TÉCNICO LISBOA

Aircraft Optimal Design

Aerospace Engineering

José Miguel Fonseca Santos

83704

3rd Assignment

24/11/2019

Contents

1 Constrained Optimization using *OpenMDAO*

The aim of the first exercise is to use *OpenMDAO* to solve the *Rosenbrock* minimization problem. In order to solve this optimization problem one needs to define the model, necessary constraints and derivatives.

1.1 Analytical partial derivatives of the *Rosenbrock* function

The *Rosenbrock* function to be minimize is defined by the following expression.

$$min \qquad (x) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$$

Computing the partial derivatives, one can easily obtain the expressions bellow.

$$\frac{\partial f}{\partial x_1} = 400x_1^3 + 2x_1 - 400x_1x_2 - 2$$

$$\frac{\partial f}{\partial x_2} = 200x_2 - 200x_1^2$$

1.2 Schematic structure of the model

Using *OpenMDAO*, the first step is to get the design structure matrix, in order to have a global insight of the model. This matrix is compact and allows the user to have visual representation of a system or project in the form of a square matrix. Also known as dependency structure matrix it lists all the the subsystems and information exchange. In the *OpenMDAO* environment the lowest level system is defined as a *Component*. On the other hand, a *Group* is used to build a complex model, incorporating other *Components* or *Groups*. This is what defines the hierarchy that constitutes and defines the model. Another useful characteristic is that it allows to understand what pieces of information are needed to start a particular *Component* and where this information generated is being fed. For the *Rosenbrock* function the obtained structure is shown in figure 1.

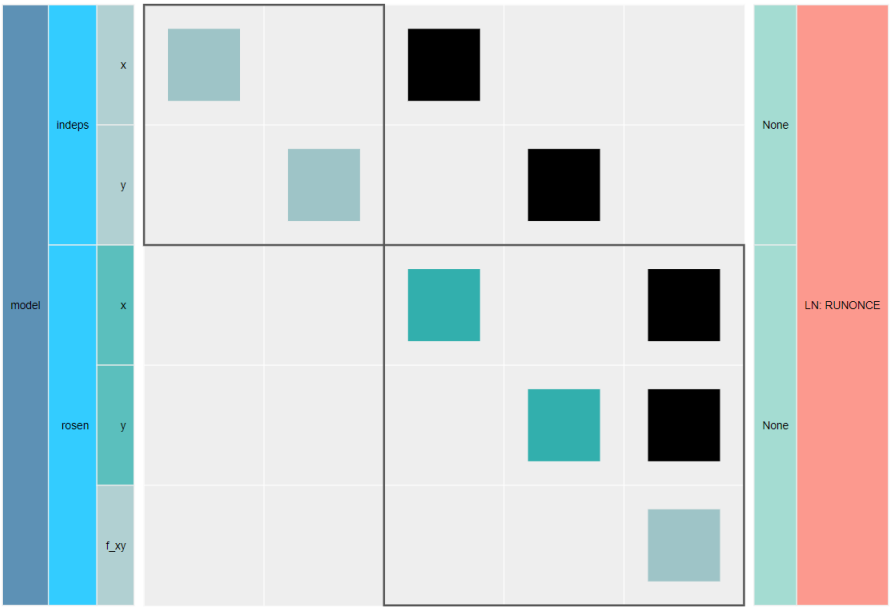

Figure 1: Schematic representation of the structure of the model

The model hierarchy is displayed on the left side, containing two sub-systems, one containing the independent variables that are connected to the *Rosenbrock* function inputs. The main diagonal represents all the inputs and outputs and the off-diagonal right side contains information regarding data connections. In this simple example, the independent variables are fed into the *Rosenbrock* function domain and inside this sub-system, they are further used to compute $f(x_1, x_2)$.

1.3 Solution of the unconstrained problem

The next step is to implement the unconstrained minimization of the function using *OpenMDAO*. The initial solution remains constant throughout this subsection in order to analyse the influence of providing the analytical partial derivatives computed above *versus* using a finite difference method. Firstly, it is required to present some of the default parameters used by *ScipyOptimizeDriver*. These are shown in table 1.

