);
```

This function returns randomly a bit (1 or 0) using the function `ran1()`.

```
void randomize(void);
```

It is used to initialise the random process at the beginning of the code. It generates an initialiser taking data from the computer time.

4.2.3 Airfoil Parametrisation and Aerodynamic Computation

These functions are related with the computation of aerodynamic coefficients.

```
void Generate_Random_parameters(double *PAR_dbl, unsigned short *PAR_ush);
```

This function takes as an input two pointers to places in the memory where it has to allocate the data that will generate. Its objective is to generate the 12 random PARSEC parameters for an individual. It saves these parameters as `double` and also linearised as `unsigned short` for its use the genetic algorithm. To do so, it generates a random number between two predefined limits for each PARSEC parameter. The data is saved in the the vectors PAR_dbl and PAR_ush, which contain the information of each individual of the population, the first stored as `double` values and the second as `unsigned short`.

```
void GenerateAirfoil(double *PAR_dbl, char a_name[]);
```

This function is the one that solves the system of equations of the PARSEC parametrisation and generates a file with the current airfoil coordinates. To solve the system, it uses the "purple" functions shown in figure 4.2. The generated file will be used by XFOIL to compute the aerodynamic coefficients.

```
double callXfoil(double alpha, double Re, double Ma, double *PAR_dbl, double *Cl,
double *Cd, double *Cm);
```

This function calls XFOIL for execution. In order to do so, it generates an input file in which the specifications of the XFOIL analysis are set. These specifications are the fluid conditions such as viscosity, Reynolds number, Mach number, and also the angle of attack to do the analysis. The file created with `GenerateAirfoil` is used. Then the function calls XFOIL with a system command: `system("xfoil.exe < xfoil.inp > xfoil.out")`. When XFOIL is called and the computation completed, it generates a series of output files. The function checks if all the files are generated and if so, it reads the DATA.dat file, in wich the results of the analysis are saved. This file contains a list with the angle or angles of attack analysed, and the Cl, Cd, Cm, and more data for these angles. Te function reads the data and saves it in memory. Finally, it deletes all the generated files and computes the fitness function, which is the output of this function. Sometimes the XFOIL gets stuck during the execution because the input airfoil has a strange shape and XFOIL calculus can not converge. If so, there is a backup code running in parallel that cheks the elapsed time since the execution of XFOIL and, if it lasts more than 3 seconds to generate the output files, it means that XFOIL has cracked, so XFOIL task is killed, and the fitness function is computed as infinite.

```
int fexists(const char * filename);
```

This function is used in `callXfoil` to verify if some files exist in the working directory. It takes the file name as an input and the output is a 1 or a 0.

```
unsigned long CountLines(char *filename, int header_lines);
```

This function is used in `callXfoil` to count the lines of the output DATA.dat file to check if the computation has been successful. Sometimes, even though XFOIL does not crack, the DATA.dat file is empty. In this case, it is also needed to compute the fitness as infinite.

4.2.4 Genetic Algorithm Functions

These functions are related to the Genetic Algorithm. Besides the main genetic algorithm function, which it has already been seen, there are the functions related to the Natural Selection, the Crossover and Mutation.

```
int Genetic_Algorithm(double *PAR_dbl,double *newPAR_dbl, unsigned short *PAR_ush,
unsigned short *newPAR_ush, double **sol_PAR_dbl, double *fits, double * Fits2Prob,
double alpha, double Re, double Ma, double *generation_TOP_fit);
```

This function has already been seen in figure 4.2. It is the main function of the genetic algorithm. The inputs are basically the data structures used to save all the information, and the output is a 1 or a 0, meaning if the algorithm has worked or not.

```
void Fits_To_Probability(double *fits, double *Fits2Prob);
```

This function is the one that transforms the vector of fits into a vector of probabilities. In order to do so, it distributes, in a vector from 0 to 1, a certain space for each individual proportional to its fitness. The higher is the fitness, the lower is the space, having 0 space for infinite fitnesses. Thus, this function generates a vector of doubles Fits2Prob which is a vector from 0 to 1 divided into as elements as individuals in the population, and the individuals with a better fitness have more space assigned in this vector.

```
void Natural_Selection (double *Fits2Prob, double *PAR_dbl, unsigned short *PAR_ush,
double **C1, double **C2, unsigned short **C1ush, unsigned short **C2ush);
```

This function selects two individuals for reproduction. It is inside of a loop of half the individuals. In each iteration, it generates a random number between 0 and 1 and looks into the Fits2Prob vector to see to which individual correspond this random number and selects it as the parent 1. Then repeats this process for parent 2. As a consequence, individuals with a better fit to the environment will be more frequently selected, but also, individuals with a low fit can have the opportunity to be selected and transfer its genetic code to the next generation. On the other hand, infeasible individuals or individuals with a very high fitness value will hardly transfer their genetic code and will, eventually, disappear completely. This function generates two pointers to the two selected parents information.

```
void Crossover(double *newPAR_dbl, unsigned short *newPAR_ush, unsigned long *NewInd,
double *C1, double *C2, unsigned short *C1ush, unsigned short *C2ush, int d);
```

The main objective of this function is to generate a completely new population. In order to do so, it takes the two parents chromosomes, and gene by gene, it calls the `OnePointCrossoverMutation` to generate the complete genome of the two children. The function saves the information of the two new children in vectors newPAR_dbl and newPAR_ush.

```
void OnePointCrossoverMutation(unsigned short p1, unsigned short p2,unsigned
short *f1, unsigned short *f2, float prob);
```

This function takes as an input two genes and performs the crossover and mutation to generate the two genes of the offspring. The explanation of the implementation of crossover and mutation is widely explained in section 4.4.

4.2.5 Solver of a System of equations

These functions are dedicated to solve the PARSEC system of equations to generate the airfoil coordinates from the 12 PARSEC parameters. As a remember, the system (3.2) and (3.1) is a system with the following form:

$$Ax = RHS \tag{4.1}$$

Where A is a 6x6 matrix and x and RHS a 6x1 vector. Thus, to solve such a system the following recombination can be done:

$$A^{-1}Ax = A^{-1}RHS \rightarrow x = A^{-1}RHS \tag{4.2}$$

All in all, it is needed to make the inverse of A function to solve the system, so the following functions have been created:

```
double determinant(double a[6][6], double k);
void cofactor(double num[6][6], double f, double inv[6][6]);
void transpose(double num[6][6], double fac[6][6], double r, double inverse[6][6]);
void multiplyMatrices(double first[6][6], double second[6][1], double mult[6][1],
int r1, int c1, int r2, int c2);.
```

These functions generate the inverse by dividing the adjugate matrix of A and the determinant of A, and then solve the system by multiplying the inverse of A by the Right Hand Side of the system.

4.3 Data Structures

Lots of data structures are used in the code. However, the most important and peculiar may be `PAR_dbl`, `PAR_ush` and its derivatives. The names stands for PARSEC parameters stored in `double` and PARSEC parameters stored in `unsigned short`.

Basically, these structures contain the genetic information of the complete population. Since a chromosome of an airfoil has 12 genes, the size of a complete population is 12 times the individuals of the population, thus, at the beginning of the code, the space required is allocated into the memory. Then, as the computation advances, the results of one population are stored in these data structures. Once the calculus for one population is completed, the results needed are saved in a `.txt` file and then the results of the new population are overwritten in the same structures.

Table 4.1 shows the scheme of this data structure, which contains the chromosomes of each individual. Notice that in order to access to the information of, let's say, individual j, it is needed to access to position 12*(j-1), considering that the first position is 0.

individual 1				*individual 2*							*individual n*			
P_1^1	P_2^1	...	P_{12}^1	P_1^2	P_2^2	...	P_{12}^2	...	...	...	P_1^n	P_2^n	...	P_{12}^n

Table 4.1: Representation of PAR_dbl and PAR_ush data structure

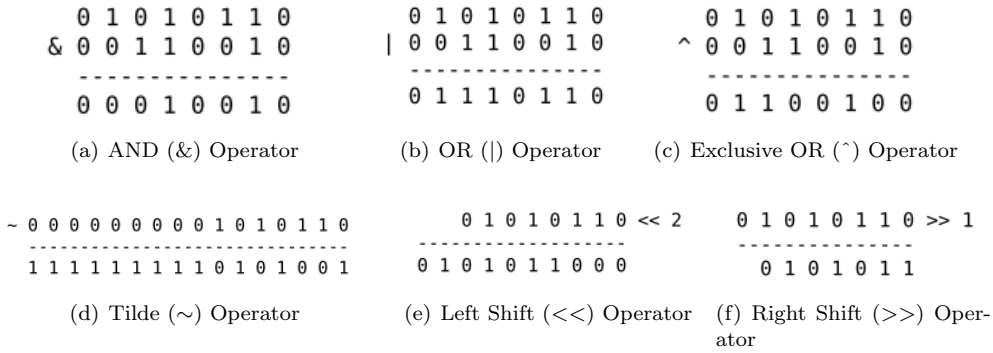
4.4 Crossover and Mutation, Bitwise Operators

This section is dedicated to the explanation of the code used to put into practice the crossover and mutations explained theoretically in section 3.2.3 and 3.2.4.

The main problems to solve are, how to treat the unsigned short values as binary numbers, how to combine these binary numbers to obtain the wanted results and how to create the masks. The answer of these questions are the bitwise operators [2].

In C, two numbers defined in any data type, will be combined as binary numbers if bitwise operators are used. The bitwise operators are the following. Check figure 4.3 for a better understanding.

- AND (&). The combination of two bits will be 1 only if the two of them are 1. Otherwise, the result will be 0.
- OR (|). The result of the combination is 1 if one of them is 1. Else, the result is 0.
- Exclusive OR (^). The result is 1 if one of them, but not both, is 1.
- Tilde (~). Is used to change the 1's per 0's and the 0's per 1's.
- Left Shift (<<). Shifts the first operand to the left as many times as the number given by the second operand. It adds zeros to the right.
- Right Shift (>>). Shifts the first operand to the right as many times as the number given by the second operand. The bits that go out of limits disappear.



(a) AND (&) Operator (b) OR (|) Operator (c) Exclusive OR (^) Operator

(d) Tilde (~) Operator (e) Left Shift (<<) Operator (f) Right Shift (>>) Operator

Figure 4.3: Bitwise Operators. Images from [2]

4.4.1 Crossover Implementation

As explained before, the idea behind the crossover is to create a mask that generates a fragmentation point and from the genes of parent 1 and parent 2, generate two children combining these genes.

As it has been seen in table 3.7, the mask for `OnePointCrossover` is a series of 1s followed by a series of 0. Thus, the first step is to generate an unsigned short (16 bits) full of ones, which is the number 65535, as seen in table 3.3. In order to do so, hexadecimal representation is used. The mask is defined as 0xFFFFU, which is an unsigned 65535 in decimal representation, and then it will be shifted a certain amount of positions to the left to generate the final mask. The shift operator will be used, and the number of positions displaced is selected by a random number.

This random number is computed doing the modulus between a random number between 0 and 65535 (`USHRTran()`) and 15 (`USHRT_WIDTH-1` = 16-1). The result of this modulus will be always an integer between 0 and 14. Thus, a 1 is added to this result to obtain numbers

between 1 and 15. This operation is the first line of the code below. The result is a 16 bit number with a certain amount of 1's and the rest of 0s.

Then, to generate the children, the operators AND (&), OR (|) and Tilde (∼) are used. First, parent 1 and the mask are combined with an AND. This will create a copy of the parent from the start to the fragmentation point, as can be seen in the simple example from table 4.2, upper left. At the same time, the parent 2 is combined with the inverse of the mask (using the tilde), thus, a copy of the right-hand side of parent 2 is obtained. These two results are combined with an OR to obtain the children 1 (f1). Regarding the children 2, the same operation is performed but is the parent 1 who is combined with the tilted mask, and the parent 2 is combined with the mask. These operations can be seen in table 4.2, notice than p2&mask and p1&∼mask operations are not shown in the table.

p1		**1**	**0**	1	1	***p2***		1	1	**0**	**1**
mask	&	1	1	0	0	*∼mask*	&	0	0	1	1
p1 & mask		**1**	**0**	0	0	*p2 & ∼mask*		0	0	**0**	**1**
p1 & mask		**1**	**0**	0	0	*p2 & mask*		1	1	0	0
p2 & ∼mask	|	0	0	**0**	**1**	*p1 & ∼mask*	|	0	0	1	1
f1		1	0	0	1	***f2***		1	1	1	1

Table 4.2: Example of a crossover

All these operations are quite simple to write in C language, and they can be seen in the code below.

```
unsigned short mask =  0xFFFFU << (1 + (USHRTran() % (USHRT_WIDTH - 1)));
*f1 = (p1 & mask) | (p2 & ~mask);
*f2 = (p2 & mask) | (p1 & ~mask);
```

4.4.2 Mutation Implementation

As explained in Chapter 3, in this work 3 mutations have been tested. In all of them, the objective is to create a mask of 16 bits with a certain amount of 0 and 1, and combine this mask with the children to modify them. Specifically, the position in which there is a 1 in the mask will be changed in the children. In addition, the mutation does not always happen, it has a probability set by the programmer. The instructions to generate the mutation can be seen below. If a uniform random number is lower than a certain probability, the child is mutated. This lottery is played one time for each child. If the mutation happens, the mask is combined with an exclusive OR, which will modify the bits of the child affected by the mutation.

```
*f1 = (p1 & mask) | (p2 & ~mask);
if(uniform() < MUTATION_PROBABILITY){
    *f1 = *f1 ^ mask_mut1;
}
*f2 = (p2 & mask) | (p1 & ~mask);
if(uniform() < MUTATION_PROBABILITY){
    *f2 = *f2 ^ mask_mut2;
}
```

The creation of the mutation masks is explained below.

1-bit Mutation

The objective is to generate a mask of 16 bit full of 0 except for one random bit, which will be a 1. To do so, the first step is to create the number 1 as an `unsigned short` (16 bits), which is the variable called `oneU` in the code below. Then, this 1, treated as binary, is shifted to the left a certain amount of positions. The number of shifted positions is set by a random number created with the operation `USHRTran() % USHRT_WIDTH)`, which as explained in the Crossover, gives numbers from 0 to 15. The operation is performed twice,

one for each child, even though it may be unused if the mutation probability is not met.

For instance, table 4.3 shows an example of the creation of the mask. The number 5 can be obtained if the random number offered by `USHRTran()` between 0 and 65535 is 32757, since 32757%16 = 5.

oneU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	<< 5
mask_mut	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	

Table 4.3: Example of creation of a mask for 1-bit mutation

The code to generate the mask for 1-bit mutation in C language can be seen below.

