49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2011, Orlando, Florida
AIAA 2011-823

Optimization of 2-D Flap Geometry Using Matlab and Fun3D

Gregory D. Howe[1]
Georgia Institute of Technology, Atlanta, GA, 30332

This paper describes work done in the process of creating a workable system for the optimization of two-element high-lift airfoil design based on a fixed "cruise configuration" baseline. Methods were developed to define airfoil flap geometry, automatically create and run unstructured computational meshes based on this geometry, and to iteratively optimize this geometry. Validation cases are presented based on different optimization algorithms and parameters. Significant work is also presented on the attempt to characterize the design space of this problem in order to better understand the performance of different optimization routines.

Nomenclature

$C_{L,\max}$	=	Maximum Lift Coefficient.
$\mathbf{P}_i$	=	Bezier Curve Control Point (as a vector)
$P_{i,x}$	=	Bezier Curve Control Point *x*-coordinate
$P_{i,y}$	=	Bezier Curve Control Point *y*-coordinate
$\mathbf{T}_\beta$	=	Unit Tangent Vector to the Curve β
α	=	Angle of Attack.
$\vec{\gamma}(t)$	=	Generic Parametric Function
ζ	=	Airfoil vertical coordinate per unit chord.
κ_β	=	Extrinsic Curvature of the Curve β
ψ	=	Airfoil chordwise coordinate per unit chord.

I. Introduction

One of the more difficult problems in aerodynamic design is that of high-lift aerodynamics. This is largely because computational fluid dynamics has only recently reached the point where reliable $C_{L,\max}$ computations can be done in a reasonably short amount of time. Accurate modeling of viscous flow phenomena is necessary not only to predict stall, but also to predict aerodynamic coefficients at angles-of-attack immediately preceding stall. Before the 1990s [(1)], the only CFD codes available to quickly compute lift coefficients near stall depended largely upon empirical methods to work the viscous part of the solution. Numerically optimizing these sorts of codes is dangerous, as the optimizer tends to get stuck in the assumptions of the empirical methods.

Today, however, CFD codes and computers have advanced to the point that true $C_{L,\max}$ optimization is possible. On a pair of 2008-vintage quad-core Intel processors, NASA's Fun3D Navier-Stokes code can run a two-dimensional airfoil in six to ten minutes. With a few of these jobs running simultaneously, a true optimization case could be run in merely a few hours. In order for this to be possible, however, a great amount of methodology needs to be developed to define airfoil geometries, create permutations, and automatically evaluate them.

A. The Philosophy of High-Lift Design

Before engaging in a process of writing a massive amount of computer code to design high-lift airfoils, a fundamental question must be answered: What are we looking for? The simple answer is that the goal is to produce as much lift as possible in the smallest wing area possible – that is, to maximize $C_{L,\max}$. Such a design would certainly be "optimized for high-lift." However, the vast majority of aircraft built today have a primary mission that involves flying quite a bit faster than their stall speed. To allow for differing takeoff and cruise geometry, the traditional idea is to build a multi-element system. This will allow the nested airfoil configuration to be optimized

[1] Undergraduate Student, Aerospace Engineering; AIAA Student Member.

for cruise conditions while still generating high lift for takeoff. Additionally, multi-element airfoils have been shown to be almost universally capable of generating more lift than single-element airfoils. A.M.O. Smith's excellent installment in the Wright Brothers Lecture series [2] goes to great lengths to prove this. While this design philosophy seems to be the obvious plan, there still remain important choices to make. One of the first is: How much should the cruise design be valued in comparison to the high-lift design.

To answer this question, it is important to understand what higher $C_{L,\max}$ accomplishes. In reality, the only effect of $C_{L,\max}$ in a transport aircraft is to improve landing and takeoff performance. This fact being known, what does improved landing and takeoff performance accomplish? Generally, these parameters are mission requirements for transport aircraft. Military transports will have requirements of certain field lengths specified by the military based on what kind of airfields the purchasing organization wishes to be able to use. For civilian aircraft, the specification is similar, but generally driven by marketing concerns. The important point to note about both of these processes is that there is no smooth correlation between landing or takeoff performance and the potential profitability of an aircraft. A marketing study will find that a certain number of customers would buy the aircraft if it had a 6,000 ft. balanced field length. There will likely be a different, more profitable, number for a 5,000 ft. field length. It is very unlikely that the number for a 5,999 ft. field will be at all different from the 6,000 ft. case, as there are no airports out there with exactly 5,999 ft. runways. Therefore, directly optimizing $C_{L,\max}$ for a complete aircraft planform is not likely to be the best goal in the long view. Similarly, minimizing drag for a given $C_{L,\max}$ is just altering a different term in the field length equations (not to mention that increased drag actually *helps* landing performance).

Ideally, landing and takeoff $C_{L,\max}$ is not a design goal at all, but rather a constraint to cruise design. The true, "big picture" way to design a wing will be to optimize the shape to reduce cruise drag for a given cruise lift with a *constraint* of maximum field length. A trade study could be done by running the optimization with multiple field length constraints. This would produce a curve relating maximum range to field length. Marketing data would then be able to theorize about which point on that curve would sell the most aircraft.

