Internal Report 2012-03 March 2012

Universiteit Leiden
Opleiding Informatica

Designing and Implementing a System
for the Evolutionary Optimization
of an Airfoil

Martin Wimmers

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Chapter 1

Introduction

For many decades, evolutionary algorithms (or EA) have been used in finding optimal solutions for problems, and for improving already existing solutions.[?, ?] Several types of evolutionary algorithms have been created, with genetic algorithms and evolutionary strategies being the two main types in use. Several programs and libraries have been made (cf. the Shark Machine Learning Library [?]) that implement these algorithms, and such libraries can be used for various purposes.

One of the applications for evolutionary algorithms is the design of an unmanned aerial vehicle (UAV). An UAV is an airplane that is not flown by a pilot but controlled remotely. The flight characteristics of a UAV are determined to a large extend by the shape of the wing, called an airfoil. The airfoil determines how much lift is generated by a wing, but also the amount of drag. The less drag there is, the more efficient an UAV becomes, but care must be taken that sufficient lift remains. The goal of this thesis is to optimize an airfoil shape using an evolutionary strategy, which is a type of evolutionary algorithm.

Currently, various techniques are applied to optimize already existing airfoil shapes[?]. In many of these techniques, evolutionary algorithms are combined with other techniques to look for improvements. This paper [?] shows an example of how evolutionary algorithms are combined with a different technique for optimization purposes.

In this paper, the used technique is a state of the art evolutionary strategy method called Covariance Matrix Adaptation, or CMA-ES[?]. This technique will not be combined with other methods.

This thesis should provide a useful insight as to combining programs in improving design, and it should make it easier for others to use an EA for their needs. Whilst one can write an EA from scratch, it can save a lot of time and effort if a library is provided.

In summary, this thesis will describe how an EA is used to optimize a given airfoil shape. The main goal is to show that coupling an EA with a program that simulates an airfoil and calculates aerodynamic properties is possible. Moreover,

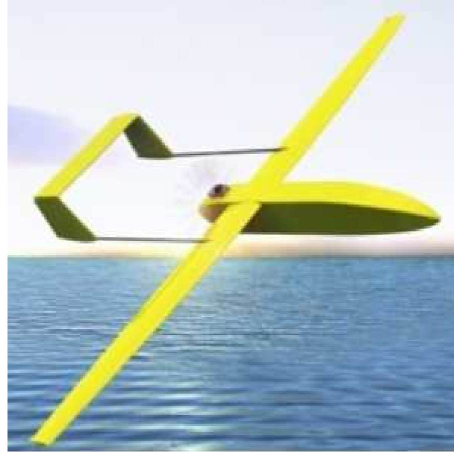

Figure 1.1: An example of an unmanned aerial vehicle, the vehicle type for which an airfoil will be designed. Source: National Oceanography Center, Southampton.[?]

the performance of an advanced EA will be compared to the performance of a very basic Monte Carlo search. The main question of this thesis is: How can we improve an existing airfoil shape using an advanced evolutionary strategy, and how well does it work?

The thesis is structured as follows: in chapter **??**, the optimization problem will be described, as well as some of the parameters used in the problem, including physics formulas and numbers. Then, in chapter **??**, the settings for the EA will be described, and how the link between the programs used is established. In chapter **??**, the technical details for setting up this experiment are provided. In **??**, experiments and their results will be described. Finally, in chapter **??**, there is a conclusion on this project. The technical details of the algorithm that calculates airfoil properties is provided in appendix **??**, and this algorithm can be implemented using different simulators and environments for finding optima.

Chapter 2

Problem Definition

The problem that has been stated is: how can we improve an existing airfoil shape using an evolutionary strategy? There is an initial shape, and that shape is to be modified and improved.

This creates an issue. One of the key operators of an EA is mutation, which corresponds to changing parameter settings. But how can one guarantee that changing a wing shape yields another wing shape? First of all, we must have a way of representing a wing. Wings that are used in large commercial airplanes are very complicated structures, with a varying cross-section over the length of the wing and different material usage. Moreover, performing computations on such shapes takes a very long time. For simplification, our airfoil is assumed to have only one single cross-section shape, that does not change. This reduces the problem to finding an optimal cross-section.

The way we represent a wing is through a set of (x,y) coordinate pairs that represent points through which the cross-section passes. However, mutating this shape can cause a cross-section to become extremely irregular and lose all the desirable aerodynamic properties. This problem can be solved, by making a set of control points. A simple example would be to have 4 control points: the leading and trailing edge, the top of the wing curve and the bottom of the wing curve. Then, we want to generate a spline that goes through the control points smoothly. That way, when we mutate the control points within a certain range, we still keep a relatively smooth aerodynamic shape, and the chance of obtaining a valid airfoil shape increases.

However, in practice we want to use more points, and this means that through mutation we can create a very rough shape, causing improvements to be very difficult if not impossible. Fortunately, we can also place restrictions on the control points, by keeping the leading and trailing edge positions fixed. This is done by modifying the program such that these points are excluded from the candidate solutions, and when drag and lift are computed, these points are automatically filled in. However, the simulator that computes lift and drag does not accept control points. The simulator assumes that the airfoil shape is a spline, and it requires as an input a file with a lot of coordinate points, through

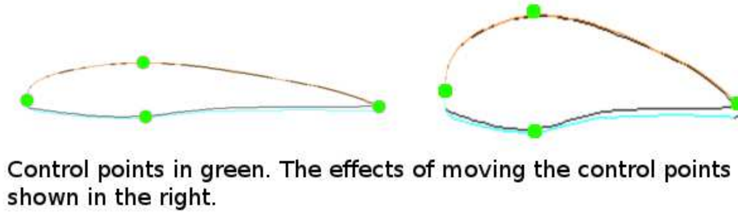

Figure 2.1: The effects of moving control points illustrated

which the airfoil passes. In order to generate such a file, several functions from a C++ library are used. The functions are used to generate a spline (a smooth curve) from control points, and exports them as a file that can be read by the simulator. This part of the program can be seen as an interpreter. This solves the problem of rough edges that can occur when mutating the spline coordinate points, by mutating control points. It also keeps the leading and trailing edge fixed, which makes comparing airfoil shapes easier.