```
1 unsigned short oneU = 1U, mask_mut1, mask_mut2;
2 mask_mut1 = (oneU << (USHRTran() % USHRT_WIDTH));
3 mask_mut2 = (oneU << (USHRTran() % USHRT_WIDTH));
```

2-bit mutation

To generate the mask for the 2-bit mutation, the process is a little bit more complicated. The first step is to randomly select the two positions in the mask in which it is wanted to set the 1 bits. For this example, lets say position 3 and 12.

First, an `unsigned short` 1 (`oneU`) of 16 bits is generated, and it is shifted 12-3=9 times to the left (step 1). Then, a 1 is added to this number. In binary, to add a one it is only necessary to add a it in the last right bit of the chain (step 2). Then, this number is shifted again to the left 3 times. In this way, a binary number with two 1's, one in position 3 and another one in position 12 is created. Notice that the first 1 has been shifted to the left two times, first 9 positions, and then 3 positions (12 in total), while the second number has only been shifted 1 time 3 positions to the left. This process can be seen in table 4.4.

pos	*15*	*14*	*13*	*12*	*11*	*10*	*09*	*08*	*07*	*06*	*05*	*04*	*03*	*02*	*01*	*00*	
oneU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	<< 9
step1	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	
step1	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	
+	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
step2	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	
step2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	<< 3
mask_mut	0	0	0	**1**	0	0	0	0	0	0	0	0	**1**	0	0	0	

Table 4.4: Example of creation of a mask for 2-bit mutation

The code for creating this mask is shown below. First, two random numbers between 0 and 15 are created. Notice that it has to be checked that these two numbers are not the same. Then, a routine checks which of the two numbers is higher so the first operation does not result in a negative number. Then, the mask is created with the command from line 13. The same process is done for `mask_mut2`.

```
1  unsigned short n1 = 1, n2 = 1, pos1, pos2;
2  while (n2 == n1){
3      n1 = Random_UD(0,15);
4      n2 = Random_UD(0,15);
5  }
6  if(n1>n2){
7      pos1 = n1;
8      pos2 = n2;
9  }else{
10     pos1 = n2;
11     pos2 = n1;
12 }
13 mask_mut1 = ( ( oneU << (pos1-pos2) ) + 1 ) << pos2;
```

Heavy Mutation

This mutation is the easiest to implement since is always the same and does not include random routines. The objective is to create a 16-bit mask filled with 0 and 1 successively. Thus, using a binary converter [31] it is possible to know which number defines such a mask. This number is 21845, expressed as 5555 in hexadecimal. The code to implement this mutation mask is shown below.

```
mask_mut1 = 0x5555U;
mask_mut2 = 0x5555U;
```

4.5 Choosing the parameters

Once the genetic algorithm is implemented, it still needed to determine which will be the parameters used in the code, since they will determine the performance and effectiveness of it. These parameters are the initial population, the mutation type, the mutation probability and, regarding the stopping conditions, the maximum number of generations allowed, the maximum number of generations evolving without improvement, and the maximum number of consecutive infeasible generations. In addition, the execution time must be also taken into account.

To determine these parameters a series of executions of the code combining them has been implemented. The main idea is to see the effects on the final results of each parameter in terms of time and the final optimal solution. The objective function for the test has been the Efficiency and the flow conditions are α=1º, Ma = 0.35 and Re = 1e6.

The combination of TEST 1 has been the following:

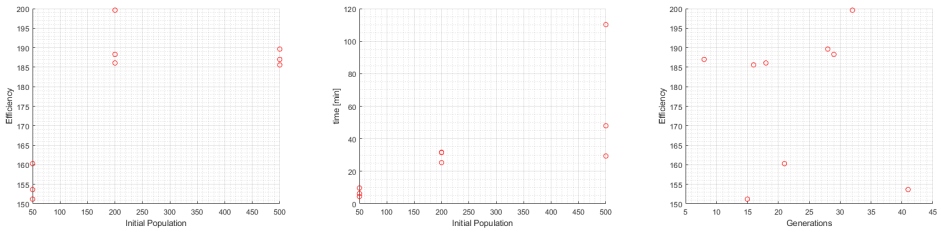
- initial population: 50, 200 and 500
- Mutation type: 1 (1-bit mutation), 2 (2-nit mutation) and 3 (heavy mutation)
- Mutation Probability: 0.05, 0.2 and 0.4
- Maximum number of generations: 150
- Maximum number of consecutive generations not improving: 6
- Maximum number of consecutive infeasible generations: 9

The test has been initially executed with only Mutation type 1. With only this test, it has been decided to abort the rest due to the amount of time used to complete it.

The main conclusions obtained are that an initial population of 500 is not worth for this problem, since the amount of time required for the analysis is extremely huge and, with 200 individuals, an optimal solution is found quicker, as it can be seen in figure 4.4-a, where the final efficiency obtained by the cases with an initial population of 500 is very similar than with 200, while the time is extremely larger (figure 4.4-b).

It also has been seen that the optimum solution for the Efficiency for these flow conditions seems to be very close to 200, being it the limit. Figure 4.5 shows the evolution of the efficiency through the generations for initial population of 50 (red), 200 (blue) and 500 (black). As can be seen, with an initial population of 50, the solution found is not the optimal one. With 200 and 500, most of the tests tend to a horizontal asymptote in 200. It is also notable that with an initial population of 500 it arrives with fewer generations to the final solution, but the amount of time needed is higher, as it has been already seen.

Finally, figure 4.4-c shows the relation between the final generation and the final efficiency. As can be seen, there is no pattern, since the number of generations needed to obtain a solution is not a variable of the problem, but a consequence of the chosen parameters.



(a) Population vs. Efficiency (b) Population vs. time (c) Final Generation vs. Efficiency

Figure 4.4: Results from Test 1, Mutation type 1

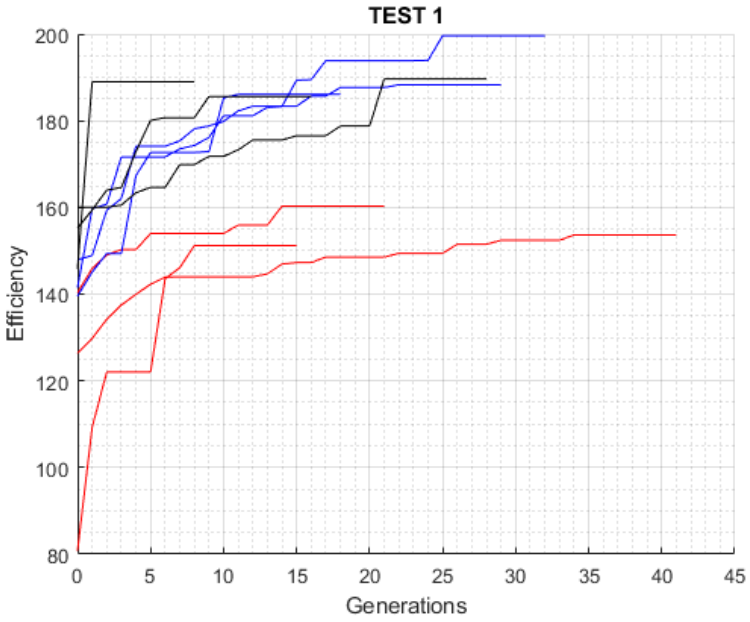


Figure 4.5: Generations vs. Efficiency (R= 50, Bl = 200, Bk = 500)

With these conclusions, it is reasonable to think that the optimal initial population would be between 50 and 200. Since only mutation 1 has been tested, it would not be fair to analyse the mutation probabilities. Thus, a second test is performed with the following conditions, including more mutation probabilities to obtain more data.

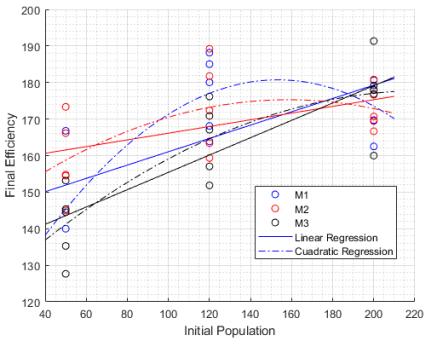
TEST 2:

- initial population: 50, 120 and 200
- Mutation type: 1, 2 and 3
- Mutation Probability: 0, 0.1, 0.25, 0.4, 0.55
- Maximum number of generations: 150
- Maximum number of consecutive generations not improving: 6
- Maximum number of consecutive infeasible generations: 9

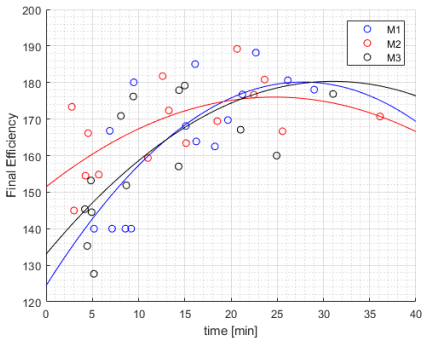
In this analysis the three different mutations have been compared.

Until now, it has been known that the optimal population size is between 50 and 200. Figure 4.6-a shows the final efficiency depending on the population size (50, 120, and 200). This figure also includes a linear and quadratic regression. In general, terms, looking at the linear regression and the previous figures, it can be said that the higher is the initial population, the higher would be the final efficiency. However, looking at the quadratic regression, for mutation types 1 and 2, the optimal population size seems to be around 150. Since population size and time are directly related, if the Efficiency is plotted with time, a similar figure is obtained.

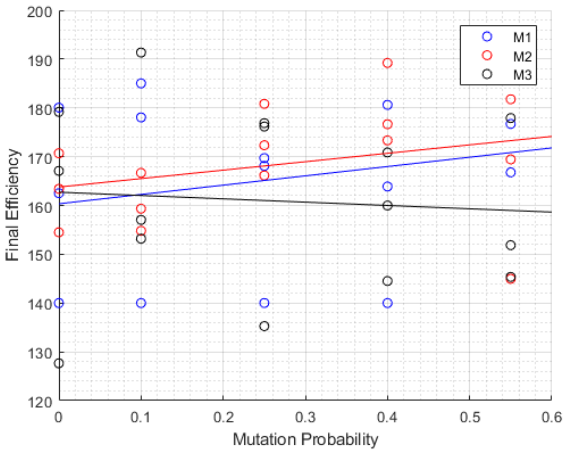
Figure 4.6-c shows the relation between the mutation probability and the final result, which is a very crucial parameter to determine. As can be seen, for mutation type 1 and 2, which are soft mutations, the higher is the mutation probability, the higher is the final efficiency. On the other hand, it seems that mutation 3 loses performance when its probability is increased. It may be logical to think that mutation 3 is so aggressive that it strongly disturbs the final results and ends up being counterproductive.



(a) Population vs. Efficiency



(b) Time vs. Efficiency



(c) Mutation Probability vs. Efficiency

Figure 4.6: Results from Test 2

Nevertheless, with all this information, it is still hard to decide which are the optimal parameters to make the genetic algorithm excel in its performance, in terms of final result and time.

A new analysis with some major changes is performed again. First, the initial population for each case of study is the same. In this way, all of them start from the same initial conditions, and the case of one random initial population being slightly better than the other ones is avoided. In addition, the population size is set as 140, since it is a reasonably good value even though it is known that it may not be the optimal one. Finally, the maximum number of generations is limited to 30 since it has been seen that the majority of optimal solutions are found before generation 30.

TEST 3:

- initial population: 140
- Mutation type: 1, 2 and 3
- Mutation Probability: 0.0, 0.1, 0.2, 0.3, 0.4, 0.5
- Maximum number of generations: 30
- Maximum number of consecutive generations not improving: 6
- Maximum number of consecutive infeasible generations: 9

Figure 4.7 shows the mutation probability versus the Relative Efficiency, like in figure 4.6-c. The Relative Efficiency has been defined considering a maximum theoretical efficiency of

200. Similar as in test 2, for Mutation 1 the final efficiency improves as the mutation probability increases. However, the performance of mutation 2 is, in this case, decreasing, and mutation 3 follows this same pattern. It is complicated to extract solid conclusions about this change in behaviour of mutation 2. In addition, it is complicated to select a mutation probability and a mutation type between 1 and 2, since the final results are very similar in terms of Efficiency, excluding time. However, it can be confirmed that mutation 3 does not work well with this problem, at least with such probabilities. It could be possible that the performance would be better with mutation probabilities 10 times lower, between 0 and 0.05.

Figure 4.8 shows the evolution of efficiency through generations for Test 3. In this case, it is also hard to find a pattern of growth for mutations 1 and 2. It can be seen though, that with high mutation probabilities in Mutation 3, the growth is very fast at the beginning, but then it slows down and the final results are less satisfactory.

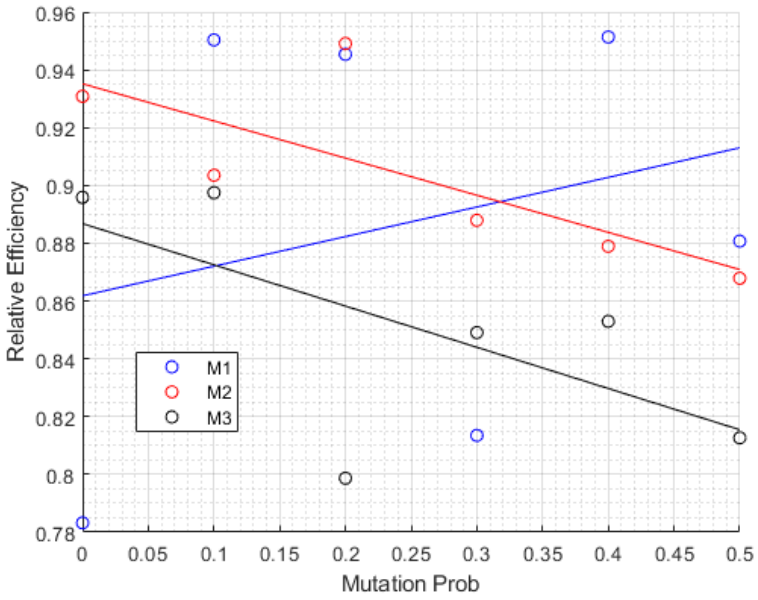
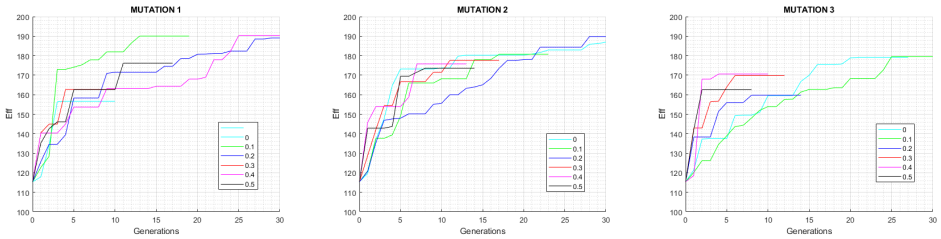


Figure 4.7: Mutation Probability vs. Relative Efficiency



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

Figure 4.8: Results from Test 3

A final test is performed with the objective of eliminating the variables that must be taken into account and make a better decision, just as in TEST 3, where the initial population has been set as 140. Now, the code is modified to stop the execution if a test arrives at an efficiency of 180 since it is a good result (is 90% of the theoretical best Efficiency). The maximum number of generations has been reduced to 25 and the consecutive generations without improving have been increased to 25 so it never stops before generation 25. In addition, the mutation probability 0 has been removed and substituted by 0.6.

TEST 4:

- initial population: 140
- Mutation type: 1, 2 and 3
- Mutation Probability: 0.1, 0.2, 0.3, 0.4, 0.5 0.6
- Maximum number of generations: 25
- Maximum number of consecutive generations not improving: 25
- Maximum number of consecutive infeasible generations: 9

With this test completed, a final plot has been made to finally select the combinations of parameters. In order to do so, the results of the executions in which the final efficiency has not arrived at 180 are not represented in the plot. Thus, the efficiency is approximately the same for each of the data points, which is a value higher than 180, considered a good result.

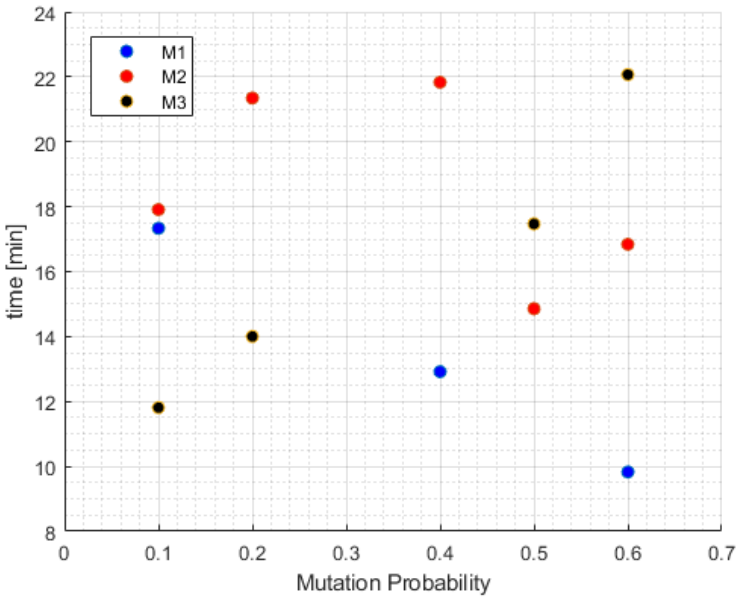


Figure 4.9: Mutation Probability vs. time

Figure 4.9 shows the results of test 4 and it is extremely helpful for drawing the conclusions reached at this point. First, it confirms that mutation 3 is not valid for this problem, at least with the probabilities set in this test. As can be clearly seen, the time increases with the probability. Secondly, it can be seen that the behaviour of mutation 2 is still hard to predict, it seems that random processes to which it is subjected combined with the specifications of this problem, make it very unpredictable. As it has been seen in figure 4.6-c and 4.7 it behaviour has changed completely. However, mutation 1 offers a clear pattern. It can be seen how the time decreases as the probability of mutation increases. Somehow, the combination of the 1-bit mutation with these concrete probabilities offer good results for this specific problem.

Now, another question arises and is if the time would be even less with higher mutation probabilities. In general, the results with these mutations and a specific mutation probability can change from one execution to the other, due to the random processes that take part. However, mutation 1 seems to have a more deterministic behaviour in this process, and this is why is chosen as the base mutation for this problem. To more accurately determine the mutation probability, a final test is performed, only with mutation 1.

TEST 5:

- initial population: 140
- Mutation type: 1 (1-bit mutation)
- Mutation Probability: 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0
- Maximum number of generations: 25
- Maximum number of consecutive generations not improving: 25
- Maximum number of consecutive infeasible generations: 9

The results of this test are shown in figure 4.10, in which two new points are included. It confirms the pattern recently discovered, in which the higher is the mutation probability, the lower is the time to reach an optimal solution. However, notice that only mutation probabilities of 0.6 and 0.7 have reached this solution, the rest could not make it but reached solutions with efficiencies slightly below 180. As said before, the random processes modify the results from one run to another, but the general tendency of this combination of parameters is maintained.

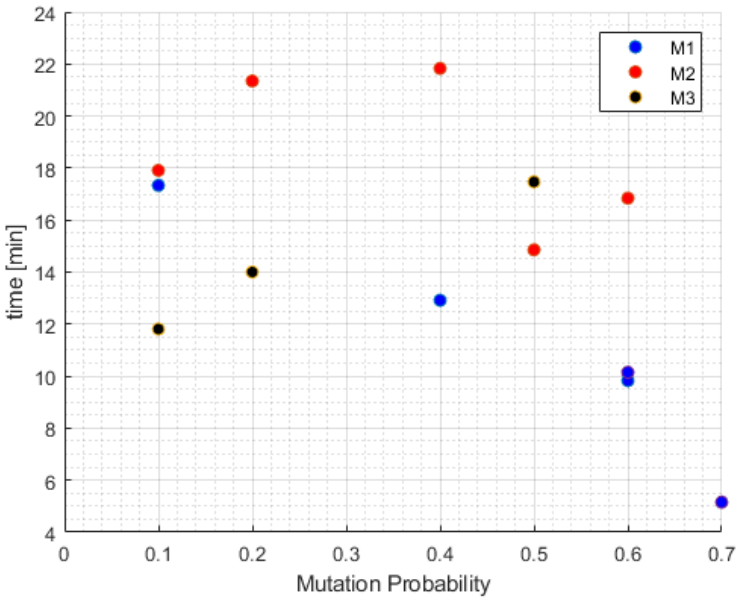


Figure 4.10: Mutation Probability vs. time - extended

All in all, the final combination of parameters is the following:

- Initial Population: 140
- Mutation Type: 1
- Mutation Probability: 60-70%
- Maximum Number of Generations: 100
- Maximum Number of consecutive generations without improvement: 8
- Maximum Number of consecutive infeasible generations: 9

However, notice that some of these parameters can be modified to improve the final result sacrificing the execution time, which would be larger. In general, the initial population can be increased until 200, and the probabilities of found a better solution would be increased. The same happens with the number of consecutive generation without improvement.

4.6 Fitness Function and data Computation

In section 3.2.5, the concept of fitness function has been explained. In the code, 4 types of fitness functions are implemented. The user has to decide in which one is interested in.

The first fitness function is the most simple yet effective, optimise the efficiency. The fit is defined as the inverse of the efficiency, which is computed as the ratio between the C_l and the C_d. Is widely used in several optimisation algorithms applied to airfoils.

However, when optimising the efficiency, sometimes the lift coefficient obtained is not good enough for the desired objectives. In such a case, the fitness function2 should be selected. This function optimises the proximity to a goal C_l and also minimises the drag. Equation (4.3) shows this idea. Since the error respect to the goal Cl is squared, high errors would highly amplify the value of this function. In general, with this fitness function, the genetic algorithm would quickly approximate to the goal C_l and then try to minimise the C_d. Thus, this function maximise the efficiency of an airfoil with a determined C_l. This fitness function is also used in airfoil optimisation problems [17].

$$min[(C_l - Cl^T)^2 + C_d] \tag{4.3}$$

In addition, two more fitness functions are included in the code, which are: maximise the C_l (or minimise $\frac{1}{C_l}$) and minimise the C_d. However, these fitness functions are only useful as a test or check, being the more important and useful the first two of them. In the code below, the implementation of these fitness functions can be seen.