That being the goal, a process is needed to determine whether or not a given planform can achieve a particular field length. An optimization sub-problem is set up where the cruise outer mold line is held constant and a flap is carved out to produce the minimum field lengths possible (with a variety of other constraints involved in this process). If these values are within the prescribed limits, then the cruise planform is a valid solution to the design problem. This then returns to the need for direct maximization of $C_{L,\max}$, though perhaps with different constraints in mind than previously.

II. Objectives

So, despite the fact that larger concerns would dictate that $C_{L,\max}$ should not be directly optimized for a whole aircraft, there is a definite place for a routine to simply maximize $C_{L,\max}$ for a fixed cruise wing. The processes created here will thus receive a predefined cruise planform (or airfoil, in the 2-D sense) and leave it unmodified throughout. In a whole-aircraft design optimization, this would be a subroutine of a wing design program attempting to maximize cruise performance. This subroutine would allow the larger design program to decide whether or not the considered geometry met the minimum takeoff and landing requirements of the design specification.

In this process, flap geometry will be created such that the trailing edge of the flap is coincident with that of the main element. The leading edge geometry of the flap as well as the location and angle of the deflected flap will be optimized to increase $C_{L,\max}$ within certain constraints. For the time being, this airfoil $C_{L,\max}$ value will be considered the primary goal.

There are several intermediate objectives that must be completed in order to have a reliable scheme for the optimization of flap $C_{L,\max}$. The major components include:

- Specification of airfoil and flap geometry
- Automation of airfoil grid generation
- Choice (and possibly design) of one or more optimization algorithms.

III. Approach

A. Airfoil/Flap Geometry Specification

In order to properly optimize flap geometry, it is extremely important to have a quality geometry definition scheme. This is as much a part of the formulation of the optimization problem as the choice of optimization algorithm and CFD solver. If the geometric methodology allows for the existence of badly-conceived cases, the optimizer will waste time evaluating them where a human designer would discard them first glance (for example,

think of carving out a flap by making a single straight cut perpendicular to the chord line). On the other hand, not allowing enough freedom in the alteration of the geometry could prevent the optimizer from finding the best case.

First of all, the definition must preserve the original airfoil contour. Specifically, this means that the trailing edge of the nested flap must be coincident with the trailing edge of the original airfoil in order to preserve cruise performance. For the moment, this will be done more like the way flapped wind tunnel models are constructed than the way production aircraft are built (the difference primarily being in the cove region). Imagine taking a block of wood carved into the shape of the cruise airfoil and cutting out a flap with a single cut of a band saw. The "cutting line" through the airfoil defines both the leading edge of the flap and the cove geometry of the main element.

Without getting into the specifics of how the airfoil itself was created, it can be assumed that each surface (upper and lower) is described as a "$\zeta(\psi)$" function, where ψ is the x-coordinate (the chordwise direction) of the airfoil divided by the chord length and ζ is the z-coordinate (the depthwise direction) divided by the chord length. We then define $\psi_{u,T}$ as the x/c location of where the cutting line begins on the upper surface. $\zeta_u(\psi_{u,T})$ then identifies the z/c location of that point. $\psi_{l,T}$ and $\zeta_l(\psi_{l,T})$ represent the location of where the cutting line exits on the lower surface. The specific definition of the clean airfoil will not be discussed here, but Brenda Kulfan's "class-shape transform" methodology has proven to be very effective. [3]

Next define a curve $\vec{\gamma}(t)$ as representing the leading edge of the flap where the endpoints are coincident with points on the aft portion of the original airfoil. Several constraints will be placed on this curve. In order to have a smooth geometry, the goal is to find a $\vec{\gamma}(t)$ that is identical to $\zeta(\psi)$ at these two contact points. The Fundamental Theorem of Curves from differential geometry says that two plane curves are identical up to position and rotation if their curvature is the same. [4] Thus, $\vec{\gamma}(t)$ should be defined such that the curvature, value (position), and unit tangent vector (rotation, up to a sign) are the same as those of the cruise airfoil at the contact points.

This curve clearly cannot be a single $y(x)$ function, as it has to make a loop roughly similar to the left half of a circle (i.e. it will fail the "vertical line test"). The two options are then whether to describe it as a pair of $y(x)$ functions, like Brenda Kulfan did for her Class-Shape Transform airfoils, or to use a parametric $(x(t),y(t))$ function. Both approaches are likely viable, though parametric curves seem easier. Parametric Bezier curves were chosen here because, as will be shown below, their position, tangent vector, and curvature at their two endpoints can all be written in relatively simple forms that are actually completely independent of the other endpoint (for sufficiently high-order curves).