The type of splines used to represent an airfoil shape is a B-spline. A B-spline is a function that is able to represent continuous, smooth curves. It can be thought of as a way to represent several Bézier curves, one linked after the other[**?**]. Since airfoil shapes are smooth continuous shapes, this lets B-splines be a good choice for representing airfoil shapes.

This paper will investigate how much of an improvement can be made to an existing airfoil shape, using a representation of an airfoil by using control points and mutating the wing shape with an evolutionary strategy.

As a novelty, a specific C++ library is used to implement the evolutionary algorithm. Lastly, a comparison between the evolutionary algorithm and a Monte Carlo search will be made. In figure **??** the initial wing shape is shown, as it is being processed by the simulator. The red and blue lines are not part of the shape, but they visualize the flow of air over the airfoil.

2.1 Related Work

As stated in the introduction, a lot of work has been done with various Evolutionary Algorithms, as well as other methods[**?**, **?**, **?**]. Frequently, methods used in the field of research combine evolutionary algorithms with other techniques, frequently applying advanced heuristics to attempt to boost the improvement rate. For instance, in [**?**] a Genetic Algorithm is used to find a set of solutions for a multi-objective function in combination with a Nash and a Pareto game, and this is applied to optimizing high lift systems, which are part of an airfoil. In [**?**], a Genetic Algorithm is used to reduce drag on an airfoil shape, and the focus of the paper lies on representing the airflow using a combination of a mesh and a meshless method.

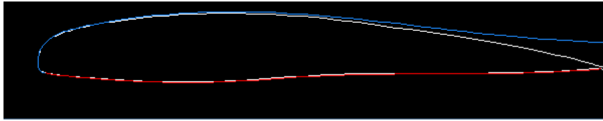

Figure 2.2: In white, the starting airfoil shape. In red and blue, the airflow as is visualized by the simulator.

Before an airfoil can be optimized, we must state what our objective is. Our objective is to minimize drag, with the added constraint that a wing must have sufficient lift in order to keep the airplane flying. Summarized shortly: the goal is to minimize drag of an airfoil whilst keeping a sufficient amount of lift. An evolutionary strategy will search for optima, and rather than mutating every piece of the wing, the ES will mutate control points and select improved shapes, which increases the likelihood for generated shapes to be feasible and becoming better.

Chapter 3

Method

To optimize a given airfoil shape, two C++ libraries, namely the Shark Machine Learning Library version 2.3.3[?] for implementing the ES, and Nurbs++[?] for translating control points into a B-spline curve, were used as well as a simulator program called Xfoil[?] to calculate properties of a given airfoil shape.

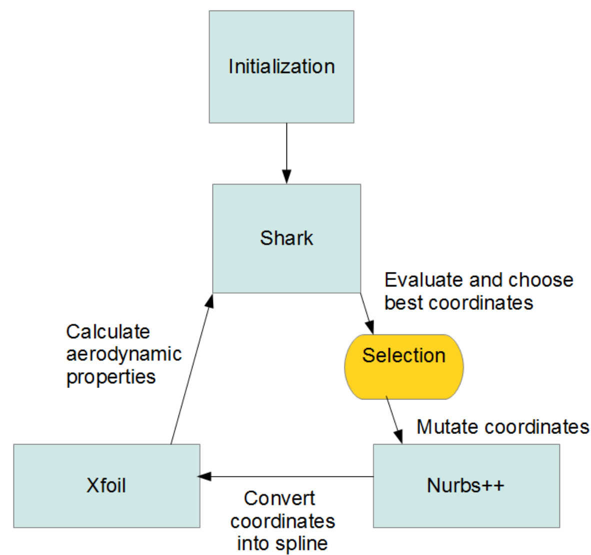

Figure 3.1: The flowchart of the program setup, illustrating the internal structure of the program that runs the experiments.

3.1 Shark

The Shark Machine Learning Library[**?**], or Shark, is a C++ library that provides machine learning capabilities and algorithms that can be implemented for various requirements and calculations. The functions that are used for this paper shall be described in detail, whilst other functionalities will be briefly mentioned, to show the large scope of Shark.

Shark has four main modules: ReClaM (Regression and Classification Models Toolbox), Fuzzy (used for Fuzzy logic), EALib (EA standing for evolutionary algorithms), and MOO-EALib (similar to EALib, but to be used for multi-objective functions). ReClaM is a set of tools for supervised learning with various methods, i.e. neural networks. For the purpose of this paper, the EALib was used.

The EALib has various classes, and methods for finding solutions from scratch, and optimizing given solutions. One can implement both genetic algorithms as well as evolutionary strategies, and the library provides functions for selection, crossover and mutation. There are also sample functions used for testing purposes, like the sphere function $\sum_{i=1}^{n} x_i^2$, or the Ackley function. For evolutionary strategies, there are also various searching techniques; the experiment in this paper used a CMA-ES[**?**], CMA being short for covariance matrix adaptation, an advanced adaptation mechanism for the mutation distribution in evolution strategies.