```
Ef = (*Cl)/(*Cd);
if (FITNESS_TYPE == 1){
    fits = 1/Ef;
}
if (FITNESS_TYPE == 2){
    fits = (*Cl-ClT)*(*Cl-ClT) + *Cd;
}
if (FITNESS_TYPE == 3){
    fits = 1 / *Cl;
}
if (FITNESS_TYPE == 4){
    fits = *Cd;
}
```

As explained before, XFOIL is used to compute the aerodynamic coefficients of an airfoil. In order to call XFOIL, a .txt file with all the inputs must be created. Below, a part of the code that calls XFOIL can be seen.

```
FILE * xfoil;
        xfoil = fopen("xfoil.inp","wt");
        if (xfoil == NULL) ExitError("the data file cannot be opened",01);
        fprintf(xfoil,"plop\n");
        fprintf(xfoil,"G\n");
        fprintf(xfoil,"\n");
        fprintf(xfoil,"load %s\n",airfoil_name);
        fprintf(xfoil,"nombreairfoil\n");
        fprintf(xfoil,"OPER\n");
        fprintf(xfoil,"ITER 100\n");
        fprintf(xfoil,"\n\n");
        fprintf(xfoil,"OPER\n");
        fprintf(xfoil,"RE %f\n",Re);
        fprintf(xfoil,"MACH %f\n",Ma);
        fprintf(xfoil,"VISC\n");
        fprintf(xfoil,"PACC\n");
        fprintf(xfoil,"\n\n");
        fprintf(xfoil,"as %f %f %f\n",alpha_i,alpha_f,delta_alpha);
        fprintf(xfoil,"dump xfoil_dump.dat\n");
        fprintf(xfoil,"cpwr xfoil_cpwr.dat\n");
        fprintf(xfoil,"pwrt\n");
        fprintf(xfoil,"DATA.dat\n");
        fprintf(xfoil,"plis\n");
        fprintf(xfoil,"\n");
        fprintf(xfoil,"QUIT");
    if (fclose(xfoil)!= 0) ExitError("the data file cannot be closed",02);
```

In these lines all the commands that should be added manually on XFOIL are shown. The inputs are the airfoil, the maximum number of iterations, the flow conditions, the name of the output files, and other configuration options. As can be seen, a file is generated and in then, with the instruction `system("xfoil.exe < xfoil.inp > xfoil.out")` XFOIL is executed.
However, sometimes the execution of XFOIL is not satisfactory. XFOIL is designed to compute aerodynamic coefficients of airfoils, but sometimes, due to the random process of the code, the shape sent to XFOIL is hardly definable as an airfoil. This usually happens with the first generation, where all the shapes are generated randomly, or after some hard mutation. In some of these cases in which XFOIL can not converge and give a solution, it gets stuck. In order to solve this problem, a parallel routine has been implemented. At the beginning of the execution, a call to another program is done with the following commands:

```
char *command = "start cmd @cmd /k \"kill_XFOIL\" ";
system(command);
```

This commands execute the program `kill_XFOIL.exe`, which its main function is shown below:

```
int main(){
    int a = 0;
    while (a == 0){
        condicio = 0;
        t_start = clock();
        while (condicio == 0){
            t_end = clock();
            elapsed_time = ((double) (t_end - t_start)) / CLOCKS_PER_SEC;
            if(fexists("DATA.dat") && fexists("xfoil.inp") &&
            fexists("xfoil.out") && fexists("xfoil_dump.dat") &&
            fexists("xfoil_cpwr.dat")){
                condicio = 1;
            }else{
                if (elapsed_time>tmax){
                    printf("killing\n");
                    system("taskkill /F /IM xfoil.exe");
                    condicio = 1;
                }

            }

        }
    }
}
```

This code enters in a never-ending *while* loop (since variable `a` is always defined as 0). In this loop, it is continuously checking if a series of files exist. If they do, it means that XFOIL is working, solving the problem, and returning a series of outputs. Since after every execution of XFOIL these files are deleted, if some of these files do not exist in the directory for enough time, it means that XFOIL has got stuck and has not returned some of the outputs. The code has a clock that checks if these files do not exist for more than 3 seconds. If so, the command `system("taskkill /F /IM xfoil.exe");` is executed and XFOIL is killed. At the same time, in the main code a similar routine is waiting for these files to be created. If they are not created it is considered that the execution has failed and the fitness function is set as infinite.

When the genetic algorithm has ended, the following command terminates this program:

```
system("taskkill /F /IM kill_XFOIL.exe");
```

4.7 Constraints

In the code, it has been included the possibility of setting some constraints regarding the geometry of the airfoil or its aerodynamic behaviour.

Five constraints have been included in the code, 4 of them regarding the geometry of the airfoil.

4.7.1 Maximum Thickness

It is possible to set a maximum thickness of the generated airfoils, which can be very useful for construction reasons, for example. In this case, the thickness of the airfoil is computed after the airfoils have been generated. If the actual thickness does not fit the constraint, the fitness of the current airfoil is set as infinite. If so, the genetic information of these kinds of airfoils would end up being lost and only airfoils with the asked thickness will survive.

4.7.2 Minimum Thickness

It is also possible to set a minimum thickness of the generated airfoils. This tool can be very useful if it is known the size of the main beam of the wing since it is possible to ensure that it will fit in the airfoil.

As explained before, the thickness is computed after the airfoil has been generated.

4.7.3 Trailing Edge Thickness

The trailing edge thickness is the distance between the upper side of the airfoil in the trailing edge and the lower side. It is possible to set a given distance for this value. In this case, since this distance is a PARSEC parameter, all the airfoils will have this exact Trailing-Edge Thickness, so this constraint is more efficient computationally.

4.7.4 Minimum Trailing-Edge Angle

As the previous case, the trailing edge angle is a PARSEC parameter and its limits can be easily modified to implement the constraint.

4.7.5 Minimum Cm

This constraint allows us to set a minimum value of Cm to not overpass. It usually happens that a high (negative) value of Cm is obtained when an airfoil is optimised considering only the Efficiency. High values of Cm are usually counterproductive since they imply bigger horizontal surfaces to achieve the aircraft to be stable. Thus, it is very important to be able to set this parameter of design.

This constraint is computed as the thickness after the airfoils have been created and analysed, the value of the Cm is checked and if it does not meet the constraint, its fit is set as infinite.

4.8 Validation

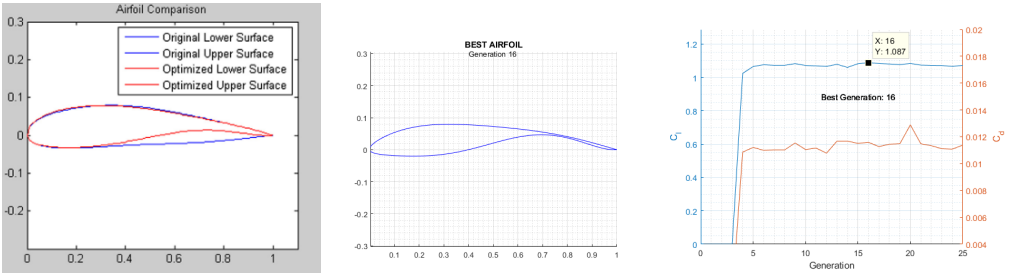
The code has been validated comparing it with similar works found in papers. In addition, a special case in which an initial population is composed only of cylinder-like shapes has also been tested. The validation of the aerodynamic data is also studied.

4.8.1 Comparison with a Genetic Algorithm

In this project, a genetic algorithm is used to optimise an airfoil shape defined with PARSEC parameters like in this work. The aerodynamic data is computed with a self-developed panel method, and the results are tested in a wind tunnel representation [18].

The PARSEC limit values used are the same as in the paper, so the solution space available is the same. The flow conditions are $\alpha = 5$, $Re = 525.905$ and $Ma = 0.072$. The objective function is to maximise the C_l.

In the work by R. Mukesh, K. Lingadurai, and U. Selvakumar [18], a Cl of 0.9681 is obtained. With the genetic algorithm of this work, a Cl of 1.087 is achieved, which implies an improvement of 12.87%.



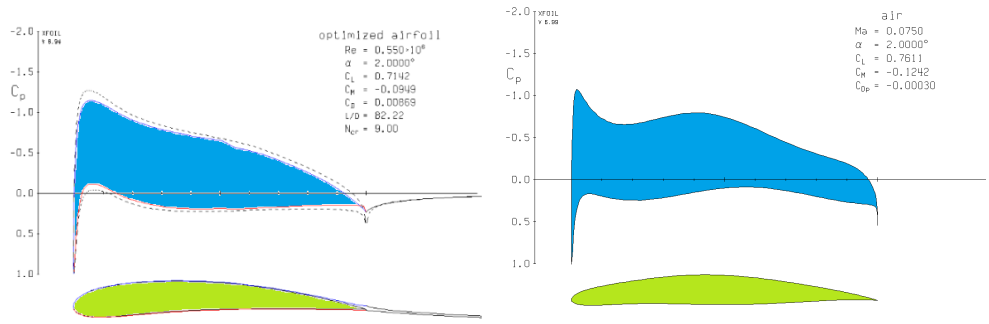
(a) Reference Airfoil. From [18] (b) Obtained Airfoil (c) C_l and C_d

Figure 4.11: Validation 1 - Airfoil Shape Comparison

4.8.2 Comparison with an Evolutionary Algorithm

In this case an evolutionary algorithm is used to optimise a NACA 2412. The flow conditions are $\alpha = 2$, $Re = 550000$ and $Ma = 0.075$ [32]. In addition, a constraint of a maximum thickness of 12% of the chord and a minimum C_m of -0.13 is taken into account.

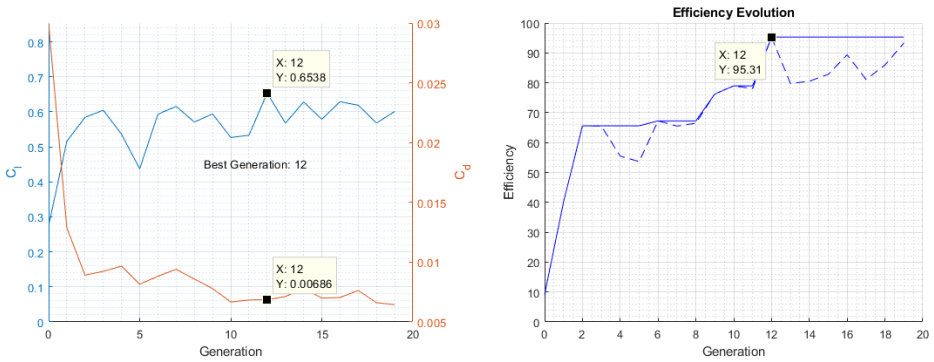
With these conditions, the evolutionary algorithm achieves an efficiency of 82 (C_l=0.7142 and C_d = 0.00869), while the Genetic Algorithm rises up to 96 (C_l=0.6538 and C_d = 0.00686), which is an improvement of a 17.07%. Figure 4.12 shows the airfoils obtained with both methods and the pressure distribution. Figure 4.13 shows the evolution of the Cl, Cd and Efficiency during the generations.



(a) Evolutionary Algorithm. From [32]

(b) Genetic Algorithm

Figure 4.12: Validation 2 - Optimised Airfoils and C_p



(a) C_l and C_d vs. Generations

(b) Efficiency vs. Generations

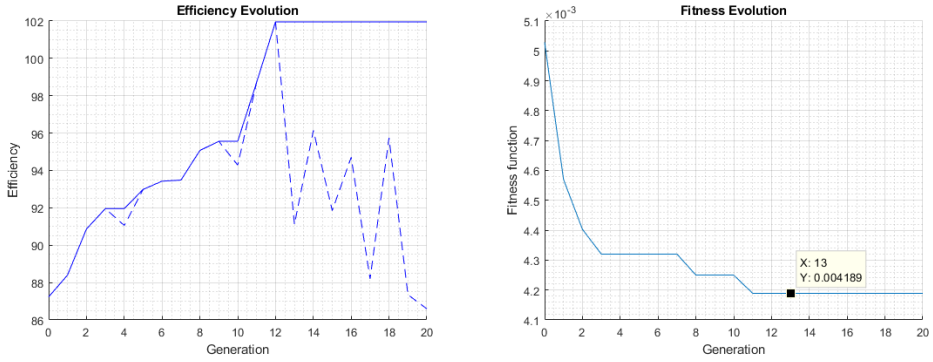
Figure 4.13: Validation 2 - Genetic Algorithm Results

4.8.3 Comparison with Swarm Algorithm with Mutations and Artificial Neural Networks

In this work, a very complex optimisation method using Neural Networks and using PARSEC parametrisation is presented. In the paper, 3 cases of a Gradient-Based Optimisation method starting from 3 different airfoils are shown to compare further results of their investigation. This validation aims to compare the results of the Genetic Algorithm with these three optimisation tests. The objective function studied is the type 2, shown in equation (4.3). The objective is to minimise the drag having an objective $C_l = 0.4$. The PARSEC limit values used are the same as shown in the paper, so the solution space available is the same. The flow conditions are $\alpha = 2$, $Re = 3 \cdot 10^6$ and $Ma = 0.35$ [17].

As said, in the paper 3 different cases are tested starting from three different airfoils: NACA 0012, NLF(1)-0115 and RAE 2822, obtaining fitness values of 0.0025, 0.0045 and 0.00045.

With the Genetic Algorithm and the constraints, a fitness value of 0.00419 is obtained, which does not imply an improvement for cases 1 and 3 (it reaches 59,66% and 10,74% of the final value), but a better result is obtained comparing it with case 2, implying an improvement of a 7.39%. Figure 4.14 shows the results for this test.



(a) Efficiency vs. Generations

(b) Fit vs. Generations

Figure 4.14: Validation 3 - Genetic Algorithm Results

4.8.4 Overview

Table 4.5 shows the results of the three validations. In general, the Genetic Algorithm generates good results, especially when compared with other Genetic or Evolutionary algorithms. However, for the third case, in which it is compared with a very robust optimisation method, it only obtains an improvement for one out of the three cases.

	Objective	**Obtained**	**Improvement**
Comparison 1	C_l = 0.9681	Cl = 1.087	12.87%
Comparison 2	Eff = 82	Eff = 96	17.07%
Comparison 3	fit = 0.0025	fit = 0.00419	-40,34%
	fit = 0.0045		07.39%
	fit = 0.00045		-89.26%

Table 4.5: Results of the validations

4.8.5 From cylinder to an airfoil

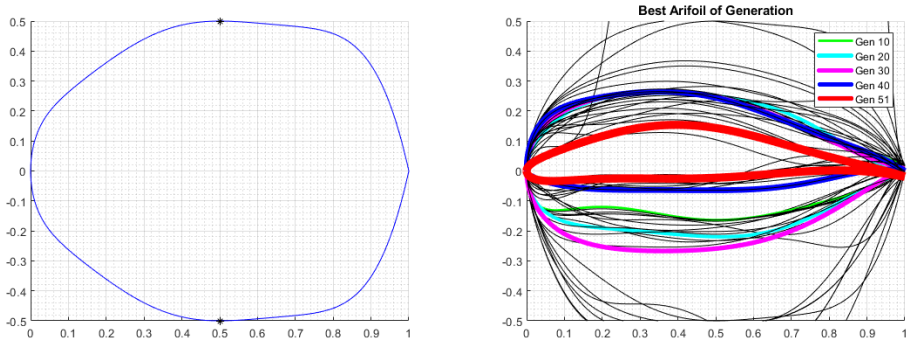
Finally, a test to check the effectiveness of the algorithm and also of the parametrisation has been implemented. The idea is to generate an initial population of only cylinders, all of them equal. Then, via the mutation and the crossover, the population slightly is transformed from the cylinder shape to airfoil-like shapes. In order to do so, a heavy mutation must be included, because at the beginning, the populations is completely infeasible.

A modification to mutation 1 has been implemented. In this modification, only the first 7 bits of the gene can mutate, so it is ensured that the mutations would have a higher impact. The used parameters are shown in table 4.6.

r_{LEup}	r_{LElo}	X_{up}	Z_{up}	X_{lo}	Z_{lo}	Z_{xxup}	Z_{xxlo}	Z_{TE}	ΔZ_{TE}	α_{TE}	β_{TE}
1	1	0.5	0.5	0.5	-0.5	-2	2	0	0	0	160

Table 4.6: Parameters for the initial population of Cylinders

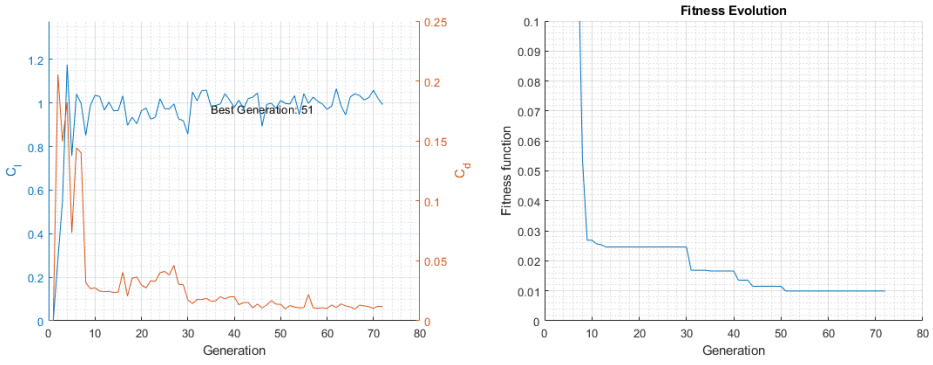
Figure 4.15-a shows the shape generated from values from table 4.6. To do the analysis, fitness function 2 has been set, imposing a C_l=1, and minimising the drag.



(a) Initial Population of Cylinders

(b) Evolution of population

Figure 4.15: Cylinder Population



(a) Cl and Cd vs. Generations

(b) Fit vs. Generations

Figure 4.16: Validation Results

As it can be seen in figure 4.16-a, the Cl of 1 is achieved in generation 10 approximately. The Cd is slowly reduced and the best fitness function is obtained in generation 51, as figure 4.16 shows. In figure 4.15-b, the final shape can be seen coloured in red.

4.8.6 Validation of the Aerodynamic Data

XFOIL has been used to compute the aerodynamic coefficients of the airfoils. XFOIL is an interactive program for the design and analysis of subsonic isolated airfoils. It consists of a collection of menu-driven routines which perform various useful functions such as: viscous (or inviscid) analysis of an existing airfoil, airfoil design and redesign by interactive modification of surface speed distributions, airfoil redesign by interactive modification of geometric parameters, blending of airfoils and plotting of geometry, pressure distributions, and multiple polars.It is written in FORTRAN [4].

XFOIL is widely used in the academic sector due to its fast and reliable results in certain conditions, which are low angles of attack and subsonic flows. Image 4.17 shows a comparison between XFOIL and RFOIL (an XFOIL variation) and real experiments from the book *Theory of Wing Sections* [33]. As seen, the results of XFOIl are very accurate. All in all, even though it does not have the precision of a CDF software, for the required purposes of this project XFOIL is a reliable, easy to use and relatively fast source of aerodynamic data.

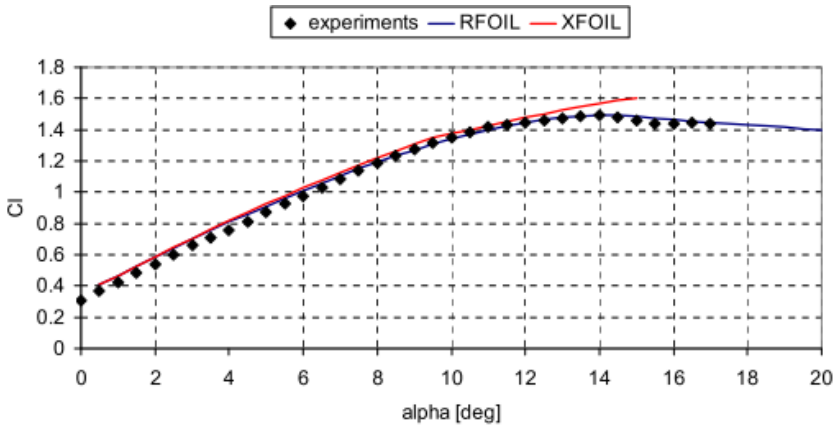


Figure 4.17: XFOIL data compared with experiments data. Image from [3].

Chapter 5

Practical Implementation

5.1 User Guide

In order to execute the code, for now, it is needed a computer with a `gcc` compiler, which is included in the Minimalist GNU for Windows (MinGW) [34]. In addition, the software `XFOIL.exe` must be in the same folder as the executable. To modify the parameters, the `.c` file must be opened. In the first lines of the code, the modifiable parameters can be easily identified. These are the following:

Genetic Algorithm Parameters

- `Generations_NOT_IMPROVE` - Set as 8.
- `MAX_INFEASIBLE` - Set as 9.
- `MAX_GENERATIONS` - Set as 100.
- `MUTATION` - Set as type 1
- `MUTATION_PROB` - Set as 0.7 (70%).