1. Differential Geometry of Bezier Curves

A Bezier curve of order n in dimension k is a parametric curve defined by $n+1$ control points $\mathbf{P}_i \in \Re^k$ on the closed interval $t \in [0,1]$, i=0, 1, 2, ... n:

$$\vec{\gamma}(t) = \sum_{i=0}^{n} B_{n,i}(t)\mathbf{P}_i \quad \text{where} \quad B_{n,i}(t) = \frac{n!}{i!(n-1)!} t^i (1-t)^{n-i} \tag{1}$$

Without belaboring the derivation, simple formulas can be found for the position, unit tangent vector, and curvature at the two endpoints of a general Bezier curve:

$$\vec{\gamma}(0) = \mathbf{P}_0 \tag{2}$$

$$\vec{\gamma}(1) = \mathbf{P}_n \tag{3}$$

$$\mathbf{T}_\alpha(0) = \frac{\mathbf{P}_1 - \mathbf{P}_0}{|\mathbf{P}_1 - \mathbf{P}_0|}, \ \mathbf{T}_\alpha \in \Re^n \tag{4}$$

$$\mathbf{T}_\alpha(1) = \frac{\mathbf{P}_n - \mathbf{P}_{n-1}}{|\mathbf{P}_n - \mathbf{P}_{n-1}|}, \ \mathbf{T}_\alpha \in \Re^n \tag{5}$$

$$\kappa_\alpha(0) = |\mathbf{P}_2 - 2\mathbf{P}_1 + \mathbf{P}_0|,\ \kappa_\alpha \in \Re \tag{6}$$

$$\kappa_\alpha(1) = |\mathbf{P}_n - 2\mathbf{P}_{n-1} + \mathbf{P}_{n-2}|,\ \kappa_\alpha \in \Re \tag{7}$$

2. Bezier Curve Boundary Conditions

Now the desired boundary conditions can be enforced. First is to require that the Bezier curve contacts a different curve at each endpoint: $y_u(x)$ at t=0 and $y_l(x)$ at t=1. Additionally, at both of these points the Bezier and intersecting curves must have the same unit tangent vector and the same curvature. Note that for this portion of the derivation, the Bezier curve is assumed to be two-dimensional, which means that the control points $\mathbf{P}_i$ can be defined by two numbers, $P_{i,x}$ and $P_{i,y}$.

From Eq. $\vec{\gamma}(0) = \mathbf{P}_0$ (2) and Eq. $\vec{\gamma}(1) = \mathbf{P}_n$ (3), directly assume that $\mathbf{P}_0$ and $\mathbf{P}_n$ are known:

$$(x_u, y_u(x_u)) = \vec{\gamma}(0) = \mathbf{P}_0 \tag{8}$$

$$(x_l, y_l(x_l)) = \vec{\gamma}(1) = \mathbf{P}_n \tag{9}$$

Since the curve $y(x)$ has partial derivatives of $x' = 1$ and $y' = \frac{\partial y}{\partial x}$, $\mathbf{P}_1$ and $\mathbf{P}_{n-1}$ are then partially determined by the slopes of $y_u(x)$ and $y_l(x)$ at the intersection (solved for $P_{1,y}$ and $P_{n-1,y}$ for future usage):

$$P_{1,y} = P_{0,y} + (P_{1,x} - P_{0,x})\frac{\partial y_u}{\partial x} \tag{10}$$

$$P_{n-1,y} = P_{n,y} + (P_{n-1,x} - P_{n,x})\frac{\partial y_l}{\partial x} \tag{11}$$

The general equation for curvature can be simplified for a two-dimensional curve, where x and y are each parametrically defined coordinates:

$$\kappa = \frac{|x'y'' - y'x''|}{\left[(x')^2 + (y')^2\right]^{3/2}} \tag{12}$$

When $y=f(x)$, the Eq. (12) simplifies:

$$\kappa = \frac{|y''|}{\left[1 + (y')^2\right]^{3/2}} \tag{13}$$

So, the curvature of the intersecting curves can be calculated (squared for future simplicity):

$$(\kappa_{y_u})^2 = \frac{\left(\frac{\partial^2 y_u}{\partial x^2}\right)^2}{\left(1 + \left(\frac{\partial y_u}{\partial x}\right)^2\right)^3} \tag{14}$$

This expression can be combined with that of the square of the curvature of the Bezier curve:

$$\left(P_{2,x}-2P_{1,x}+P_{0,x}\right)^2+\left(P_{2,y}-2P_{1,y}+P_{0,y}\right)^2=\frac{\left(\frac{\partial^2 y_u}{\partial x^2}\right)^2}{\left(1+\left(\frac{\partial y_u}{\partial x}\right)^2\right)^3} \tag{15}$$

$$\left(P_{n,x}-2P_{n-1,x}+P_{n-2,x}\right)^2+\left(P_{n,y}-2P_{n-1,y}+P_{n-2,y}\right)^2=\frac{\left(\frac{\partial^2 y_l}{\partial x^2}\right)^2}{\left(1+\left(\frac{\partial y_l}{\partial x}\right)^2\right)^3} \tag{16}$$

Now substitute the expressions for $P_{1,y}$ and $P_{n\text{-}1,y}$ developed in Eq. (10) and Eq. (11) above:

$$\left(P_{2,x}-2P_{1,x}+P_{0,x}\right)^2+\left(P_{2,y}-2\left(P_{1,x}-P_{0,x}\right)\frac{\partial y_u}{\partial x}-P_{0,y}\right)^2=\frac{\left(\frac{\partial^2 y_u}{\partial x^2}\right)^2}{\left(1+\left(\frac{\partial y_u}{\partial x}\right)^2\right)^3} \tag{17}$$

$$\left(P_{n,x}-2P_{n-1,x}+P_{n-2,x}\right)^2+\left(-P_{n,y}-2\left(P_{n-1,x}-P_{n,x}\right)\frac{\partial y_l}{\partial x}+P_{n-2,y}\right)^2=\frac{\left(\frac{\partial^2 y_l}{\partial x^2}\right)^2}{\left(1+\left(\frac{\partial y_l}{\partial x}\right)^2\right)^3} \tag{18}$$