This experiment uses several default settings provided by Shark in the CMA-ES search: when the search is initialized, only the objective function whose value is to be minimized is provided, as well as a step size. It sets a value for λ (amount of children generated by the evolutionary strategy). Sharks default setting is: $\max(8, 4 + 3 * \log(dimension))$, where dimension is the amount of parameters that are varied. The population size μ is set as $\frac{1}{4}$ of λ, and no specific recombination method is set. When an elitist strategy is used, the (1+1)-CMA is chosen, where $\mu = \lambda = 1$.

In this experiment, the population consists of vectors of real valued numbers, represented as arrays of double precision floating point numbers, that correspond to the control points.

3.1.1 Covariance Matrix Adaptation

A covariance matrix adaptation evolutionary strategy (or CMA-ES) is an optimization method that is based on a conventional ES. One of the disadvantages of a conventional ES is that there is either no convergence preference (when there is a single step size for all parameters), or that the convergence preference can only be directed in a perpendicular direction. In most cases, the optimal solution does not lie in a direction parallel to an axis, but lies at an angle, as is in the image below. Figure **??** also shows the basic principle of how a CMA-ES works. The optimum lies at an angle, and not in a direction parallel to the axises. An ordinary ES that does not use a Covariance Matrix is only able to

converge quickly in a direction parallel to the x and y-axises, not in another direction. But a CMA-ES is able to move towards the optimum more quickly, since it is able to obtain a preference direction at an angle to the x and y-axises.

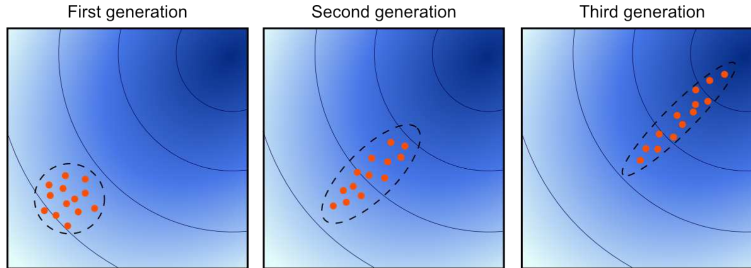

Figure 3.2: The advantage of CMA-ES visualized: despite the optimum not lying perpendicular to the starting population, a preference of the population for moving towards the optimum is obtained. Original source: [?]

The covariance between parameters is expressed in a covariance matrix. The covariance matrix is adapted at each step to maximize the chance of getting an improved solution at the next step. The exact algorithm has been implemented in Shark. For a tutorial on how the CMA-ES works, the reader is referred to:[?].

3.2 NURBS

Nurbs++ is a C++ library that is used for generating and manipulating Non-uniform rational B-splines (or NURBS). Since Nurbs++ is sometimes abbreviated to Nurbs, when Nurbs is not written in all capital letters, the library is meant. When NURBS in all capitals is written, it's the abbreviation for the Non-uniform rational B-spline. For this papers purpose, this library is used to generate a B-spline that represents an airfoil shape. As an input, it receives the set of control points, and it outputs a spline based on the control points.

As an exception, the points at the leading and trailing edge are fixed. The Nurbs function takes care of that. Since the candidates of real-valued number vectors are represented by arrays of doubles, the function automatically fills the fixed points in when they appear, and these points are not elements of the array that becomes mutated. Also, the two points at the leading edge are only adjustable in the y-direction, so the x-coordinate is fixed as well. Hence, the fixed numbers are not part of the candidate solution. By keeping the fixed coordinates out of the candidate solutions, we do not have to worry that these solutions are mutated, and that way the resulting shape will more likely resemble a wing.

3.3 Xfoil

Xfoil is a simulator that is used to calculate properties of a given airfoil shape. The airfoil shapes are assumed to be B-splines, and Xfoil requires as an input a file that has all the coordinate points through which a wing passes. The B-spline curve is seen as a large set of (x, y) coordinate pairs, through which the spline passes.

In this experiment, Xfoil is run twice for each evaluation: once it is started, it simulates cruising conditions, to measure the drag, and then it simulates landing conditions, to measure lift.

3.4 Objective Function

The objective function, that is fed to Shark, works as follows: it takes the current solution, the set of control points, and provides them into Nurbs++. Nurbs++ then converts these points into a B-spline, and outputs a set of coordinates through which the B-spline goes into a file. Then, Xfoil is run twice, once simulating cruising conditions, and once simulating landing conditions. The same file with coordinates is fed to Xfoil both times. The value provided by the objective function is the drag value at cruising conditions, as long as there is sufficient lift at landing conditions. Otherwise, it will return the highest value a double precision floating point number can have.

This method of choosing a maximal value as a penalty was chosen, rather than a penalty function, because a feasible wing shape is given, rather than searched. If a penalty function were implemented, a definition of wing-like shape would need to be provided, and splines that are similar to feasible wings would have a better fitness value than splines that do not resemble wings at all. Finding and formally describing the properties of a wing-like shape is beyond the scope of this project.

As has been stated in chapter **??**, an initial wing shape is provided, and we merely are interested in optimizing, rather than generating. Whether the decision to give all infeasible wings the same penalty value would cause the algorithm to converge to a local, rather than global optimum is uncertain. However, since the initial shape is an airfoil, and airfoils are all similar shapes, this seems to be a feasible goal.