Optimizer	Max. Iterations	Relative Tolerance
SLSQP	200	1E-6

Table 1: Default options for *ScipyOptimizeDriver*

The results obtained in table 2 show that the optimizer was able to found the global minimum of the function in both cases. When the partial derivatives are computed recurring to the finite difference method the number of iterations, function and gradient evaluations are slightly smaller comparing to when the same partial derivatives are calculated using the analytical expression calculated in the first subsection. The exit mode is 0 for both cases which means the optimizer stoped due to the default tolerance value being reached. A profiling capability was implemented allowing the analysis of the number of function calls and the CPU time taken to solve the optimization problem. As expected, the number of function calls is more than double for the finite difference method. As for the CPU time, it is

also double which means that even for an expression with relatively simplicity, the performance increases significantly knowing the exact partial derivatives.

x_0	Der. method	$f(x^*)$	x^*	Iter.	F. eval	Grad. eval	Exit	F. called	CPU time
(-1.2,1)	'fd'	1.498×10^{-7}	(0.9996,0.9992]	32	44	32	0	109	0.003510
(-1.2,1)	'analytical'	4.114×10^{-7}	(0.9994,0.9987)	34	45	34	0	46	0.001571

Table 2: Unconstrained minimization details

1.4 Solution of the constrained problem

Having results for the unconstrained problem, the next step is to define a constraint given by equation 1.

$$g(x) = x_1 + x_2 \leq 1 \tag{1}$$

Once again, the dependency structure matrix (figure 2) was obtained in order to analyse the structure of the problem in the *OpenMDAO* environment. This time, there is present one extra group or sub-system responsible for computing the constraint and evaluating if the solution respects it. One can realise that the groups corresponding to the *Rosenbrock* function and the constraint are not directly linked, e.g. there is no direct data exchange between them.

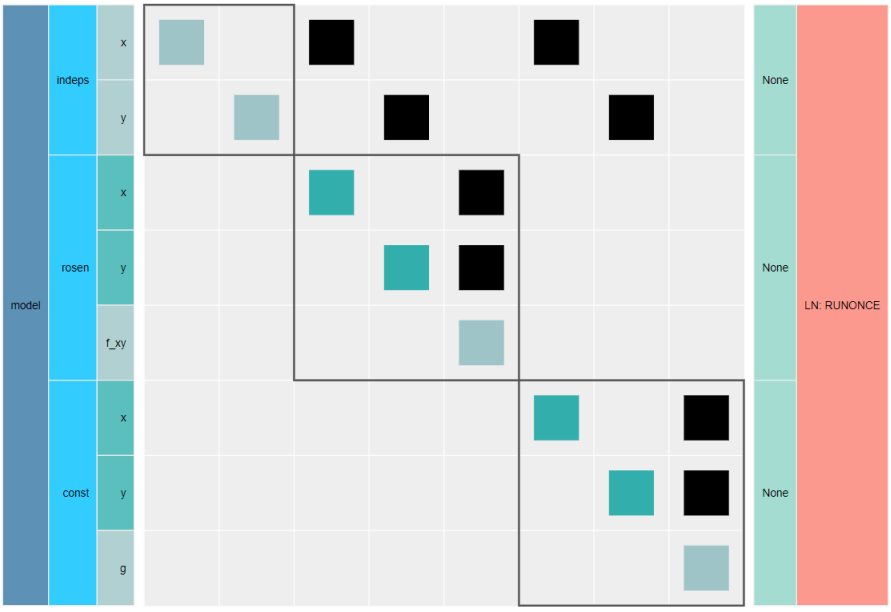

Figure 2: Schematic representation of the structure of the constrained model

This time, there will be two initial solutions to study. The first one is the same as the one from the unconstrained problem. The second one is the optimal solution found above. The results are shown in the following table.