These equation can be solved for the value of $P_{1,x}$ or $P_{n\text{-}1,x}$ given a known $\mathbf{P}_2$ or $\mathbf{P}_{n\text{-}2}$. Analytical solutions to these equations are extremely difficult (if in fact they exist), but a computer will have no difficulty solving them numerically. Simply subtract one side from both to get an expression that is equal to zero and then ask a computer routine to find the zeros of that expression by varying $P_{1,x}$ or $P_{n\text{-}1,x}$. These values can then be inserted into Eq. (10) and Eq. (11) to find the corresponding $P_{1,y}$ and $P_{n\text{-}1,y}$ values.

3. Application to Flap Geometry

Using the above computations, it is now possible to define the leading edge of a flap with a 4th-order Bezier curve using only four control variables:

- The *x/c* of contact with the upper airfoil surface.
- The *x/c* of contact with the lower airfoil surface.
- The *x/c* location of the point $\mathbf{P}_2$
- The *z/c* location of the point $\mathbf{P}_2$

Adding just three more variables produces deflected flap geometry: the ball gap, overlap, and angle. For this part, the "cove trailing edge point" is defined as the furthest aft point on the lower side of the upper cove (i.e. the point on the main element that should be closest to the upper surface of the flap).

- Ball gap separation is defined as the distance between the cove trailing edge point and the nearest point on the flap.
- Overlap is defined as the *x*/*c* location of the cove trailing edge point minus the x/c location of the furthest forward point on the flap (i.e. smallest *x*/*c* value).
- Angle is defined as the clockwise rotation of the flap element from its nested position.

At last, the method will fully define a flap geometry given a base cruise airfoil. Figure 1 shows an RAE2882 Airfoil with a flap created with this method. Figure 2 shows a detail view of the cove area.

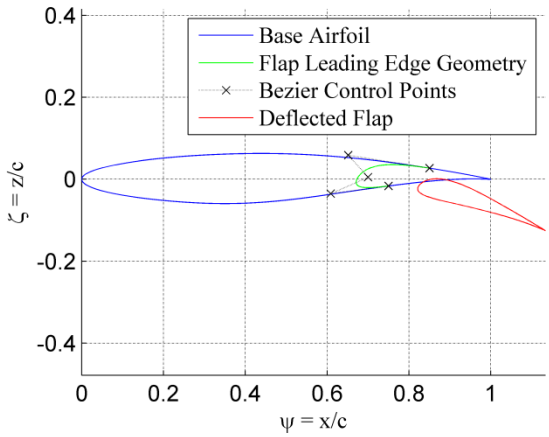

Figure 1. RAE2822 with Flap

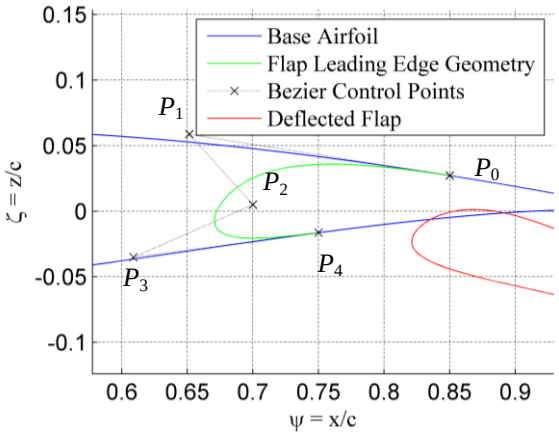

Figure 2. RAE2822 Flap, Cove Zoom

4. Cove Geometry

The methodology used to define the airfoil cove (i.e. the "slot" in the main element that the flap fits into when nest) was fairly simple. As mentioned before, the shape was simply assumed to be the same as that of the leading edge of the flap. The small problem is that this alone would lead to two cusped trailing edges on the airfoil main element. This may look attractive, but it is physically impossible to build. Thus, a minimum thickness is specified. The program starts at the leading edge of the nested flap and follows the upper surface until the distance between the flap leading edge and the main airfoil upper surface is equal to the specified minimum thickness. The upper trailing edge of the main element is truncated at this point. The process is repeated for the lower side.

5. Applied Constraints

In order to be sure that the resulting flap geometries are realizable, several constraints are checked when cases are generated. These are:

- The minimum *x*/*c* point cannot impinge upon a specified rear spar location.
- A minimum ball gap value of 0.008. This is purely a mesh-related constraint. Otherwise, the minimum value would be 0.
- The lower trailing edge of the main element cannot intersect the deflected flap.

Any geometries failing these conditions will not be run by the solver and are ignored in an optimization procedure.