3.5 Local Monte Carlo Search

In order to compare the ES, a Monte Carlo[**?**] search has to be performed. However, since we are starting with an initial wing shape, we have to perform a search based on the wing shape. This problem, however, is easily solved by considering the population that a MC search generates as an offset; a MC run generates a vector of real number called an offset vector, and when these values are added to the original wing shape, the resulting shape will still resemble a wing. Since we are not randomly creating new shapes, but making random

modifications, we are actually performing a local Monte Carlo search. If we would be attempting to generate airfoil shapes randomly, then most of the time we would end up with shapes that do not resemble wings. If we would be keeping the previous shape, and generate a new shape, and take the best of the two, we would perform a (1+1) search, which is not true Monte Carlo but rather a greedy hill climbing algorithm. Two types of local Monte Carlo search were performed, in the first run, a search was performed with uniformly distributed random numbers between $[-stepsize; +stepsize]$. In the second run, Gauss normal distributed random numbers were used, the numbers being generated with$\mu = 0$ and $\sigma = stepsize$.

Chapter 4

Parameters

A few parameters had to be set up for the experiment. These are divided into two parameters: natural and algorithm parameters. The natural parameters are the physics constants, as provided by the project description. These parameters are never changed, and determine properties of the wing and the environment.

4.0.1 Environmental Parameters

The natural parameters that are used are: air density, speed during landing and cruise, vehicle mass, wing area, Reynolds number and angle of attack. The air density refers to the amount of molecules in air, expressed in mass, in a given volume. The angle of attack is a term used in fluid dynamics to describe the angle between a reference line on a lifting body (often the chord line of an airfoil) and the vector representing the relative motion between the lifting body and the fluid through which it is moving. The angle of attack is the angle between the lifting body's reference line and the oncoming flow[**?**]. The Reynolds number Re is a dimensionless number that gives a measure of the ratio of inertial forces to viscous forces and consequently quantifies the relative importance of these two types of forces for given flow conditions[**?**].

We assume a few things: the air density is $1.225\frac{kg}{m^3}$, the cruise speed is $30\frac{m}{s}$, landing speed is $18\frac{m}{s}$, and the mass of the vehicle (to which the wing is attached) is $15kg$, and the maximum landing weight is $11.4kg$. The total wing area is $1.2m^2$, this coincides with a 3 meter long wing that is $40cm$ wide. The Reynolds number at cruise is 10^6 and $6 * 10^5$ during landing, which is in the turbulent regime. The angle attack at landing is 10 degrees and its floating during cruise (not fixed).

Xfoil requires the Reynolds number, air density, the speed and angle of attack. With the other numbers, the minimal lift coefficient could be calculated, with the formula below. The result of that equaled $C_{L,land} \approx 0.469$.

$$C_{L,land} \geq \frac{m_{land} * g}{\frac{1}{2}\rho U_\infty^2 S}$$

m_{land} is the mass of the unmanned aerial vehicle. $m_{land} = 11.4kg$

g is the gravitational acceleration. $g = 9.81$

ρ is the air density. $\rho = 1.225\frac{kg}{m^3}$

S is the wing area. $S = 1.2m^2$

U_∞ is the speed.$U_{\infty,land} = 18\frac{m}{s}$.

4.0.2 Algorithm Parameters

For the evolutionary strategy, I mostly used the standard parameters Shark provides. I did not explicitly state a population size or crossover method, so the default values were used. The method used, a covariance matrix adaptation search, or CMA-ES, was implemented by the library, and the library usually provides good standard parameters.

However, I did have to change the initial step size, as the default step size of 1 is so large that the ES would create mutations that hardly represent wing shapes. Using a smaller step size, the algorithm creates smaller perturbations that still resemble airfoil shapes, which is beneficial for maintaining progress the calculations. During experimentation, I have tried several step sizes, to see differences in convergence and end results.

Chapter 5

Compiling and Running

5.1 Installation

The program that conducts the experiments is written in C++ and runs on Linux. Three pieces of software are required: Xfoil, the Shark Machine Learning Library and Nurbs++.

Firstly, the Shark Machine Learning Library version 2.3.3 has to be installed. The directory where the installation has to take place is: `~/software/`. The result should be a folder `~/software/libshark-i386-2.3.3-linux`, with the library being installed in that folder.

Then, Xfoil 6.97 has to be installed. Xfoil can be downloaded at [?]. For Xfoil, a FORTRAN compiler is required. Also, when installing Xfoil, there is an error in the source code: this website [?] provides the solution.

Finally, the Nurbs++ library has to be installed. The easiest method for installation is to extract a zip file provided by the author of this paper. Upon extraction, the makefile and the Nurbs++ library are all present. If the Shark library was installed in the exact directory as described(`~/software/libshark-i386-2.3.3-linux`), then the makefile should work correctly and the program can be executed. The program itself is located in the `example.cpp` file.

5.2 Setting parameters

If the user wishes to change parameters when conducting these experiments, the user is referred to the `example.cpp` file. If a user wishes to change parameters involving the Monte Carlo run, then the function `MonteCarlo(double range)` provides all the desired parameters available. For changing paramters in the ES run, the user should look in the `main()` function. After all parameters are set, the program can be compiled using `make`.

5.3 Running

When running the program, it is recommended to pipe the output to a file by issuing the command

```
./example name > output_file.txt.
```

Some of the output of the program is sent to the terminal through the I/O error stream, to monitor progress of the program. All data involving indiviudual solution and fitness values are sent to the normal I/O, and by sending it to a file, the data can be examined more easily.

Lastly, when the output file is examined, there will be many lines containing the following text:

```
15 19
in generate curve
```

This text is generated by some of the Nurbs++ functions called. These lines can be ignored safely. The useful information are the matrices of individuals and the matrices of fitness values, which are demarcated by a line of text.