Der. method	$\boldsymbol{x_0}$	$f(\boldsymbol{x^*})$	$\boldsymbol{x^*}$	Iter.	F. Eval	Grad. Eval.	Exit
'fd'	$(-1.2, 1.0)$	0.1456	$(0.6188, 0.3812)$	22	32	22	0
'fd'	$(1.0, 1.0)$	0.1456	$(0.6188, 0.3812)$	12	21	12	0
'analytical'	$(-1.2, 1.0)$	0.1456	$(0.6188, 0.3812)$	22	32	22	0
'analytical'	$(1.0, 1.0)$	0.1456	$(0.6188, 0.3812)$	12	21	12	0

Table 3: Constrained minimization with different initial guesses

Since the results were exactly the same, one could presume that the analytical partial derivatives were not being well implemented in the code. Thus, the function *check_partials* was used to compute the relative error for each one of the derivatives. The results obtained are shown in table

	Relative error	
	'forward'	'reverse'
'f_xy','x'	0.000455	0.000455
'f_xy','y'	0.000292	0.000292

Table 4: Relative error of the partial derivatives

The relative error shows that there is almost no deviation from calculating the partial derivatives using the finite difference method or using the analytical expression what explains the similarity of results obtained previously. The profiling capability previously mentioned was again implemented in order to get access to the number of time each function is called and the CPU time taken to solve each problem. The results shown in table

Der. method	$\boldsymbol{x_0}$	F. calls	CPU time
'fd'	$(-1.2, 1.0)$	77	0.00246
'fd'	$(1.0, 1.0)$	46	0.00150
'analytical'	$(-1.2, 1.0)$	33	0.00115
'analytical'	$(1.0, 1.0)$	22	0.000749

Table 5: Constrained minimization with different initial guesses

Once again it is possible to confirm the importance of previously knowing the analytical expression for the partial derivatives. One conclusion that as already been stated in past reports and confirmed here again is that the partial derivatives, when known functions, allow the optimization to reduce CPU time and function calls. In more complex problems, it might be a crucial help to reduce CPU power and time.

2 Analysis Models in *AeroStruct*

The aerodynamics model of *OpenAeroStruct* uses a vortex lattice method (VLM) to compute the aerodynamic loads acting on each defined lifting surfaces. A flowchart diagram is shown in figure 3. The variables are shown as green ellipses with a dashed line and the python functions are shown in rectangles. The arrows represent the data exchange between functions (inputs and outputs). To start the user needs to inform the number of lifting surfaces to be incorporated. Each lifting surface has a corresponding mesh that the user also needs to define. This first mesh is related to the geometry of the lifting surface, e.g. it coincides with the leading edge, trailing edge, root and tip of the surface. The function *collocation_points.py* will provide information regarding the location of collocation points, force points and bound points. Bound points are the points that connect the bound vortex. In order to compute the circulations it is required to get the velocity at each of the collocation points, the AIC matrix and the normal vector to each panel. The *geometry.py* function provides the normal vector to each panel, calculated with the cross product of the two diagonals. The velocity is obtained with both the *freestream_velocities* variable (that depends on flow conditions such as α, V_∞, rotational velocities) and the velocity vector that also contains geometric information for each panel in the collocation points. The function *mtx_rhs.py* is responsible for calculating the AIC matrix and the right hand side of equation 2. After having calculated the circulation one need to decompose them into the horseshoe circulations, and that is the objective of function *horseshoe_circulation.py*. Since the induced velocities need to be computed in the force points, the function *get_vectors.py* is used again getting this time as input the force points. The induced velocities in force points are then calculated recurring to *eval_mtx.py* and *eval_velocities.py*. Having already the horseshoe circulations, the velocity on each force point, which is the sum of the induced velocity with the freestream velocity) and knowing the location of the bound points, function *panel_forces.py* will be able to calculate the force on each panel based in equation 3.

$$\boldsymbol{A}\Gamma = -V_\infty \cdot n \tag{2}$$

$$\boldsymbol{F}_i = \rho\Gamma_i(\boldsymbol{V_\infty} + \boldsymbol{v_i}) \times \boldsymbol{l}_i \tag{3}$$

Once having the forces obtained for each panel it is necessary to rearrange the information, associating each panel to the corresponding surface. The lift and drag coefficients are therefore calculated. If one wants to consider viscous flow the drag coefficient can also be calculated modified after defining the Mach and Reynolds number. Wind contributions were ignored in order to keep simplicity of the flowchart diagram.