- `individuals` - Set as 160 (more than 200 is not recommended)

Fitness function

- `FITNESS_TYPE` - To choose between 1, 2, 3 and 4. Recomended 1 and 2.
- `ClT` - Objective Cl. Useful only if fitness type 2 is selected.

Constraints

- `MAX_THICKNESS` - To choose between 0 and 1. Set 1 for no constraint.
- `MIN_THICKNESS` - To choose between 0 and 1. Set 0 for no constraint.
- `TE_THICKNESS` - Set 0 for no constraint
- `MIN_TE_ANGLE` - To choose between 1 and 25. Set 1 for no constraint
- `MIN_CM` - To choose between -1 and 0. Set -1 for no constraint
- `EFF_LIM` - Sometimes an error occurs in XFOIL execution and it returns huge values of Eff (because it computes very low values of Cd). To avoid this error, an Efficiency limit related to the problem dimension is set. If the computed efficiency overcomes this value, the fit is set as infinite and the airfoil is not considered. It is recommended to set it as 500 or so, and then modify it after a first execution, taking into account the efficiency values computed.

The execution generates several files, including the best airfoil of each generation (`.dat` files) and a `.txt` file containing the aerodynamic coefficients and fit of the best airfoil of each generation. These files can be analysed with a MATLAB code. Thus, it is needed to have the software installed to use it.

All in all, the needed files to execute the code and analyse the results are:

- `GA_airfoil_Camps.c`
- `GA_airfoil_Camps.exe`
- `kill_XFOIL.c`
- `kill_XFOIL.exe`
- `XFOIL.exe`
- `plot_airfoils.m`

The MATLAB file generates a plot in which all the airfoils and its evolution can be seen in the Appendices. It also shows the C_l and C_d evolution, the C_m evolution, the Efficiency evolution and the fitness evolution.

5.2 Results

In this section the code is tested with some real examples to demonstrate its effectiveness in the design process. The idea is to execute the code to find an optimum airfoil for a very specific purpose and compare the results with previous real designs.

The studied cases are real designs for the Air Cargo Challenge and the Paper Air Challenge.

5.2.1 Air Cargo Challenge (heavy lifter)

The Air Cargo Challenge is an aeronautical competition which takes place in an European city every two years:

> This competition was held for the first time in 2003 and it was founded by a group of Aerospace students in Lisbon. The competition is primarily directed to aeronautical and aerospace engineering students, similarly to the north-American Design/Build/Fly.
>
> The main objective is to design and build a radio-controlled aircraft that is able to fly with the highest possible payload according with the rules established in the competition regulations, which vary in each edition. The team's score is not only given by the performance demonstrated in the flight competition part, but also by the technical quality of the project, through the evaluation of the design report and drawings.
>
> The event´s first edition (ACC'03) was organized by the APAE: Associação Portuguesa de Aeronáutica e Espaço (Portuguese Association of Aeronautics and Space), an aerospace group from Instituto Superior Técnico. From the ACC'07 onwards, the competition grew to an international level under the umbrella of EUROAVIA, the European Association of Aerospace Students, and the winning team got the possibility of organizing the next edition. The ACC of 2011 was held in the University of Stuttgart, August 2011, organized by the AKAModell Stuttgart together with the EUROAVIA Stuttgart.[1] The Universidade da Beira Interior was the winner of this edition, thus it took place in Portugal. Again, the Team from Stuttgart won this Edition, and got the organization responsibility. The last edition of ACC took place in 2017 in Zagreb. The last competition started end of 2018. The competition took place in Stuttgart 12.-17.August 2019. The next competition will be held in Munich in the summer of 2021. [35].

In this work it is studied the airfoil that UPC Venturi used in its aircraft (V-17) in the Air Cargo Challenge 2017, Zagreb. With this aircraft UPC Venturi reached the 10th position out of 28 teams, which is the best position of an Spanish team in the competition [36,37].

In the 2017 edition, the main objective was to create an aircraft capable of lifting high payloads and, at the same time, be fast enough to complete the air circuit with the less amount of time possible. The objective of UPC Venturi was to lift up a payload of 13.8 kg with an OEW of 5.2 Kg, making an MTOW of 13.8 kg. The result was an aircraft with a wingspan of 3.812 m, and a mean chord of 36 cm [38].

The airfoil used in most of the V17's wing was a modified Selig 1223. This airfoil was also used by many other teams in the competition, due to its high performance in terms of lift even though it also generates lots of drag. The regulations of that year made this airfoil very competitive. The V17 can be seen in figure 5.1.

Figure 5.1: V17 flying in the Air Cargo Challenge 2017, Stuttgart

As said before, the inputs required by the code are the angle of attack, the Reynolds number and the Mach number. In order to obtain the Reynolds number, it is needed to know the mean aerodynamic chord ($\hat{c} = 0.36m$), the speed of flight, which for the V17 in the cruise was approximately 65 km/h ($v = 18,05m/s$), the air density, which will be assumed as $\rho = 1.225kg/m^3$ and the air viscosity, which is taken as $\mu = 1.74 \cdot 10^{-5} N \cdot s/m^{-2}$, then:

$$Re = \frac{\rho v c}{\mu} = \frac{1.225 \cdot 18.05 \cdot 0.36}{1.74 \cdot 10^{-5}} = 457474.13 \tag{5.1}$$

To compute the Mach, the sound speed is needed, and thus, the heat capacity ratio of the air ($\gamma = 1.4$), the specific gas constant ($R = 287J \cdot Kg^{-1}K^{-1}$) and the temperature, assumed as 20ºC ($T = 298K$).

$$a = \sqrt{\gamma RT} = \sqrt{1.4 \cdot 278 \cdot 298} = 346,029m/s \tag{5.2}$$

Finally, the Mach is computed as follows:

$$M = \frac{v}{a} = \frac{18.05}{346.029} = 0.05 \tag{5.3}$$

With these conditions, a Selig 1223 is analysed using XFLR5, which is a software that uses XFOIL to compute aerodynamic data, is more user friendly and allows to plot the results easily.

Then, the inputs are set in the code, and the first optimisation is performed with these ones:

- $\alpha = 2^{o}$
- Re = 457474.13
- Ma = 0.05
- Fitness type: 1 (Efficiency)
- Constraints: No

The results are shown in figure 5.2, where the blue data corresponds to the Selig 1223 and the red data to the airfoil generated with the genetic algorithm. As can be seen, the efficiency (lower right) is extremely larger in the angle of attack of 0^{o}, which is the theoretical angle of attack on cruise. Thus, for an ideal case in which the wing is static in this angle of attack, results would be very satisfactory.

Unfortunately, in the reality the aircraft has to climb with an angle of attack between 3 and 5º, and do some turns in similar angles. In these cases, the performance of the aircraft would be significantly poor. The figure shows efficiency losses after $\alpha = 3$, which is very critical for the take-off phase. In addition, looking at the $C_l - \alpha$ plot (upper left) the Cl also drops with these angles of attack and offers worst results than the Selig 1223. As said, these are bad results since the Cl should be very high in $\alpha = 3$ because it is the design angle of attack for the take-off [38].

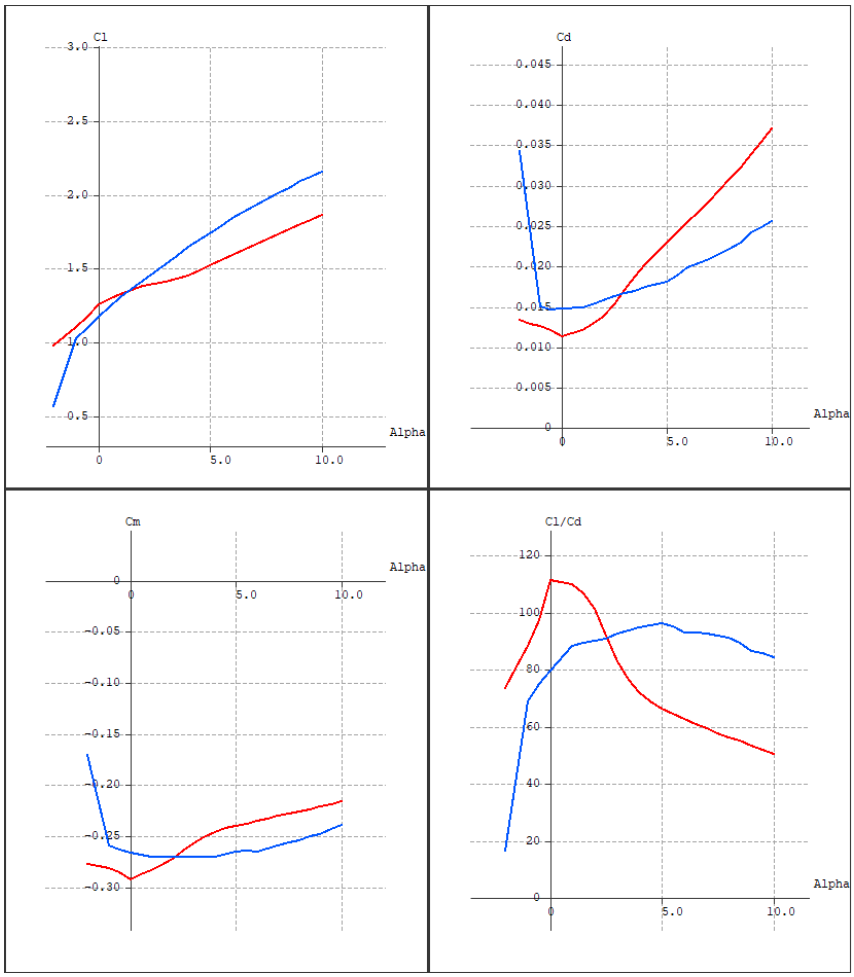


Figure 5.2: Selig 1223 results compared with first optimisation

A second optimisation is performed, in this case with fitness function 2. It is wanted to ensure that the Cl and Eff are high in a wider range of angles of attack, so the inputs are the following:

- $\alpha = 5º$
- Re = 457474.13
- Ma = 0.05
- Fitness type: 2 (Objective $C_l = 1.8$. Original C_l in S1223 at 5º: 1.73)
- Constraints: No

In this case, better results in terms of Cl and Efficiency are obtained, as figure 5.3 shows (colour green). The Cl (upper left) is better for a wide range of α's, but decreases drastically after $\alpha = 8$. The efficiency is high and quite constant in the same range of α's, which is good since the aircraft is not always flying in a specific angle of attack.

On the other hand, the Cm is quite lower than the original. A low Cm value implies high control surfaces, such as the tail, to stabilise the aircraft in the desired design angle of attack, which usually corresponds to the cruise angle of attack (0-1º). Thus, another optimisation is performed applying a constraint of a minimum Cm=-0.28 (the same as S1223) to check how the constraint works.

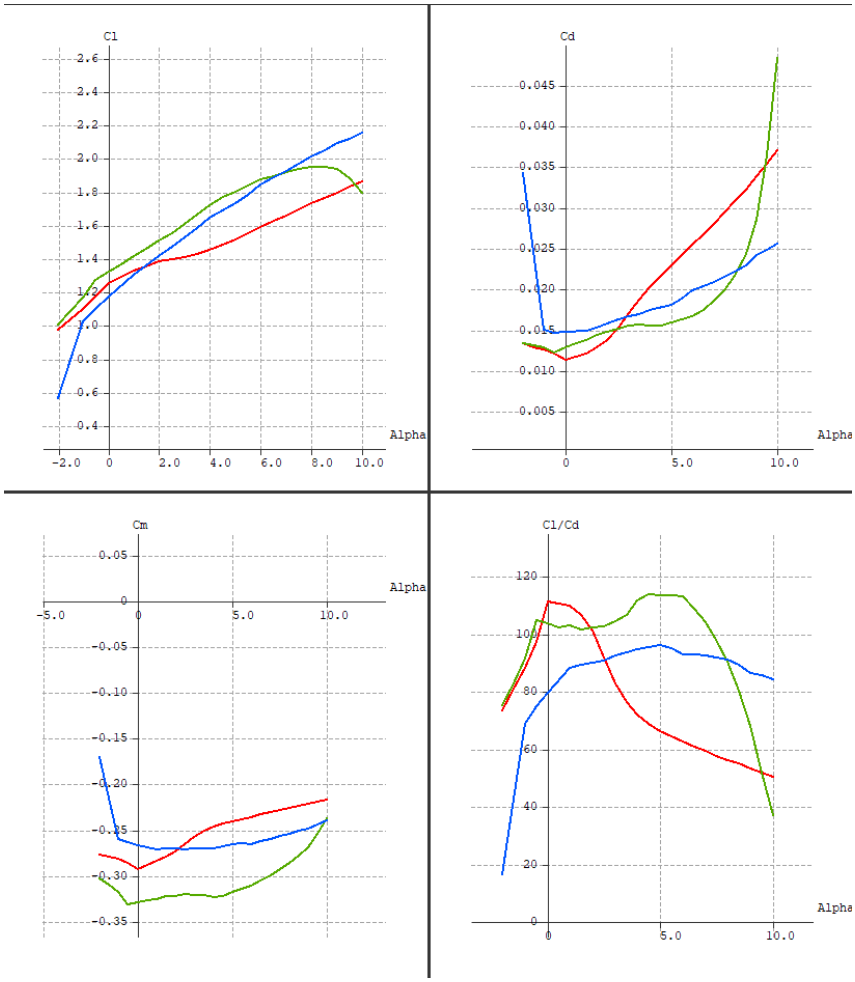


Figure 5.3: Selig 1223 results compared with second optimisation

In addition to the Cm limit, the angle of attack to optimise is set as 3 to try to avoid the loss of performance after $alpha = 8$ º. The new inputs are:

- $\alpha = 3^{\text{o}}$
- Re = 457474.13
- Ma = 0.05
- Fitness type: 2 (Objective $C_l = 1.58$. Original C_l in S1223 at 3º: 1.525)
- Constraints: $C_m > -0.28$

The results of this optimisation, with the Cm constraint can be seen in figure 5.4 with colour black. The first thing to notice from the figure is that the Cm constraint works well since the Cm is not below -0.28 (lower left) ub the design angle of attack. Regarding the efficiency

and the Cl, the results are better than the original and, even though the efficiency or the Cl are below the results of previous iterations in low angles of attack, for high angles of attack, the loss of performance is avoided. All in all, figure 5.4 shows good and solid results for the desired range of αs.

As said before, the efficiency and Cl obtained with optimisation 1 in $\alpha = 0$ are spectacular, however, it would be wrong to choose that airfoil because the aircraft also has to climb or turn, even though it is true that most of the time will be flying between 0 and 2 degrees of angle of attack. However, it is important to notice that these are results of a 2D analysis, which would be equivalent to a 3D analysis with a finite wing of infinite wingspan. When the airfoil is used in a wing, other aspects like wing tip vortices and other interactions related to the wing geometry would make these results to differ.