Chapter 6

Results

This chapter consists of three parts: the first section describes the initial airfoil shape. The second section describes the results obtained by conducting a Monte Carlo search on the wing. The third section describes the results obtained by performing a CMA-ES experiment.

6.1 Initial Wing

Before the calculation could start, an initial wing shape has to be defined by the control points. Table **??** lists in correct order the values of each element of the vector that represents a wing.

The algorithms used to find better shapes work by modifying offsets, i.e. values that are added to this vector. If the offset vector is a null-vector (a vector where all values are 0), then the candidate solution represented is the original shape. A picture of the resulting initial airfoil shape is located in chapter **??**.

Initially, two experiments were conducted, each with a different set up. Both the Monte Carlo and the CMA were called 100 times. As a Monte Carlo algorithm only calls the fitness function one time each run, and the CMA calls the fitness function at least 10 times, at least once for each of the children, the comparison does not seem fair. Moreover, the Monte Carlo search was performed by generating offsets, whereas the CMA modified the initial airfoil shape. Fortunately, some useful information can be gathered from this initial trial, as it turns out that whilemthe CMA does find a better shape, the Monte Carlo search turned out better solutions in all the cases. Since the fitness function was called a lot fewer times, the calculation time for the Monte Carlo was also significantly less than for the CMA.

6.2 Monte Carlo Search

For the Monte Carlo search, the search is performed by generating random offsets. Two types of runs were conducted, one using uniformly distributed

index	value
0	0.9
1	0.03
2	0.7
3	0.06
4	0.5
5	0.09
6	0.3
7	0.1
8	0.1
9	0.08
10	0.05
11	-0.01
12	0.3
13	-0.03
14	0.5
15	-0.01
16	0.7
17	-0.01
18	0.9
19	-0.01

Table 6.1: The coordinates of the control points of the initial wing, excluding the fixed points.

range	best fitness
0.005	0.00785
0.01	0.0722
0.05	0.00672
0.1	0.1696

Table 6.2: Fitness values obtained through a local Monte Carlo search with 1000 iterations. The numbers are generated using uniformly distributed random numbers

step size	best fitness
0.0001	0.0089
0.0005	0.0088
0.001	0.0087
0.005	0.0068
0.01	0.0065

Table 6.3: Initial results for a Monte Carlo run with Gaussian random numbers, with $\mu = 0$ and $\sigma = stepsize$

random numbers within a search space $[-range; range]$, and one using a Gauss random number generator, with $\mu = 0$ and $\sigma = range$. For each step size, 1000 evaluations were performed. The results of the Monte Carlo search is shown below:

The results are displayed below:

For larger step sizes, no feasible results were found.

However, this Monte Carlo run was very restricted: the random numbers that were generated were generated within a fixed range, so it was not possible to generate all shapes. When the airfoil shape with a fitness value of 0.00323 was verified, the shape did not converge. This is one of the many faulty values that were generated by Monte Carlo.

Another run could be made, with random numbers generated using a Gauss normal distribution. This would allow all shapes to be generated, as it is theoretically possible to generate all shapes, albeit not all with equal probability. These experiments have also been conducted, and are described in the next section.

6.3 CMA-ES

Initially, a (μ, λ), CMA-ES was performed, and rather than an offset, the actual airfoil shape was modified. For μ and λ, default settings in the Shark library were used. Five evaluation runs were conducted, and each with a different stepsize, but feasible solutions were only found for two stepsizes used. The

stepsizes, and fitness progression over time are visible in figure **??**.

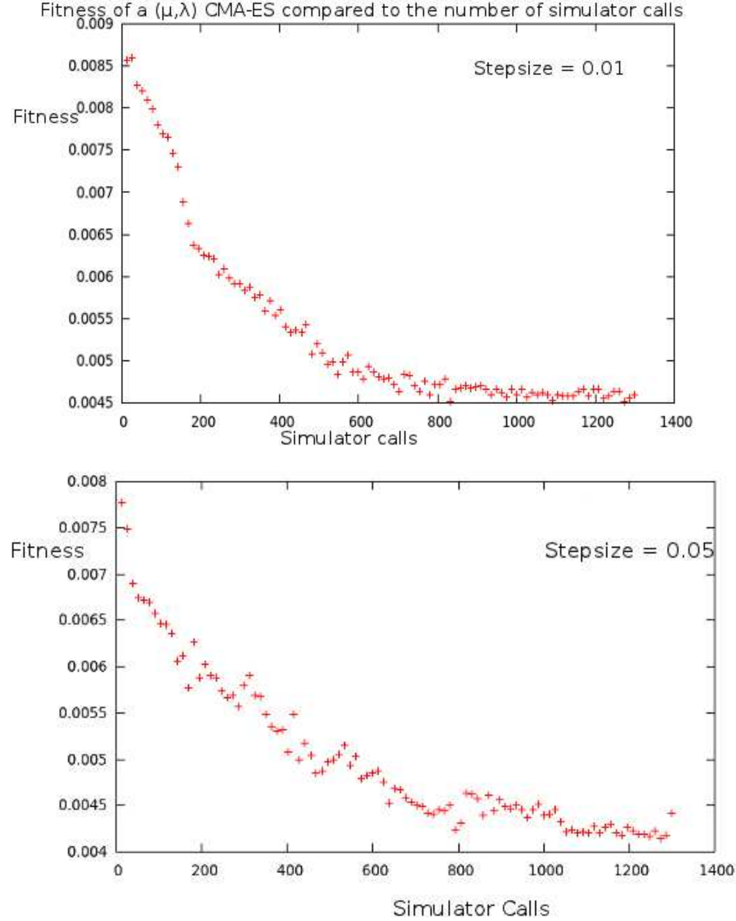

Figure 6.1: Two (μ, λ)CMA-ES runs, with step sizes provided and further parameter settings left default. In these runs, the actual airfoil shapes were modified, as opposed to offsets in other experiments.