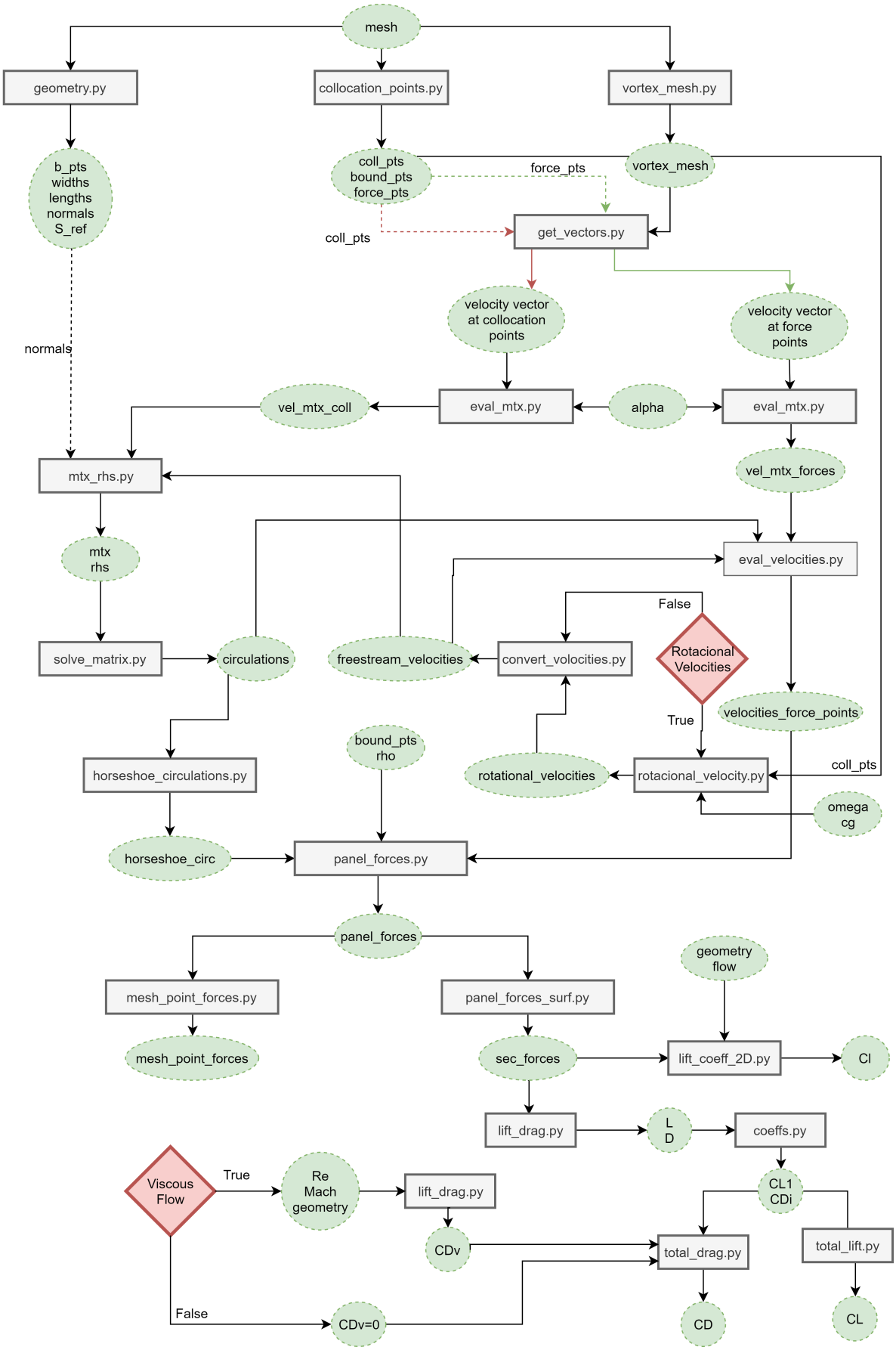

Figure 3: Flowchart representing the method to compute forces using VLM theory

3 Aerodynamic Optimization using *OpenAeroStruct*

3.1 Optimal solution - invicid flow

The theoretically optimum solution is the famous elliptical distribution for lift which causes the least induced drag. However, this lift distribution shape has some disadvantages such as the fact that the wing is subjected to stall simultaneously over the span and the difficulty of producing such surfaces. *OpenAeroStruct* will try to converge to an elliptical lift distribution if one intends to reduce drag and no constraints are provided as confirmed in the next subsections.

3.2 Solutions obtained for the optimization problem

In this subsection, results are shown for the optimization problem defined in equation 4. The design variables will be altered in between optimization problems in order to have a better understanding of the program functioning as well as constraint conditions. In order to be able to compare results one additional constraint was added. This constraint