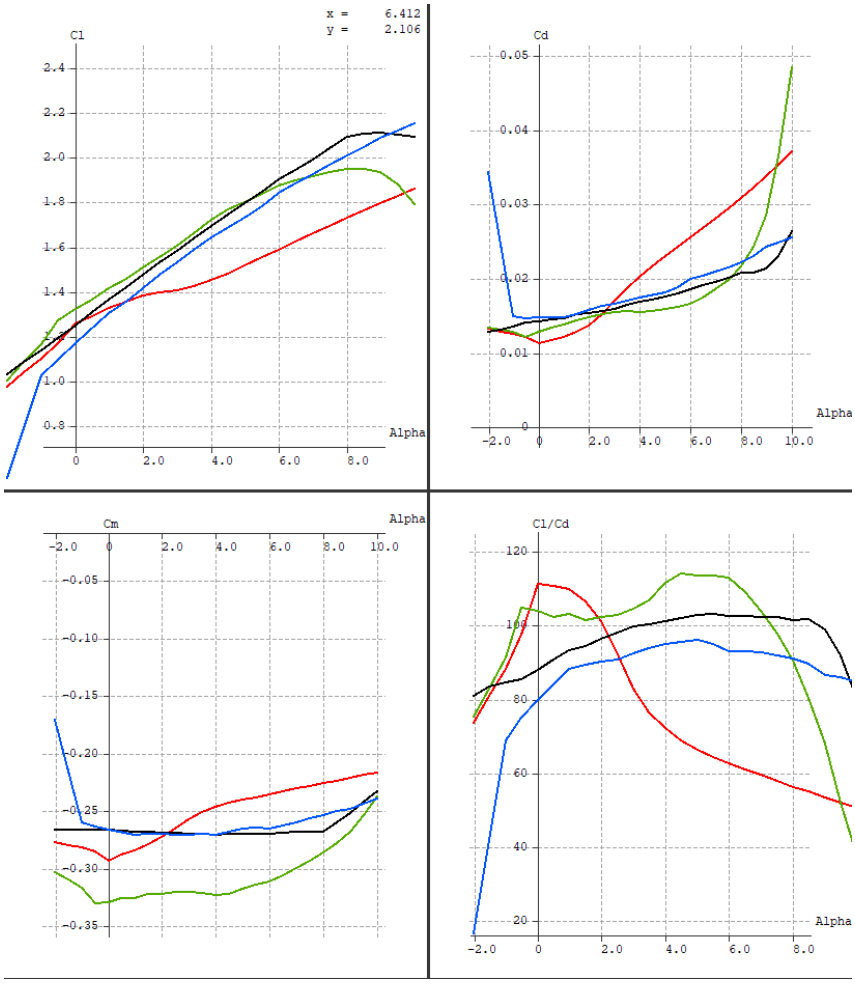
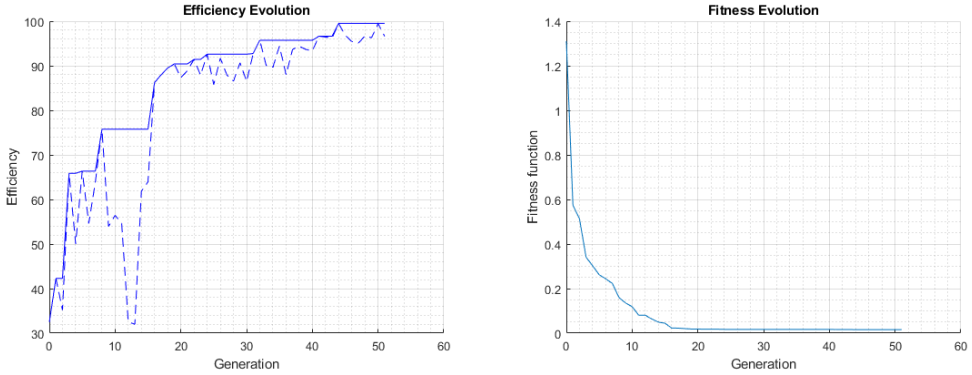


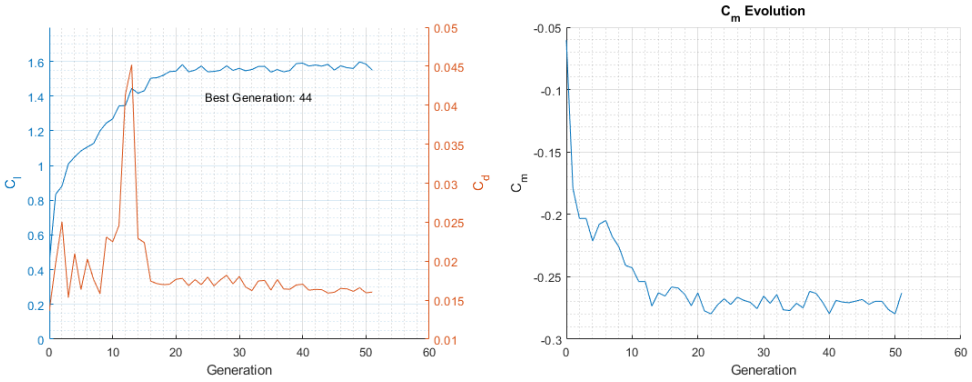
Figure 5.4: Selig 1223 results compared with third optimisation

The results of the genetic algorithm process for the last airfoil can be seen in the following images. In figure 5.5-a, the evolution of Efficiency through the generations can be seen. The efficiency grows with steps after some generations. In the beginning, the growth is very fast, and it coincides with the growth of the C_l. As a remember, the fitness function is to minimise the expression $(C_l - C_l^T)^2 + Cd$, thus, the algorithm wants to reduce the difference in the C_l, because it is squared. The C_l^T is fixed as 1.58, and as it can be seen in figure 5.6-a, the C_l quickly reaches that point. Then, it tries to minimise the C_d, and this is more complicated and slow and is why the efficiency grows in a slower rhythm after generation 20. Figure 5.5-b shows the fitness value, which again, drops very fast at the beginning and then is reduced slowly. Regarding the C_m, which can be seen in figure 5.6-b, it is seen that it is never below the constraint of -0.28.



(a) Efficiency vs. Generations (b) Fitness vs. Generations

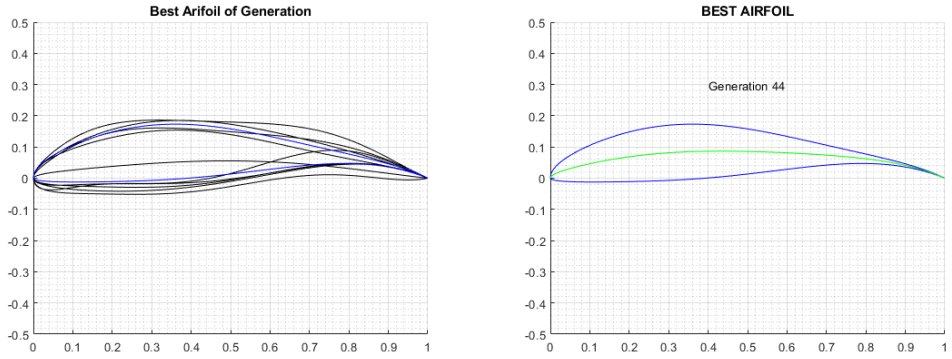
Figure 5.5: ACC - Fitness results of optimisation 3



(a) Cl and Cd vs. Generations (b) Cm vs. Generations

Figure 5.6: ACC - Aerodynamic coefficients of optimisation 3

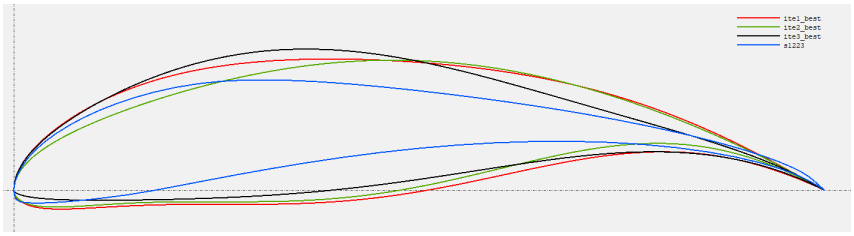
The airfoils generated in all this process can be seen in the figures below. First, figure 5.7-a shows some of the airfoils from the last optimisation process. On it, airfoils from generations 0, 10, 20, 30, 40 and 44 can be seen. Figure 5.7-b shows the final airfoil created with the genetic algorithm with its chamber line. Figure 5.8-a shows the airfoils from the 3 iterations with its respective colors and the Selig 1223, in blue. On the other hand, figure 5.8-b shows only the initial and the final airfoil. It is interesting to see that both the leading edge and the trailing edge are very similar. However, they are quite different in the intermediate part, in terms of thickness.



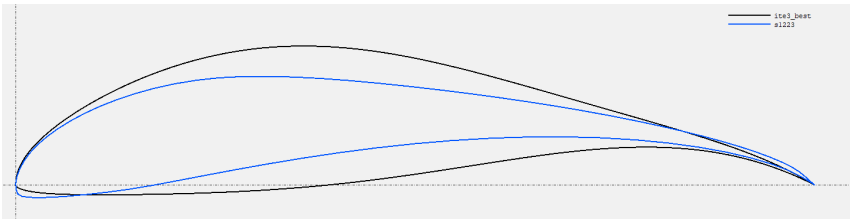
(a) Airfoils from generations 0, 10, 20, 30 and 40

(b) Final Airfoil. Generation 44

Figure 5.7: ACC - Airfoils from optimisation 3



(a) All Airfoils



(b) Orginal airfoil vs. GA final airfoil

Figure 5.8: ACC - Airfoil comparison

All in all, these two airfoils have the exact same C_m and a very similar Efficiency curve, despite the fact that the one from the airfoil created with the GA is higher. This similar behavior in the C_m is because the chamber line displacement is very similar in both airfoils: 8.66% in the GA airfoil and 8.67% in the S1223. The chamber line curvature is directly related to the momentum of the airfoil. The thickness of the created airfoil is 17.76%, while the one from the S1223 is 12.13%.

It can be considered that the result of the optimisation process via the genetic algorithm has been a success since a great airfoil has been generated from nothing but taking as a reference a similar one. It is true that without the Seilig 1223 reference, the process may have been slower, but if the designer has the problem well defined, with two or three executions of the program, a good airfoil can be found. Nevertheless, it must be said that maybe the aircraft analysed with the created airfoil behaves worse than with the original airfoil, because, as said before, in an aircraft several variables affect the aerodynamics. In that case, an iterative process should be considered, in order to better adapt the airfoil to the geometric characteristics of the aircraft.

5.2.2 Paper Air Challenge (glider)

The Air Paper Challenge is a competition created by Trencalòs Team [39] that takes place twice a year in The School of Industrial, Aerospace and Audiovisual Engineering of Terrassa, ESEIAAT UPC [40].

The objective of the competition is to design and build a plane made out of paper and carton, which will be launched from a platform that impulses it. The plane has to glide the maximum amount of distance from that platform. The planes usually have between 1 and 2.5 meters of wingspan.

Since the objective of the aircraft is to have a maximum range, it is needed to know how to achieve it. The forces that affect a glider are the weight (W), the lift (L) and the drag (D). The equations of motion in the vertical and horizontal axes can be represented as:

$$-D + Wsin(\gamma_d) = 0 \tag{5.4}$$

$$Wcos(\gamma_d) - L = 0 \tag{5.5}$$

Where equation (5.4) is for the x-axis and (5.5) for the y-axis, and γ_d is the angle of descend.

The equations can be modified to obtain:

$$D = Wsin(\gamma_d) \tag{5.6}$$

$$L = Wcos(\gamma_d) \tag{5.7}$$

And finally, dividing them:

$$tan(\gamma_d) = \frac{D}{L} = \frac{1}{E_{ff}} \tag{5.8}$$

Thus, the glide angle is computed as the inverse of the aerodynamic efficiency of the whole plane. When a plane flights with maximum efficiency (E_{max}), the angle of descent is minimum and, as a consequence, the range of the glider maximum [41].

To sum up, it is needed to design a plane that stabilises in a given angle of attack, usually 0-2 degrees, and achieve a maximum flight efficiency in that angle. The results of the optimisation process will be compared with a plane that participated in the competition in 2015, achieving the third position. The plane used a Clark-YM18 airfoil due to structural reasons: other airfoils offered better efficiencies but its thickness was too small. Clark-YM18 has a thickness of 18%, enough for the main beam. The mean aerodynamic chord of the plane was 0.275m, the design speed of flight was 6.5m/s and the design stable angle was approximately $2^{\underline{o}}$. [42].

With these data, the Reynolds and Mach number can be easily computed:

$$Re = \frac{\rho v c}{\mu} = \frac{1.225 \cdot 6.5 \cdot 0.275}{1.74 \cdot 10^{-5}} = 125844,1 \sim 126000 \tag{5.9}$$

$$a = \sqrt{\gamma RT} = \sqrt{1.4 \cdot 278 \cdot 298} = 346,029 m/s \tag{5.10}$$

$$M = \frac{v}{a} = \frac{6.5}{346.029} = 0.0188 \tag{5.11}$$

To generate the airfoil with the genetic algorithm the following inputs and constraints are used:

- $\alpha = 2$
- Re = 160000
- Ma = 0.0188
- fitness type: 1 (Efficiency)

- Constraints:
  - Minimum thickness: 17%
  - Minimum trailing-edge angle: 6º

Where the constraints are used for structural reasons (to ensure that a theoretical beam would fit in the airfoil).

The analysis is performed and the results can be seen in figure 5.9. The goal is to optimise the efficiency of the Clarky-M18 (black line), and the results of the optimisation (blue lines) are considerably better, especially for the angle of attack of 2º. The original Cl at 2º is 0.75, while the new Cl is 1.11, which means an increase of the 48%. On the other hand, the Cd also increases from 0.0175 to 0.0215, which is an increase of 22% and is not good. However, the general combination ends up with an increase in the Efficiency from 43 to 52 (21%).

The thickness of the final airfoil is 17.43% and it could be considered as a good result. Nevertheless, the obtained Cm is quite low, so another optimisation is done including a Cm constraint.

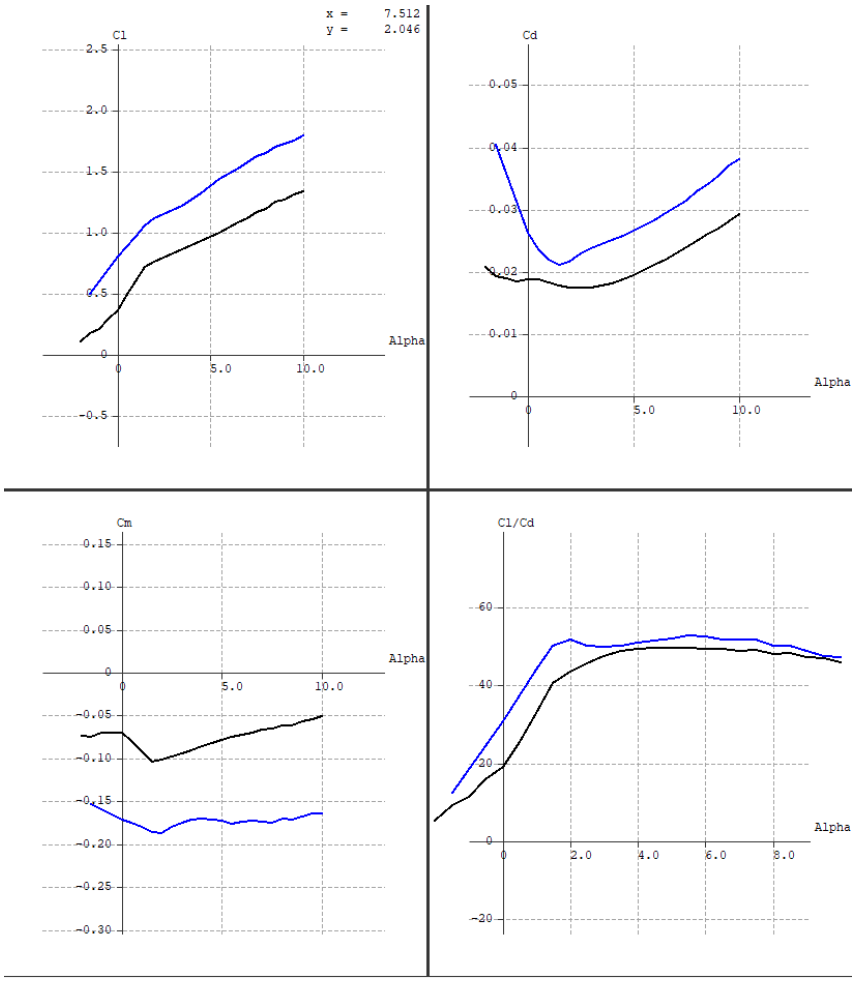


Figure 5.9: ClarkY-M18 results compared with first optimisation

The execution is the same as before but in this case it includes a minimum Cm of -0.12. It is expected a less cambered airfoil and, as a consequence, the lift provided by the airfoil would probably be lower than with the previous optimisation.

- $\alpha = 2$
- Re = 160000
- Ma = 0.0188
- fitness type: 1 (Efficiency)
- Constraints:
  - Minimum thickness: 17%
  - Minimum trailing-edge angle: 6^{o}
  - Minimum C_m: -0.12

The results are shown in figure 5.10 colored in pink. As expected, since the Cm must be lower, the C_l is limited because the airfoil must be less chambered. On the other hand, in this case, the C_d is also very low for the design angle of attack. The improvement on the efficiency in $\alpha = 2$ is a 09,65%, from 43 to 47,15. The fact of limiting the C_m and the thickness make the solution space smaller and as a consequence, the solution found is not as optimum as the first one.

All in all, the solution found is considered to be right, since the constraints are completely fulfilled and the results are good enough. In addition, the efficiency of the airfoil in other angles of attack has good behavior, since it is maintained high and does not drop drastically.

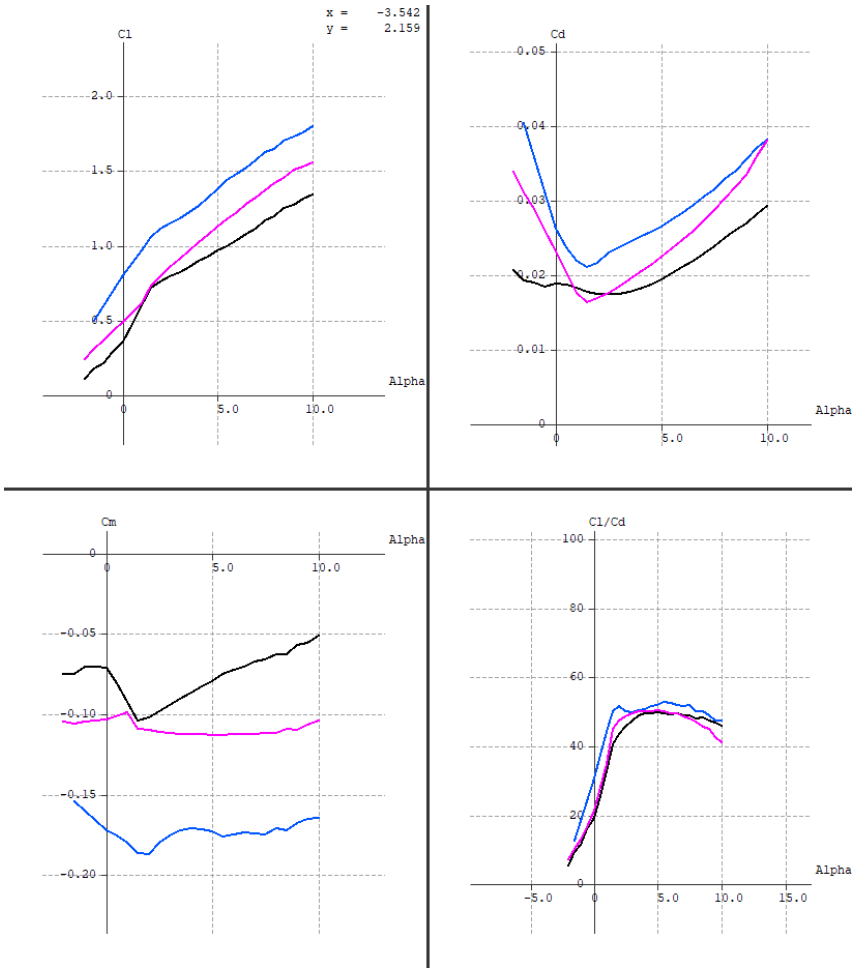
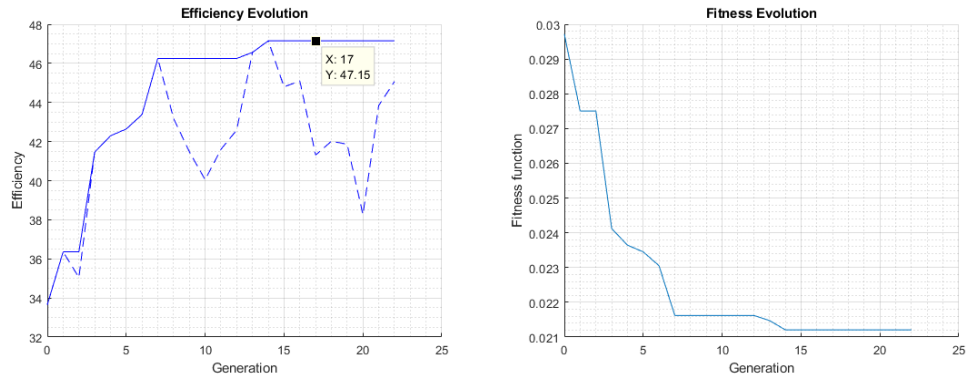


Figure 5.10: ClarkY-M18 results compared with first and second optimisation

In the figures below, the results of the genetic algorithm for achieving the previous results are shown. Figure 5.11 shows both the evolution of the efficiency and the fitness, which in this case is the exact inverse, since the fitness type selected was 1 (Maximise efficiency). As in the previous case, the efficiency grows up quickly at the beginning and then struggles to increase more.

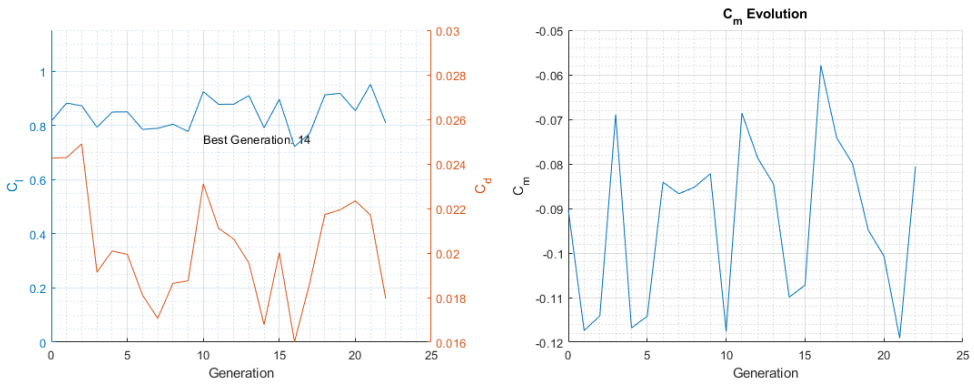
Looking at figure 5.11, it can be seen that the C_l encountered is approximately constant, and the algorithm tries to reduce the C_d. Then, after some bad generations between generations 7 and 10, the C_d drops again and the best generation is achieved in generation 14. After generation 14, 9 worst generations are found and the execution is terminated. In this case, a low number of generations have been studied. Regarding the C_m, it oscillates in values between -0.12 and -0.06, achieving the best generation with a low value of C_m, (-0.11), since with a low C_m a more chambered airfoil can be found and thus, generate more lift.



(a) Efficiency vs. Generations

(b) Fitness vs. Generations

Figure 5.11: PAC - Fitness results of optimisation 2

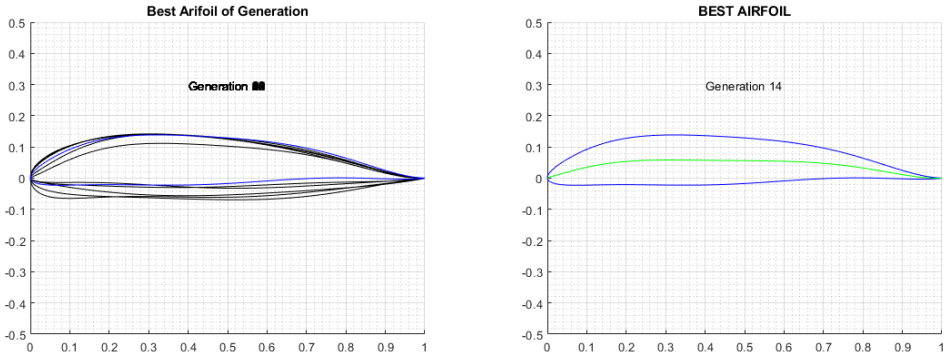


(a) Cl and Cd vs. Generations