Two issues arose from these results: how could these values be compared to Monte Carlo? And could the strategy be improved? It was from this experiment that the decision to use offsets in all experiments was made. Secondly, an elitist strategy seemed to be more appropriate in this setting, as the goal is to optimize, rather than generate, an airfoil shape. Using a comma strategy, a risk exists that infeasible solutions will be generated. Since all infeasible solutions are treated equally in the fitness function, the CMA-ES has no possibility to find a feasible solution if the current solution is infeasible, except through sheer luck. As a consequence, for the next experiments, a $(1 + 1)$ CMA-ES strategy was used.

Three experiments of five runs were made. In the first experiment, an initial

step size	best fitness
0.005	0.00487
0.01	0.00487
0.05	no value found
0.1	no value found
0.5	no value found

Table 6.4: Results of a (1+1) CMA-ES, starting with a random offset generated by the stepsize.

step size	best fitness
0.005	0.00489
0.01	0.00425
0.05	0.00566
0.1	0.00456
0.5	0.00379

Table 6.5: Results for a $(1 + 1)$ CMA-ES, with a zero-offset vector as an initial solution. The result for step size = 0.5 was verified and proven correct.

vector was randomly generated, with each element of the vector being a Gauss random number, generated with $\mu = 0$ and $\sigma = stepsize$. Since the initial offset vector generated when the step size equaled 0.05, 0.1 and 0.5 was infeasible, it shows that the algorithm cannot find a feasible value if there is not feasible candidate to start with.

The previous experiments show that the only way the $(1 + 1)$ CMA-ES can converge, is to guarantee that the initial solution is feasible. A way to accomplish this, is to give as a starting solution a vector consisting of pure zeros, so that the initial solution is the actual wing. These experiments were conducted, and the results are in this table:

The best airfoil shape, with fitness of 0.00379, was verified, and the fitness value was independently verified. The airfoil shape is shown in the image below:

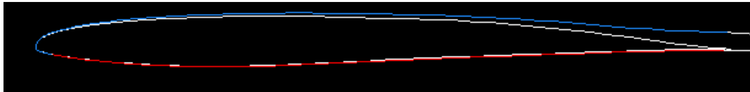

Figure 6.2: The best airfoil shape found and verified in all experiments conducted.

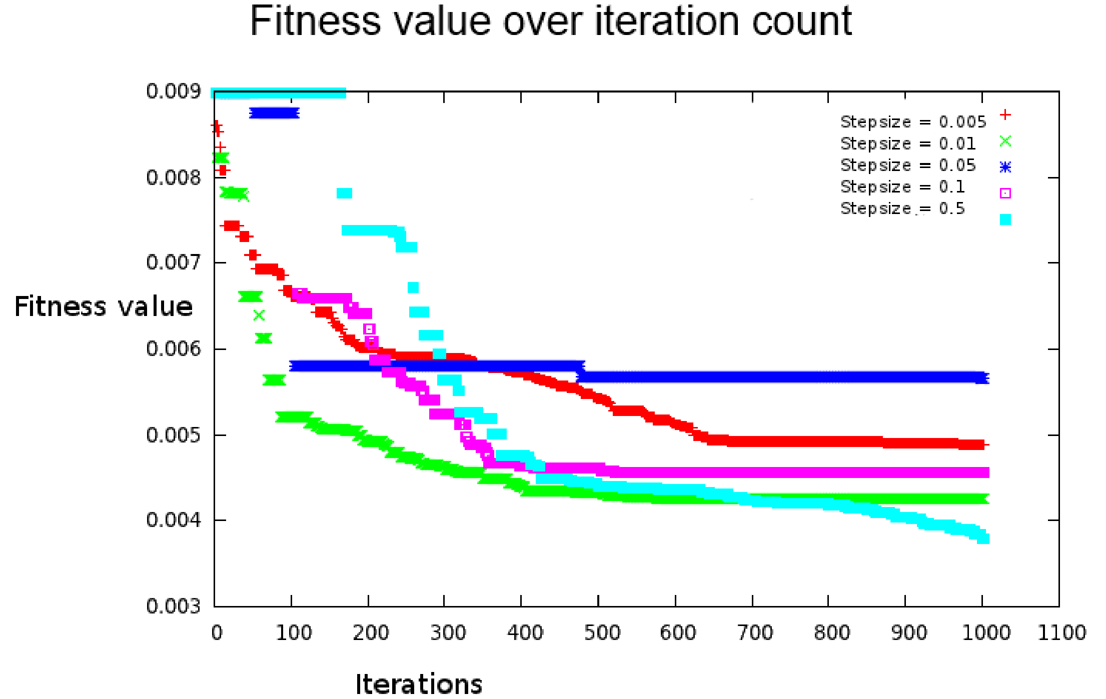

Figure 6.3: The fitness value of various (1 + 1) CMA-ES runs over time for various initial step sizes

The results are clear: starting with an offset of zero, good results were obtained. The largest step size chosen in the previous experiment works best. Whilst a smaller step size did find a better solution relatively quickly, it seemed to get stuck on a local minima. With the larger step size, the algorithm was able to improve much more, once it found a feasible solution. Since the largest step size suddenly provided the best result, another run was conducted, again with the initial offset vector being the null vector, but now with the step size of 0.5 being the median step size. These results are shown below:

This again suggests that for this specific problem, the optimal step size is located at 0.5. This is about $\frac{1}{4}$ of the range in which the coordinates of the initial airfoil shape lie. In graph **??**, the best fitness value for each iteration of the CMA-ES are shown.

step size	best fitness
0.1	0.00505
0.25	0.00507
0.5	0.00385
0.75	0.00401
1.0	0.00478

Table 6.6: Results for a (1 + 1) CMA-ES, with initial offset being zero. The optimum initial stepsize seems to be around 0.5 for this problem.