B. Automatic Unstructured Grid Generation in Matlab

Unstructured meshes are generated within Matlab in a three-step process. First a boundary layer mesh in generated around each constituent element by placing grid points individually along normal vectors from the surface. The second step uses an automatic gridding process [5] to generate a field of points between the boundary layer grid and the farfield boundary. Finally, a Delaunay triangulation algorithm creates connectivity information between all of the generated points. Figure 3 shows a zoomed-in view of an airfoil trailing edge gridded with this methodology. The difference between the "boundary layer" and "farfield" grids is clearly evident.

While this gridding system has flexibility in parameters like surface point distribution, much of this information is fixed in the optimizer-generated cases. 150 grid points are distributed along the upper and lower surfaces of the cruise airfoil utilizing a typical "cosine spacing" that clusters points near the leading and trailing edges. These points are eventually split between the main element and the flap. 60 points are similarly spaced out along the arclength of the cove. The leading edge of the flap is defined by 80 points in a "tangent spacing" that clusters points near the center of the distribution (i.e. near the leading edge point of the flap). 5 points are placed along the blunt face of each of the main element's trailing edges.

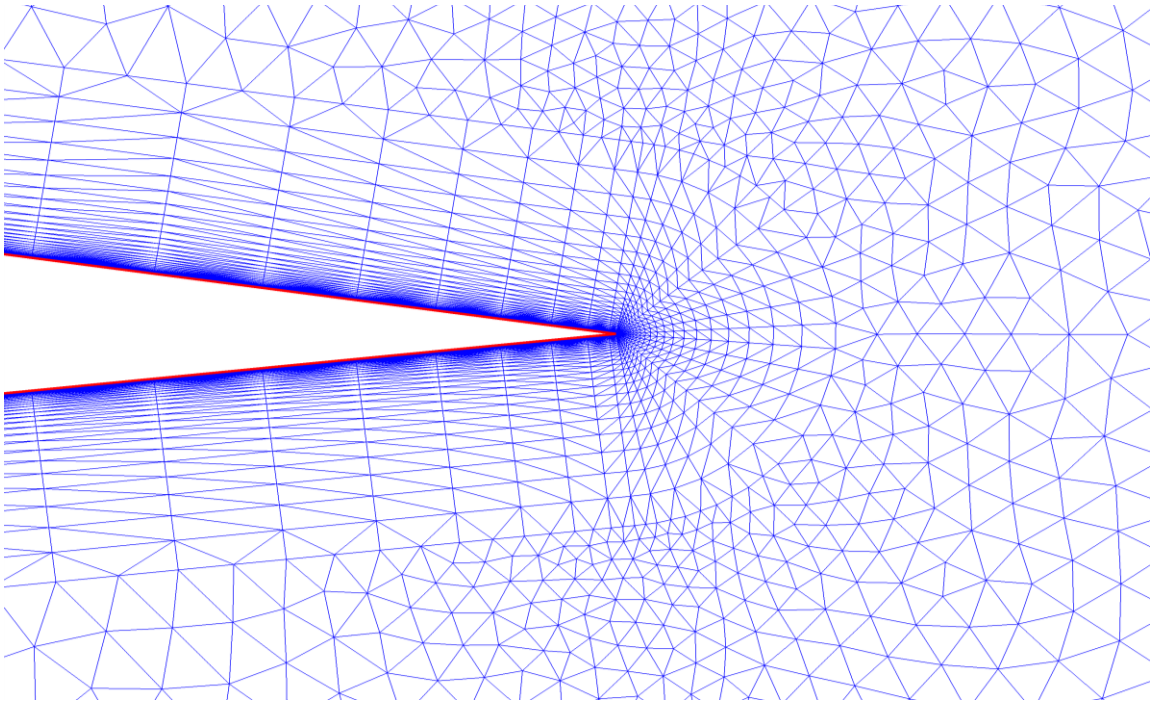

Figure 3. Airfoil Trailing Edge Showing Boundary Layer and Farfield Grids

C. Fun3D - "Fully Unstructured Navier-Stokes"

Fun3D is an unstructured Navier-Stokes CFD solver written and maintained by a group at NASA Langley. [6] Fun3D makes an ideal solver candidate for this type of problem for a number of reasons. Firstly, it is based upon Fun2D, which was one of the first 2-D fully-unstructured Navier-Stokes flow solvers. Much of the original use of Fun2D was in high-lift analysis and it showed extremely good results in that area. [7] This legacy means that Fun3D has an increased level of support for 2-D computations compared to many modern CFD codes. Enabling 2-D mode in Fun3D causes the solver to actually force the solution to be two-dimensional. That is, it artificially prevents any gradients or velocities from appearing in the span-wise direction. Many other modern 3-D solvers do not directly support 2-D computations, so the user is left the task of creating a slice of an infinite wing and hoping that there are no asymmetries in the mesh that might cause three-dimensional flow phenomena to appear.

An additional major advantage of Fun3D is the high standards with which it is coded. The Fun3D team claims to have uncovered bugs in virtually every Fortran 95 compiler in existence because of the quality of their code. This makes the code easier to get compiled (assuming a compiler that has those bugs fixed). It also aids Fun3D in being considerably faster than many other unstructured Navier-Stokes codes. In the anecdotal experience of the author, for example, Fun3D runs significantly faster than USM3D.