(b) Cm vs. Generations

Figure 5.12: PAC - Aerodynamic coefficients of optimisation 2

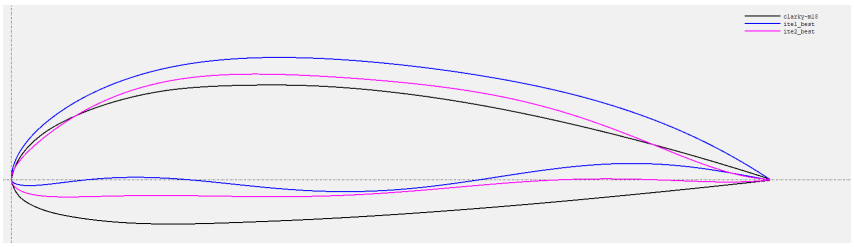
Figure 5.13-a show the airfoils for generation 0, 3, 7, 9, 12 and 14. It can be seen that the thickness of the final airfoil is approximately the same for the previous generations, but the airfoils are less chambered, being the one from generation 14 the more chambered airfoil. Figure 5.13-b shows only the final airfoil with the chamber line. In figure 5.14, the airfoils from optimisation 1 and 2 are compared with ClarkY-M18. It can be appreciated that the curvature of the airfoil from optimisation 1 (blue) is much more chambered, generating a higher momentum. The final airfoil is less chambered, the thickness is similar to the ClarkY-M18 (-02%) and the leading edge radius is also very similar.



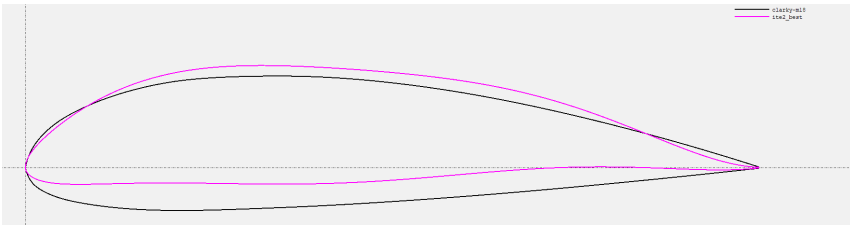
(a) Airfoils from generations 0, 10, 20, 30 and 40

(b) Final Airfoil. Generation 44

Figure 5.13: PAC - Airfoils from optimisation 2



(a) All Airfoils



(b) Orginal airfoil vs. GA airfoil

Figure 5.14: PAC - Airfoil comparison

5.2.3 Overview

In the following tables, the final results of the optimisation processes can be seen. In the columns there are the airfoils and the relative improvement respect the original airfoil. Regarding the Cd and Cm, an improvement (+, green) is a reduction respect to the original.

	Selig 1223	**Opt. 1 (2º)**	*Rel. [%]*	**Opt.2 (5º)**	*Rel. [%]*	**Opt. 3 (3º)**	*Rel. [%]*
Thickness [%]	12.13	17.67	*+05.54*	16.65	*+04.52*	17.76	*+05.63*
Chamber [%]	8.67	8.32	*-00.35*	8.87	*+00.20*	8.66	*-00.01*
Cl	2º - 1.41 5º - 1.73 3º - 1.53	1.38	*+02.12*	1.8	*+04.05*	1.58	*+03.27*
Cd	2º - 0.0155 5º - 0.0180 3º - 0.0165	0.0135	*+12.90*	0.016	*+11.11*	0.016	*+03.30*
Min Cm	-0.27	-0.29	*-07.41*	-0.33	*-22.22*	-0.27	*00.00*
Eff	2º - 90 5º - 96 3º - 92	101	*+11.22*	113	*+11.77*	100	*+10.87*

Table 5.1: Results of Air Cargo Challenge Optimisation

	ClarkY-M18 (2º)	**Opt. 1 (2º)**	*Rel. [%]*	**Opt.2 (5º)**	*Rel. [%]*
Thickness [%]	18	17.43	*-00.57*	16.09	*-01.91*
Chamber [%]	3.56	7.65	*+4.09*	05.82	*+2.26*
Cl	0.75	1.1	*+46.66*	0.8	*+06.66*
Cd	0.0175	0.0215	*-22.85*	0.0168	*+04.00*
Min Cm	-0.105	-0.188	*-79.04*	-0.113	*-07.62*
Eff	43.5	51.5	*+18.39*	47.3	*+08.73*

Table 5.2: Results of Paper Air Challenge Optimisation

In both tables 5.1 and 5.2, a similar phenomenon happens. Looking only at C_l or Efficiency, the best optimisation is not the final one, and this is due to the constraints in C_m or in geometry.

Chapter 6

Conclusions

In this thesis a genetic algorithm applied to airfoil optimisation has been developed from scratch. In order to do so, several steps have been required.

First of all, it has been needed to correctly identify the problem to understand what was expected from the algorithm. This is, what kind of inputs the algorithm is going to need, and what outputs are expected. Regarding the latter, the algorithm has been planned as a tool for aerospace engineering students or the aeromodelling community that can help them during their aircraft design processes. With this objective in mind, all this work is uploaded in the GitHub repository https://github.com/perecamps/Airfoil_Optimisation_GA.

The basics of a simple aircraft design process have been explained and, with it, it has been demonstrated that the existence of software capable of finding an optimum airfoil for specific flow conditions would be extremely helpful. The amount of time needed and the complexity of the process would be considerably reduced. With all, the main idea of this thesis have been to develop an algorithm that finds an optimum airfoil for an specific angle of attack, Reynolds number and Mach number, including some geometric or aerodynamic constraints.

This work includes a brief introduction to airfoil geometry and the basic concepts of aerodynamic forces and coefficients, so the non-familiarised reader can understand the development of the thesis. In addition, there is a full chapter dedicated to the theoretical explanation of this specific genetic algorithm. This means that this work does not aim to be a complete guide of the genetic algorithm but contains the necessary information to understand the process followed to reach the objectives.

In Chapter 3, after a brief introduction to optimisation methods, the basic concepts of a Genetic Algorithm are explained. A Genetic Algorithm is a stochastic method because it incorporates random processes in the search of the solution. It is inspired by Darwin's theory of Evolution because it takes into account the concepts of Natural Selection, Reproduction and Mutation. The main idea is to generate an initial population of individuals, being the individuals the object to optimise, in this case, an airfoil. Thus, an initial population of airfoils is generated and their fit or adaptability to the system is computed. In the algorithm, the fit is the characteristic of the individual that wants to be optimised, and usually is a function to minimise. The individuals with a better fit will have more chances to be chosen for reproduction and transfer its characteristics to further generations. In the reproduction, mutations are also included, and it is necessary to understand that they help to explore other space solutions and avoid encountering a local minimum. With the new generation created the fitness is again computed, and the process continues until a solution is found or a stopping condition is met.

In the developed Genetic Algorithm, the airfoil has been parametrised using the PARSEC parametrisation. With it, the airfoil can be fully defined with 12 parameters with a physical meaning. These 12 parameters are transformed with an encoding to binary numbers to from the chromosome of the individual. The chromosome is the genetic information of the indi-

vidual and it is composed of 12 genes, which are the PARSEC parameters encoded. To do the encoding, the data type selected has been an `unsigned short`, which has a storage size of 16 bits. All in all, a chromosome of a single airfoil has 12 genes of 16 bits each. Regarding the crossover, the One-Point-Crossover method has been selected and it is fully explained in Chapter 3. In the Crossover, three types of Mutations have been considered: 1-bit mutation, 2-bit mutation and Heavy Mutation. The selected individuals that reproduce and create the new generation are chosen with a random process: the ones with a better fit to the system have more chances to be chosen, but an individual with a low fit can also have the opportunity to transfer its genome. The fit can be computed in several ways, in this case, with an airfoil, the function to minimise can be the inverse of the airfoil Efficiency or the difference between an objective Cl and the real one. These possibilities are fully explored in section 4.6.

Chapter 4 is focused on the implementation of a C code of the concepts explained in Chapter 3. On it, the diverse possibilities are explored and the best solutions implemented in the final code. In addition, the structure of the whole code can be seen. The main idea of the Genetic Algorithm is to create randomly an initial population. Then, the fitness of each individual of the population is computed and stored. The execution enters into a *while* loop until the solution is found or until one of the stopping conditions is met. In this while, the selection of individuals for the reproduction is performed, and also the crossover and mutation. Once the new complete generation is created, its fitness is again evaluated and the best individual saved. If none of the stopping conditions are met, the reproduction starts again. The stopping conditions of this algorithm can be to reach a maximum number of generations or not improving during certain generations in a row. When this happens, it is considered that the problem has been solved. The functions used in the algorithm are also explained in Chapter 4. They are divided into Errors and User information functions, Random Numbers, Airfoil Parametrisation and Aerodynamic Computation, Genetic Algorithm functions and functions dedicated to solve a System of Equations.

Once the Genetic Algorithm is completed, it is still needed to do an study to choose the parameters that affect the execution of the algorithm, which are the size of the initial population, the type of mutation, the mutation probability, the maximum numbers of generations allowed... A complete study has been developed in which several executions have been performed to analyse the results of the algorithm combining different values of these parameters. It has been demonstrated that huge population sizes where not effective because the time consumption was enormous and the improvements respect to smaller sizes where insignificant. It also has been seen that the ideal mutation type is 1-bit mutation and the ideal mutation probability is a 70%. This probability is a big number, and the explanation of why this mutation probability offers great results is because the encoding generates a 16 bit number for each of the PARSEC parameters, which are small decimal numbers. Thus, a 1-bit mutation on one of these 16 bit can be really insignificant in the final result, because a modification in the right hand side of the bit chain implies very small changes. All in all, the combination of the characteristics of this problem and 1-bit mutation type has ended up being the most effective and efficient when solving the problem of optimisation and the demonstration is fully explained in Chapter 4.