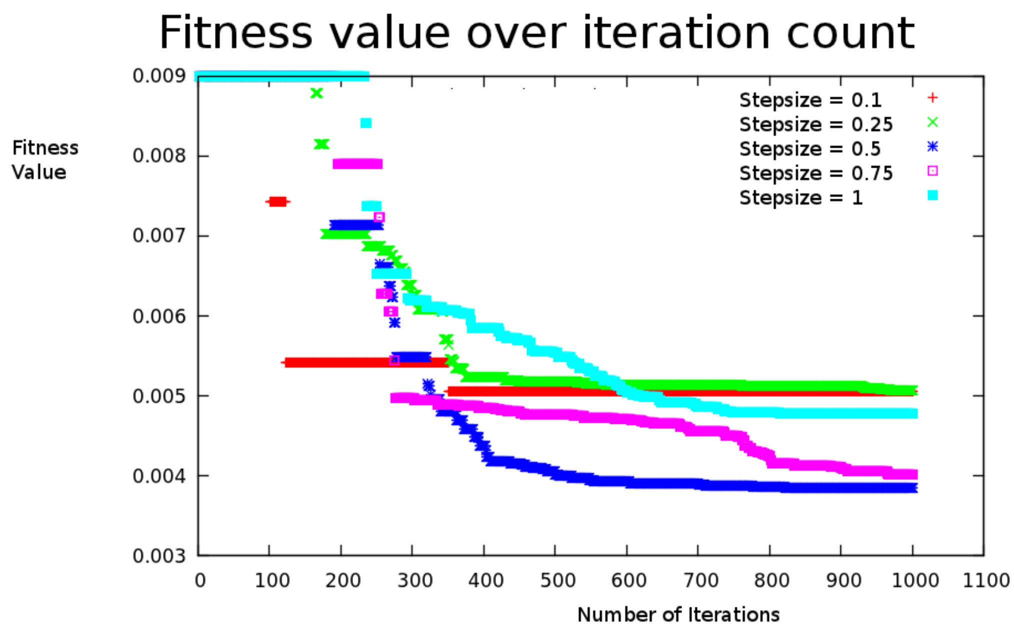

Figure 6.4: The best fitness value for a (1+1)CMA-ES run, with the final results being displayed in the table above.

Chapter 7

Conclusions and Outlook

7.1 Conclusion

Whilst the Monte Carlo search did result in some optimized airfoil shapes, the better results were provided by the $(1 + 1)$ CMA-ES. The drag was reduced from an initial 0.00899 to 0.00379, which is an improvement of 58%. Moreover, when looking at the graphs where the progress of the fitness value is plotted, it is evident that the optimal step size for the CMA-ES in this particular example lies at 0.5. For Monte Carlo runs, the optimal range was difficult to assess, but the Monte Carlo has only yielded an improvement of 30%, about half the improvement achieved using the $(1 + 1)$ CMA-ES

When comparing the optimal airfoil shape (figure **??**) to the starting airfoil shape (figure **??**), the shape is a lot thinner than the starting shape, as can be seen in **??**. Possibly long and sleek shapes are the most aerodynamic cross-sections that exist.

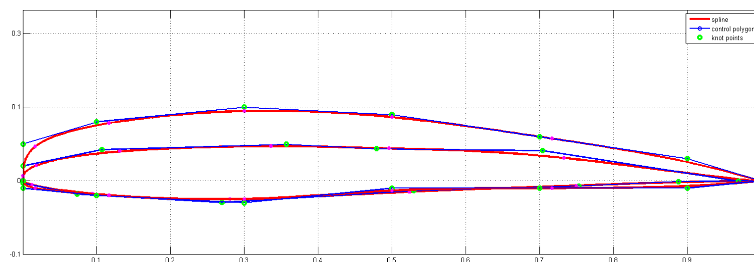

Figure 7.1: The initial and the best airfoils, plotted in one graph. The thinner shape is the best airfoil.

7.2 Discussion

The algorithm which we used is restricted to using a wing with a single cross section. However, most modern aircraft have complex wingshapes. Testing to see whether a CMA-ES would also perform well on these types of wings is a possibility for future research. A different simulator is needed to simulate these airfoil types.

The $(1+1)$ CMA-ES worked performed better than the (μ, λ) CMA-ES. A possible explanation for that is the observation that when no feasible solution is in the population, the CMA-ES can only find one through luck, as all infeasible solutions have the same penalizing fitness value. A $(1+1)$ CMA-ES guarantees that there is always a feasible solution, provided that the initial solution is feasible. In table **??**, it is evident that having no feasible initial solution results in no feasible solution being found. Moreover, the fitness landscape for this problem is very complex, and a $(1+1)$ CMA-ES is better at traversing paths through complex landscapes, whereas a μ, λ searches the nearby area. When there is a narrow path, surrounded by infeasible solutions, the various other candidates may distract from traversing the path to the optima.