Though space is too limited to show it here, significant work has been done validating Fun3d and the previously described grid generation methodology to wind tunnel data for flapped airfoils. Though there are still some discrepancies, agreement has generally been very good and many of these discrepancies can be explained by the realities of wind tunnel testing and the inherent difficulty of reproducing exactly the circumstances of the test. Included in this testing was an attempt to chose among the various turbulence models available in Fun3d. The ultimate result was that the Spalart-Allmaras (S-A) model was chosen to be the most accurate for flap $C_{L,\max}$ calculations. This model performed better in tests than the supposedly "more advanced" Menter-SST model. The reason for this is currently a mystery, but the result was consistent enough that all of the Fun3d results seen in this paper use the S-A model.

D. Optimization Algorithms

If optimization is to be done, then obviously algorithms need to be investigated. The major consideration here that is different from many of the typical optimization problems seen today is that of parallelization. The Fun3D code is fully setup to run in a message-passing interface (MPI) parallelization scheme to allow one solution to run on multiple processors. However, this is not sufficient to be the only level of parallelization present. It takes on the order of 30-60 seconds to generate and partition a mesh to run a case. This currently has to be done on a single

machine (as much because of Matlab liscensing issues as any technical reason). Therefore, it becomes a hard wall of the minimum time necessary to run a single case. Running Fun3D with 400 processors might yield a flow solution in 10 seconds, but it would be unproductive because the optimizer then has to wait for the next grid. Additionally, the runtime of an MPI code does not scale linearly with the number of processors used. Eventually, the amount of time that cluster nodes spend communicating with each other over the network becomes the majority of the time of the run. Fun3D has been found to run extremely efficiently in a "shared-memory" situation. This is when the MPI communication between processors is taking place within a single motherboard rather than involving network hardware.

Therefore, the best usage of computational resources will be running many cases simulatneously. That is, every Fun3D run will be using, say, 8 processors (two quad-core chips on a single motherboard), but the optimizer will be running 10 or 15 jobs at once. The problem is that this requires that an optimization algorithm be chosen that can make use of this kind of parallelization. Two of these are explained below. Note that these explanations will refer to the number of simultaneous processes being run. Each of these processes will usually be run on multiple processors.

1. Parallel Gradient Search

This is a slight modification on the "classical" gradient-based method. The simplest description is that steps are taken that are proportional to the negative of the gradient of the function at the current point. Gradient-based methods can be somewhat unstable in the CFD environment because of the rugged nature of the objective function. The high amount of noise in CFD results causes there to be many "false minima" that a gradient-based method has no way to ignore. There is also the problem of merely computing the gradient of the objective function. Because of the discontinuous nature of the solution space, gradients must be calculated by methods of finite differences.

The first part of the gradient search is parallelized simply by calculating all of the finite difference derivatives at the same time. Therefore, when optimizing in *n*-space (and computing two-sided derivatives), there will be up to 2**n* points that can be evaluated at once. The number of processes used, *P*, should be chosen to be an integer factor of 2**n* to avoid idle CPU time. A sticking point here is that the gradient algorithm then needs to evaluate the fitness of its new location, a step that would leave much of the available computer resources idle. There is also the problem that a gradient optimization is awfully dependent on that point meeting the constraints of the optimization. If these constraints are encountered frequently, the optimizer can hang for quite some time with one process trying to find a point that it likes before it can move on to the next iteration. Thus, the algorithm instead test *P* points distributed evenly along the gradient vector. After points not meeting constraints are thrown away, the best of the remaining points is chosen as the center point for the next iteration.

The gradient descent search algorithm operates on an *n*-dimensional ordinate vector $\mathbf{X}$ with objective function $y(\mathbf{X})$ and finite difference interval d_{diff} as follows:

- For i=[1,n]: $\mathbf{X}_{i,1} = \mathbf{X} + \hat{x}_i d_{diff}$ $\mathbf{X}_{i,2} = \mathbf{X} - \hat{x}_i d_{diff}$ where $\hat{x}_i$ is the unit vector in the i-th dimension
- Evaluate $y(\mathbf{X}_{i,j})$ for i=[1,n] and j=[1,2]
- For i=[1,n]: $y'_i = \dfrac{y(\mathbf{X}_{i,2}) - y(\mathbf{X}_{i,1})}{2d_{diff}}$
- For k=[1,P]: $\mathbf{X}_k = \mathbf{X} - y' d_{diff}$
- Select the new $\mathbf{X}$ as the minimum of $y(\mathbf{X}_k)$

2. Particle Swarm Optimization

In particle swarm optimization, a large number of "particles" are modeled in *n*-dimensional space. These particles move through space at every iteration based on an inertial velocity from previous iterations and a random linear combination of the vector to their own previous best location and the global best location. One of the largest advantages of this method is that no gradients or derivatives are calculated at any time during evaluation.

The PSO algorithm lends itself extremely well to parallelization, since there are many function evaluations per iteration that are all completely independent of each other (they are all based on the previous iteration). If *P* processes are to be run simultaneously, the obvious choice would be to run the algorithm with k=P particles. Caution must be taken, however, because if the number of particles is too low, space is not densely populated enough to find minima. So, it is instead better to simply require that k is an integer multiple of P. If this is not the case, some computer resources will have idle time.