is in respect with the wing twist at the root, which was set to a value of zero. The results shown on the following table are obtained for a wing mesh with 144 panels, corresponding to a division in the span wise direction of 25 and in the chord direction of 7.

$$\begin{cases} minimize \qquad C_D \\ w.r.t. \qquad \alpha, \gamma(y), c(y) \\ C_L = C_L^* \\ C_l \leq C_{l_{max}} \\ S_{ref} = S_{ref}^* \end{cases} \tag{4}$$

D.Var.	Constraints	α^*	$c(y)^*$	$\gamma(y)^*$	C_D	Iter.	FEval	GradEval
α	C_L,C_l	3.06	$(1.47, 1.47, 1.47)$	$(0, 0, 0, 0, 0)$	0.005	4	4	4
α,γ	C_L,C_l	3.45	$(1.47, 1.47, 1.47)$	$(-1.5, -0.9, -0.18, -0.02, 0)$	0.005	16	17	16
α,c	C_L,S_{ref}	3.05	$(0.71, 2.26, 1.88)$	$(0, 0, 0, 0, 0)$	0.005	12	13	12
α,γ,c	C_L,S_{ref},C_l	2.20	$(0.10, 0.8, 3.9)$	$(10, 3.3, 1.2, -0.14, 0)$	0.005	52	58	52

Table 6: Optimization problem with different design variables

In order to have graphic visualisation of the solution *SqliteRecorder* is used and it is possible to represent the following figures. The script *plot_wing* had also to be altered in order to show the desired plots.