7.3 Summary

It was possible to link 3 programs in order to optimize an airfoil shape. 2 methods were used to find improved shapes: a CMA-ES, and a Monte Carlo search. The $(1+1)$ CMA-ES search provided the best results. The Monte Carlo search did yield some optimized airfoil shapes, but these were not as good as the best ones found by the $(1+1)$ CMA-ES, using an equal amount of function evaluations.

Bibliography

[1] http://www.noc.soton.ac.uk/nmf/usl_index.php?page=iop

[2] Thomas Bäck - Evolution Strategies, Evolutionary Programming, Genetic Algorithms Oxford University Press, New York 1996

[3] H. Wang, J. Périaux, "Bump Aerofoil Design Optimization Using Hierarchical Genetic Algorithms and Hybrid Mesh/Meshless Methods for Drag Reduction", EUROGEN 2011

[4] D. S. Lee, J. Periaux, G. Bugeda, and E. Onate, "Multi-Objective High Lift Systems Design Optimisation Using Hybridised Evolutionary Algorithm With Nash-Game", EUROGEN 2011

[5] A brief introduction to the CMA search algorithm http://www.lri.fr/~hansen/cmaesintro.html

[6] The website explaining how to fix the Xfoil installation http://blmath.wordpress.com/2010/04/04/installing-xfoil-on-fedora-12-linux+

[7] The Shark Machine Learning Library http://shark-project.sourceforge.net

[8] Xfoil website raphael.mit.edu/xfoil/

[9] Nurbs++ website http://libnurbs.sourceforge.net/

[10] MOENS Frederic, "Numerical Optimisation of the Flap Position of a Three-Element High-Lift Airfoil in 2D and 2.5D Flow using Navier-Stokes Solver", EUROGEN 2005

[11] The Monte Carlo method explained http://mathworld.wolfram.com/MonteCarloMethod.html

[12] The definition of Angle of Attack: http://en.wikipedia.org/wiki/Angle_of_attack

[13] The definition of the Reynolds number: http://en.wikipedia.org/wiki/Reynolds_number

[14] A.J. Keane "The Place of Evolutionary Search Tools in Problem Solving Environments for Aerospace Design",EUROGEN 2001

[15] The wikipedia article on CMA-ES http://en.wikipedia.org/wiki/CMA-ES

[16] Nikolaus Hansen, TutorialThe Covariance Matrix Adaptation Evolution Strategy (CMA-ES),2008 `http://www.lorentzcenter.nl/lc/web/2008/305/presentations/Hansen.pdf`

[17] Thomas W. Sederberg, An Introduction to B-Spline Curves, 2005 `http://tom.cs.byu.edu/~455/bs.pdf`

Appendix A

The fitness function

The key objective of this thesis is to create and implement an algorithm that returns the fitness value of an airfoil shape. The algorithm behind the fitness function is explained in more detail in this section. The algorithm will be explained without details that are specific to the implementation, so terms like vectors of real values, rather than double arrays, are used. By leaving out environmental specific details, the algorithm can be implemented in different environments, for instance a different simulator can be used, or a different algorithm for finding optimal values.

A.1 Initializing

The fitness function has a single parameter, a vector of real valued numbers that corresponds to the offsets of the initial solution. This offset vector is then added to the initial solution vector, and the sum of the vectors is saved into a vector. This vector is the actual representation of the airfoil shape, and will be called the wing vector. The wing vector is then converted into a spline, this is done in a function that calls the Nurbs library. This function is adapted specifically for this problem, since some key control points that are not allowed to be mutated are left out of the wing representation.

A.1.1 Converting the wing vector into a spline

The wing vector is provided as a parameter to the function that converts the wing vector into a spline. The function copies the wing vector into another vector called the control points vector. However, the control points that are fixed, and thus not part of the wing vector, are also added to the control points vector in the correct place. Then, the knot points are created, and finally, Nurbs generates the B-spline, which is saved to a file.

Since the simulator used in this implementation does not accept closed curves, and a wing is a closed shape, 0.0002 is added to the last control points.

This does not affect the aerodynamic properties significantly but it prevents the shape from being closed, so that the simulator accepts and calculates the lift and drag coefficients correctly. This is done specifically for the simulator used here, and different simulators might not need this offset to work.

A.1.2 Simulating the airfoil

The B-spline is saved to a file, and the file is a set of coordinate points through which the airfoil passes. Since the simulator is a separate program, a system call is used to run the simulator. When the simulator is called, it is first run at landing conditions, to check for sufficient lift. Since this is run on Linux, this actual C++ statement is used:

```
+system("~/Xfoil/bin/xfoil curve000.dat < ~/Xfoil/land.in >output &");
```

. Xfoil is the simulator used for this experiment, curve000.dat is the spline file, land.in is the file with the commands for Xfoil that simulate landing conditions, and the output is written to a file named output. Since the simulator used in this experiment setup does not terminate if the spline is not aerodynamic, an ampersand is added to the end of the system call, so that the program can continue running.

The program then pauses for a few seconds (any time is possible, but 5 seconds is used as all wings with good fitness values are computed within that time). After the time passed, the simulator is terminated using a system call with the `killall` command. If the simulator was still running, the wing is considered inadmissable, else the landing coefficient is collected from the output file.

Finally, the simulator is run again, in the same way, except it now simulates cruising conditions, and if the simulator terminates within the same amount of time again, the drag is obtained from the output file. The fitness value is the drag coefficient value, if an airfoil shape is admissible (sufficient lift, and computed within reasonable time), else the fitness value is the largest value possible.