The particle swarm optimization algorithm operates on k n-dimensional ordinate vectors $\mathbf{X}_i$ with velocities $\mathbf{v}_i$ and objective function $y(\mathbf{X})$ as follows. $\hat{\mathbf{X}}$ and $\hat{\mathbf{g}}$ represent the local and global best ordinate vectors, respectively:

- For each particle $1 \le i \le k$:
  - Create random vectors $\mathbf{r}_1, \mathbf{r}_2$: $\mathbf{r}_1, \mathbf{r}_2 \in [0,1]$
  - Update the particle velocities: $\mathbf{v}_i \leftarrow \omega \mathbf{v}_i + c_1 \mathbf{r}_1 \circ \left(\hat{\mathbf{X}}_i - \mathbf{X}_i\right) + c_2 \mathbf{r}_2 \circ \left(\hat{\mathbf{g}} - \mathbf{X}_i\right)$
  - Update the particle positions: $\mathbf{X}_i \leftarrow \mathbf{X}_i + \mathbf{v}_i$
  - Update local bests: If $y(\mathbf{X}_i) < y(\hat{\mathbf{X}}_i)$, $\hat{\mathbf{X}}_i \leftarrow \mathbf{X}_i$
  - Update global best: If $y(\mathbf{X}_i) < y(\hat{\mathbf{g}})$, $\hat{\mathbf{g}} \leftarrow \mathbf{X}_i$

Where ω, c_1, and c_2 are constants specified by the user that represent the weightings given the inertia of the particle and the attraction to the local and global bests, respectively.

IV. Results

A. Preliminary Optimization Results

Several tests were run with both the Particle Swarm and Gradient algorithms. Both algorithms were started at the same point which was chosen essentially arbitrarily with the exception that the ball gap distance was intentionally chosen to be larger than ideal. The angle of the flap was fixed for all of these optimizations, since increasing flap angle to get higher C_L was considered a trivial solution. While the ultimate goal is an increase of $C_{L,\max}$, the process of actually *finding* $C_{L,\max}$ for every case run was deemed too computationally intensive, especially for a proof-of-concept test like this. Therefore, the decision was made to simply run all cases at a fixed angle-of-attack and optimize C_L at that condition. This value was arbitrarily chosen as 8° for the tests seen here. All tests were conducted at a Mach number of 0.3 and a Reynolds number of 6×10^6.

The time history of one case from both optimizers can be seen in Fig. 4. In these cases, the optimizer was allowed to run for 5 hours (18,000 seconds) with 8 simultaneous processes running on 8 processors each. The interesting result that can be seen is the fact that both histories are dominated by a single huge jump in C_L. Other than this single fantastic iteration, both methods seem to be making only extremely small improvements to the geometry. The particle swarm method finds this discontinuous jump earlier because it's "search radius" per iteration is larger. In general, particle swarm optimizers seem to function better when they are initially allowed to examine a large area around them, while gradient optimizers do better when their finite difference step size is set very small, making the partial derivative calculations more accurate. The gradient method may actually be taking as large of steps as the particle swarm, but it is doing so based on information gathered from a smaller region around the current point. When the significant improvement is as discontinuous as it seems to be in this case, the gradient method has to "blunder into" the improvement.

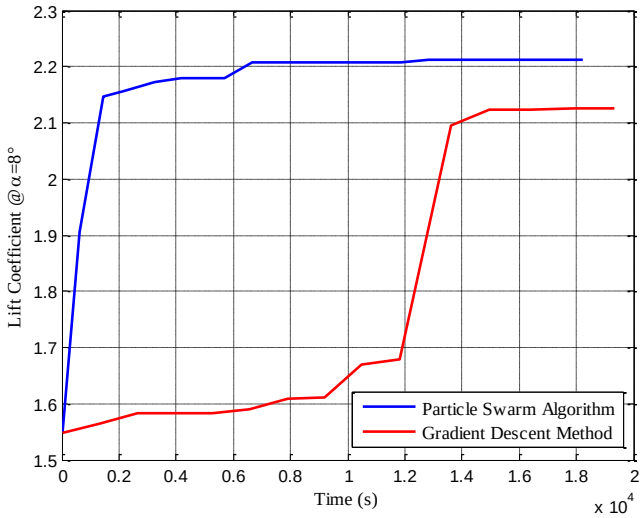

Figure 4 - Preliminary Optimization Results

B. Solution Space Investigation

With all of this effort to set up an optimization problem that appears well-formulated, the immediate instinct was to code it all up and run it. Inevitably, problems arose, many of them simple program bugs. Over time these were fixed, but problem persisted. Additionally, the optimizers exhibited somewhat odd behavior, as seen above, where they would have one large jump in solution quality but otherwise improve the lift rather slowly. In order to be sure that the optimization problem was as well-posed as expected, a study of the solution space was conducted. Essentially, the goal here was to attempt to understand whether remaining problems were due to bugs or badly

designed code or if they were simply inherent to the problem. Large blocks of cases were run in which a pair of variables in the flap geometry specification were varied linearly across a defined range. The lift coefficient at each of these cases was evaluated.