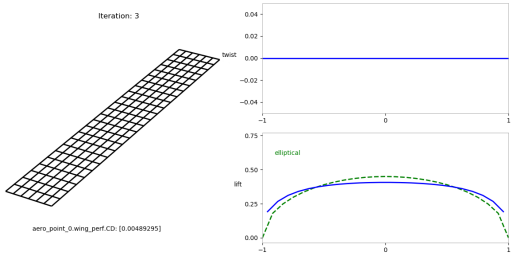

Figure 4: α as design variable

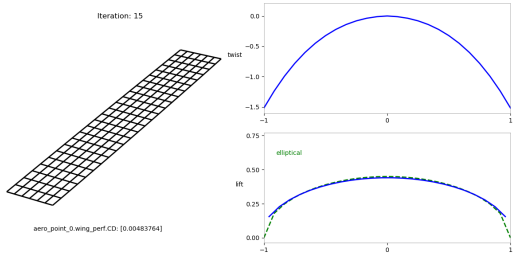

Figure 5: α and γ as design variable

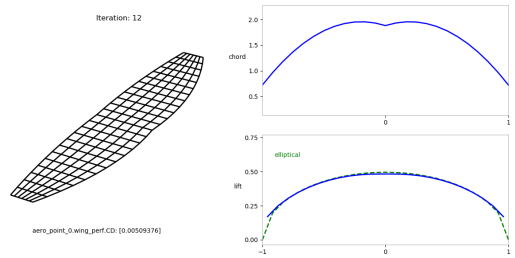

Figure 6: α and c as design variable

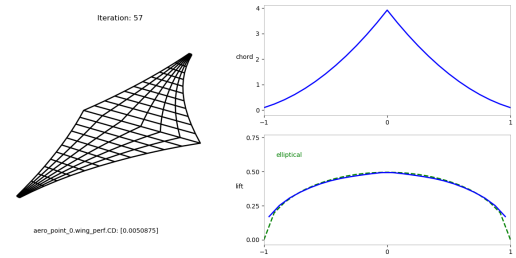

Figure 7: α, γ and c as design variable

One can observe that the constraint defined on the root of the wing, implying that the twist at that location is zero, is respected. The drag coefficient does not show significant changes, although the best result is obtained for a rectangular wing, meaning that the design variables were angle of attack and the twist distribution. As for the number of iterations and function evaluations, as expected, the maximum number happens for the case in which the higher number of design variables and constraints occurs.

3.3 Implementation of new constraint

In this subsection a new function is defined and used to compute a new constraint for the optimization problem. The function implemented defines the Reynolds number at the trailing edge. Thus, it is possible to implement a constraint to this number, which will be directly related to the chord of the wing. The Reynolds number is calculated according to equation 5.

$$Re = \frac{\rho v c}{\mu} \tag{5}$$

The implementation of this equation requires the computation of partial derivatives. All the partial derivatives are trivial, except for the partial derivative of Re with respect to the chord. Since the chord is a vector counting the value of all the chords defined in the mesh, the Reynolds number will also be a vector and thus, the partial derivative will be a matrix of dimension $n \times n$, being n the number span wise divisions on the mesh. After the new function being defined, it is necessary to edit the files *aero_groups.py* and *functionals.py*. The first editing is really simple since it is just adding the variable viscosity as input in the VLMFunctionals group. The second is the introduction of the new *Component* to the VLMFunctionals group. After this process it is then necessary to decide on the Reynolds number to give as a constraint. The higher the Reynolds number imposed, the higher minimum cord value is

obtained. The value chosen for the Reynolds number will be 5×10^6, which means that the trailing edge will always be in a turbulent flow regime. The results obtained are shown in table 8. The exit mode is omitted due to lack of space and also because it is always zero for these experiments.

D.Var.	Constraints	α^*	$c(y)^*$	$\gamma(y)^*$	C_D	Iter.	FEval	GradEval
α	C_L,C_l, Re	3.06	$(1.47, 1.47, 1.47)$	$(0, 0, 0, 0, 0)$	0.005	4	4	4
α,γ	C_L,C_l, Re	3.45	$(1.47, 1.47, 1.47)$	$(-1.5, -0.9, -0.18, -0.02, 0)$	0.005	16	17	16
α,c	C_L,S_{ref}, Re	3.09	$(1.36, 1.36, 2.08)$	$(0, 0, 0, 0, 0)$	0.005	7	7	7
α,γ,c	C_L,S_{ref},C_l, Re	3.99	$(1.36, 2.08, 1.36)$	$(-1.8, -1.3, -1.0, -0.5, 0)$	0.005	29	29	29