Airfoils have some characteristics that define them, such as their thickness, their Efficiency, their lift, their drag... Some of these characteristics are more important than others in function of the type of aircraft in which the airfoil will be used. For instance, in gliders, Efficiency is an extremely important parameter, since their range is directly related to it. This is why the code has incorporated the possibility to choose the fitness function. It is possible to maximise Efficiency, to minimise the Cd with an objective Cl, to maximise the Cl or to minimise the Cd. In this way, the designer can choose and create an airfoil that better fits the aircraft requirements. Related to this, the aerodynamic coefficients have been computed using external software, XFOIL, which has been demonstrated to be very useful for the purpose of this work, yet the time consumption is not as optimum as desirable. The fact of calling an external executable from a C file and copy the results from a `.txt` takes more time than the calculus itself. In addition, the use of random processes makes that sometimes, the airfoil shapes generated are impossible to process by XFOIL and it gets

stuck. In order to solve this problem, parallel execution of a code that checks if XFOIL is functioning well has been developed. Its functioning is fully explained in Chapter 4. Despite these problems, XFOIL is a very solid software that fits well enough in the structure of this project and the results offered by it are satisfactory.

The code has been validated by comparing its results with similar algorithms found in papers. Specifically, in section 4.8, the results have been compared with a similar Genetic Algorithm, with an Evolutionary Algorithm and with a Swarm Algorithm with Mutations and Artificial Neural Networks. The results have been better in 3 out of 5 tests found in these papers, although the 2 fails are related with the Swarm Algorithm, which is very complex and significantly different from this work. In addition, a common test in airfoil optimisation algorithms has been performed. The test consists in generate an initial population of cylinders and achieve that the evolution of the generations eventually reaches an airfoil shape. The test has been completed with success, generating the best airfoil in generation 51. Finally, the obtention of the aerodynamic data with XFOIL has also been validated with literature.

With the validation done, the final parameters have been set and the code finished. The code has been tested comparing its results with two real study cases, the Air Cargo Challenge and the Paper Air Challenge. In both of them, more than one optimisation has been performed and the results have been satisfactory, seeing an improvement in the final cases. In both cases, the best airfoils in terms of C_l or Efficiency has not been the selected one, since they did not meet the constraints, which are crucial in these types of problems. A detailed explanation can be seen in Chapter 5.

However, some issues have been detected during some tests with all the optimisation processes performed during the work. First, it has been seen that when applying an upper and lower thickness constraint at the same time, some strange behaviours are seen. It is believed that this is because at the beginning, very few random airfoil fit in the thickness allowed and they are very weird in shape and, thus, the results are very negative. A solution could be to create a less restrictive constraint that does not completely exclude the airfoil who not fit in it. In addition, two equal simulations, in some particular cases they offer significantly different results. It is believed that it is due to a weird combination of random parameters. A final issue is that sometimes XFOIL encounters fake results of C_d, which affect the final lecture of the solution. These problems could be solved using better software such a CDF, but it would mean to sacrifice more time.

In general, I am very satisfied with the obtained results and the topics learned. Although a single optimisation may not be enough for defining a perfect airfoil, it certainly generates a good airfoil that behalves optimally in the desired flow conditions, from which it is possible to work and define an even better airfoil. The objective has been well accomplished and it only remains to be able to share this work with the community, and improve it with some reviews and commentaries of the users. This is a long project, with much to do, and I will personally try to make it possible.

Bibliography

[1] S. Franchini and López, *Introducción a la Ingeniería Aeroespacial*, 2nd ed. Madrid: Garceta, 2012.

[2] S. Summit, "Bitwise Operators," pp. 15–17. [Online]. Available: https://www.eskimo.com/~scs/cclass/int/sx4ab.html

[3] N. Bizzarrini, F. Grasso, and D. P. Coiro, "Genetic Algorithms in Wind Turbine Airfoil Design," *Ewea*, no. March, pp. 14–17, 2011.

[4] "XFOIL webpage - MIT." [Online]. Available: http://web.mit.edu/drela/Public/web/xfoil/

[5] "XFLR5 webpage." [Online]. Available: http://www.xflr5.com/xflr5.htm

[6] "MATLAB Documentation - MathWorks España." [Online]. Available: https://es.mathworks.com/help/matlab/

[7] F. White, "Fluid Mechanics," *McGraw-Hill,New York*, 2010.

[8] M. Cavazzuti, *Optimization methods : from theory to design.* Springer, 2013. [Online]. Available: https://www.researchgate.net/publication/261750290_Optimization_methods_from_theory_to_design

[9] F. Werner, "A survey of genetic algorithms for shop scheduling problems," *Mathematical Research Summaries*, vol. 2, p. 15, 2017.

[10] P. Della Vecchia, E. Daniele, and E. D'Amato, "An airfoil shape optimization technique coupling PARSEC parameterization and evolutionary algorithm," *Aerospace Science and Technology*, vol. 32, no. 1, pp. 103–110, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.ast.2013.11.006

[11] Wikipedia, "Genetic algorithm - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Genetic_algorithm

[12] D. A. Masters, N. J. Taylor, T. Rendall, C. B. Allen, and D. J. Poole, "Review of Aerofoil Parameterisation Methods for Aerodynamic Shape Optimisation," *53rd AIAA Aerospace Sciences Meeting*, no. January, 2015. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2015-0761

[13] "NACA airfoil - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/NACA_airfoil

[14] J. Hajek, "Parameterization of Airfoils and Its Application in Aerodynamic Optimization," *WDS'07 Proceedings of Contirbuted Papers, Part 1*, pp. 233–240, 2007.

[15] H. Sobieczky, "Parametric Airfoils and Wings." Vieweg+Teubner Verlag, 1999, pp. 71–87. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-322-89952-1_4

[16] V. H. Schulz and C. Schillings, "Optimal aerodynamic design under shape uncertainties," no. June, pp. 1–25, 2009.

[17] M. S. Khurana, H. Winarto, and A. K. Sinha, "Airfoil optimisation by swarm algorithm with mutation and Artificial Neural Networks," *47th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, no. February, 2009.

[18] R. Mukesh, K. Lingadurai, and U. Selvakumar, "Airfoil shape optimization using non-traditional optimization technique and its validation," *Journal of King Saud University - Engineering Sciences*, vol. 26, no. 2, pp. 191–197, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.jksues.2013.04.003

[19] R. Alves, S. Neto, L. Gonçalves Noleto, A. Cesar, and P. Brasil, "Parsec Parameterization Methodology For Enhancing Airfoils Geometry Using Pso Algorithm."

[20] A. Arias-Montaño, C. A. Coello, and E. Mezura-Montes, "Evolutionary Algorithms Applied to Multi-Objective Aerodynamic Shape Optimization," 2014. [Online]. Available: https://www.researchgate.net/publication/226702706_Evolutionary_Algorithms_Applied_to_Multi-Objective_Aerodynamic_Shape_Optimization

[21] Tutorialspoint, "C - Data Types." [Online]. Available: https://www.tutorialspoint.com/cprogramming/c_data_types.htm

[22] A. J. Umbarkar and P. D. Sheth, "CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW," *ICTACT JOURNAL ON SOFT COMPUTING*, p. 1, 2015.

[23] "Genetic Operators," Tech. Rep. [Online]. Available: http://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf

[24] O. Abdoun, J. Abouchabaka, and C. Tajani, "Analyzing the Performance of Mutation Operators to Solve the Travelling Salesman Problem," Tech. Rep.

[25] Wikipedia, "Natural selection." [Online]. Available: https://en.wikipedia.org/wiki/Natural_selection

[26] P. Camps, "Study: Airfoil optimization for the blades of a quadrotor," p. 60, 2018. [Online]. Available: https://upcommons.upc.edu/handle/2117/186230

[27] ——, "Genetic Algorithm - Application for tumor treatment," 2020.

[28] L. Alseda and A. Ruiz, "Genetic Algorithms application," pp. 1–2, 2017.

[29] Wikipedia, "Pseudorandom number generator ." [Online]. Available: https://en.wikipedia.org/wiki/Pseudorandom_number_generator

[30] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, "Numerical Recipes in C - The Art of Scientific Computing," Tech. Rep., 1988. [Online]. Available: http://www.nr.com

[31] RapidTables, "Binary to Decimal Converter." [Online]. Available: https://www.rapidtables.com/convert/number/binary-to-decimal.html?x=0101010101010101

[32] O. Traisak, T. Dolwichai, J. Srisertpol, and C. Thumthae, "Study of airfoil shape optimization by using the evolutionary method," no. January, 2016.

[33] S. M. Berkowitz, "Theory of wing sections," *Journal of the Franklin Institute*, vol. 249, no. 3, p. 254, 1950.

[34] "MinGW — Minimalist GNU for Windows." [Online]. Available: http://www.mingw.org/

[35] "Air Cargo Challenge - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/Air_Cargo_Challenge#cite_note-acc2013-1

[36] Venturi, "UPC Venturi Home Page." [Online]. Available: https://upcventuri.com/

[37] ——, "UPC Venturi - YouTube." [Online]. Available: https://www.youtube.com/channel/UCvLk8u2MuTgJ3lbO0skS7Lg

[38] M. Barahona, P. Bernard, X. Carrillo, P. Camps, D. Campos, P. Cos, J. Garreta, A. Ibáñez, P. Márquez, J. A. Martínez, L. Montilla, L. Rubio, G. Valero, A. Yébenes, and E. Zubillaga, "Final Report - UPC Venturi - ACC 2017," UPC Venturi, Tech. Rep., 2017.

[39] "Trencalos Team." [Online]. Available: https://trencalos-team.webnode.com/

[40] "The School of Industrial, Aerospace and Audiovisual Engineering of Terrassa. ESEIAAT — UPC. Universitat Politècnica de Catalunya." [Online]. Available: https://eseiaat.upc.edu/en?set_language=en

[41] M. Gómez, M. Pérez, and C. Puentes, *Mecánica de vuelo*. Madrid: Garceta.

[42] P. Camps, "Paper Air Challenge 2015 - Punto Extra - YouTube." [Online]. Available: https://www.youtube.com/watch?v=iuu49wNZWB0&feature=youtu.be