Table 7: Optimization problem with different design variables

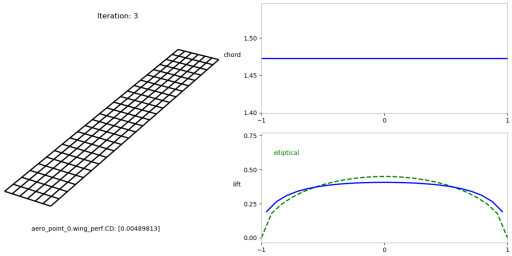

Figure 8: α as design variable

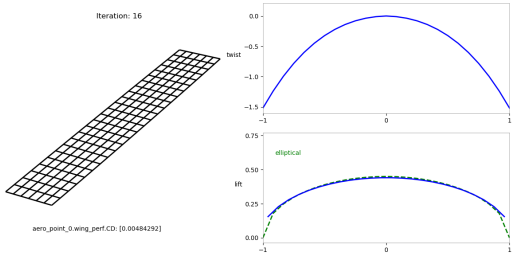

Figure 9: α and γ as design variable

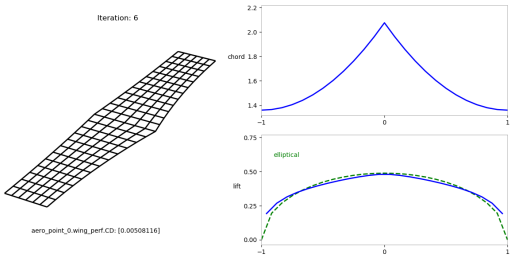

Figure 10: α and c as design variable

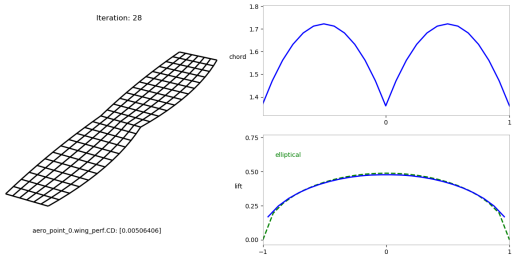

Figure 11: α, γ and c as design variable

These results show that the minimum value of chord was 1.36, which is a value a lot higher than for the values obtained when there was no constraint on the Reynolds number. As for the drag coefficient, it remains almost the same. Despite not having improvements on the drag coefficient, the wing design observed in this part is a lot more consistent, especially considering the case in which all three design variables are considered. In the first part, the chord had tip value of 0.12, which is clearly a value too small and can cause structural problems. One last analysis was made, considering viscous effects, in order to have a notion of how much the design values changed, as well as the objective function.

D.Var.	Constraints	α^*	$c(y)^*$	$\gamma(y)^*$	C_D	Iter.	FEval	GradEval
α,γ,c	C_L,S_{ref},C_l, Re	2.75	$(1.42, 1.10, 2.33)$	$(-0.6, -0.2, 1.1, -0.2, 0)$	0.012	31	51	30

Table 8: Optimization problem with different design variables

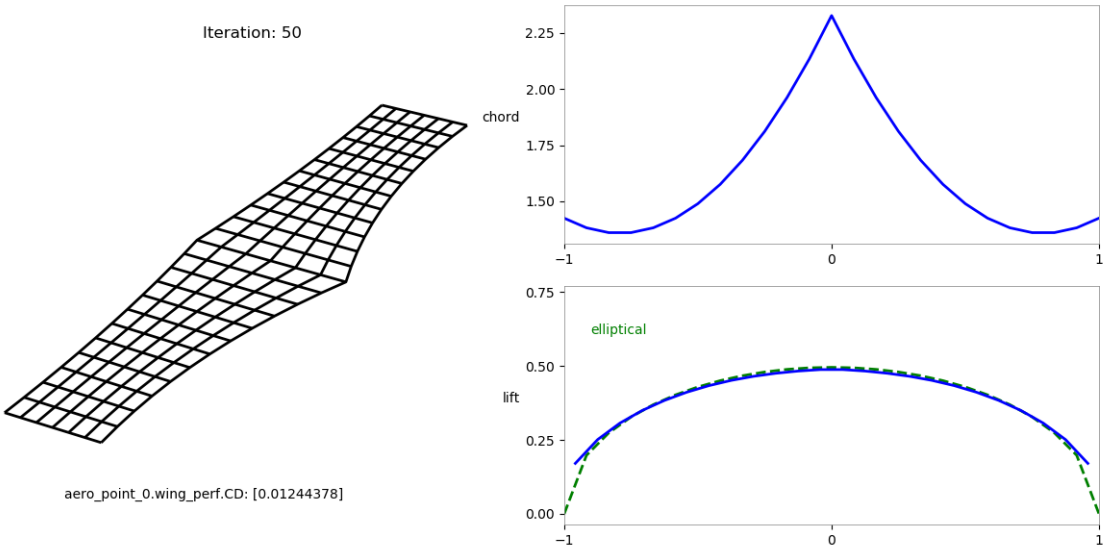

Figure 12: α, γ and c as design variable

The value of the drag coefficient, as expected, increase with the contribution from the viscous drag. The chord