The significant test conducted was simply to "slide" a flap around by modifying the ball gap and overlap values without changing the shape or angle of the flap. These cases were run on an RAE2822 airfoil at Mach 0.3, a Reynolds number of $6x10^6$, an angle-of-attack of 8°, and a flap deflection of 20°. The other design parameters (upper and lower surface contact locations, $P_{2,x}$ and $P_{2,y}$ from III.A.3) were fixed at 0.85, 0.75, 0.75, and 0.002, respectively.

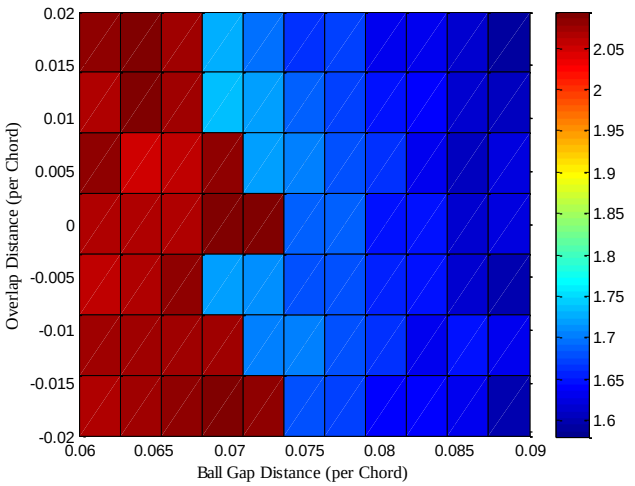

Figure 5. C_L Contours with Varying Gap and Overlap

The results are shown in Fig. 5. Note the extremely sharp contrast between C_L values. No solutions were found between values of 1.738 and 2.052. Clearly this corresponds to the sharp jumps seen in the optimization results. Closer examination of two of these points reveals the difference. Mach Number contours and streamlines around the flap region of these two cases are seen Fig. 6 and 7. Both cases had a ball gap separation of 0.0682. The extreme increase in lift is clearly caused by the attachment of flow through the slot and over the flap. When the flap is too far away from the main element or the flap is too far forward of the trailing edge, the local flow over its upper surface is dominated by the freestream instead of the wake of the main element. The altered flow in the wake is what allows flow over a body at an extreme angle-of-attack like 20° to remain attached.

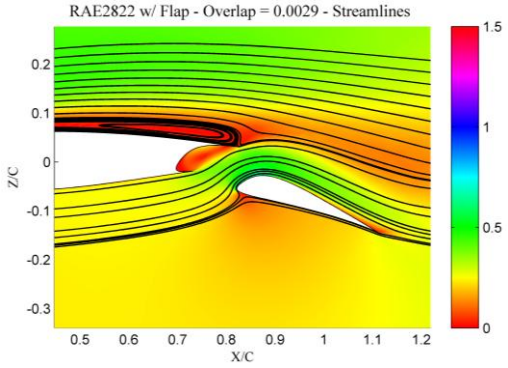

Figure 6. Case with C_L = 2.08

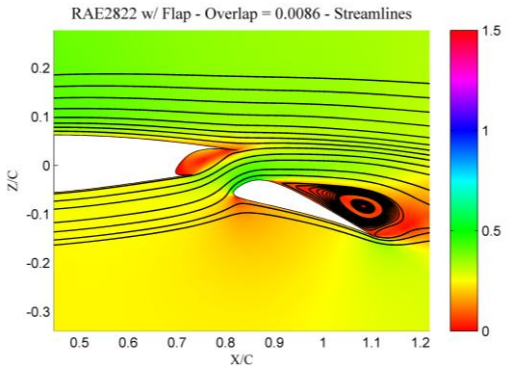

Figure 7. Case with C_L = 1.7377

Another extremely interesting takeaway from these figures is the fact that on the *higher* lift case, there was a region of separated flow on the upper surface of the main element. This represents a small drop in the lift created by the main element that is more than compensated for by the lift created by the flap. After closer examination of many cases coming out of the optimizers and generated manually like these cases, this was seen to not be an odd case, but rather the typical result at 8° angle-of-attack with flow attached over a 20° flap, despite the fact that the plain RAE2822 at similar conditions does not exhibit any flow separation until 12-13° angle-of-attack. It is also worthwhile to note that, even with this separation region, the flap has increased the C_L of the airfoil at 8° angle-of-attack from 1.10 to 2.08.

C. A Final Case

Any good engineer knows that there's more to a good design than single-point performance. Thus it is worth spending some time examining the results from optimization cases to set if they perform well in a wider sense. Figure 8 to the right shows lift curves generated for both the base RAE2822 airfoil and the final airfoil from a Particle Swarm optimization run (the last case from Fig. 4 above).

There are several interesting things to note about these lift curves. The first is that the flapped airfoil has indeed left the linear region by 8° angle-of-attack, well before the base airfoil. Thus, though $C_{L,\max}$ has definitely been increased, it also occurs earlier. It is not surprising at all that it occurs very near the single angle-of-attack that was run. The optimizer was ignoring all other points to improve this one.

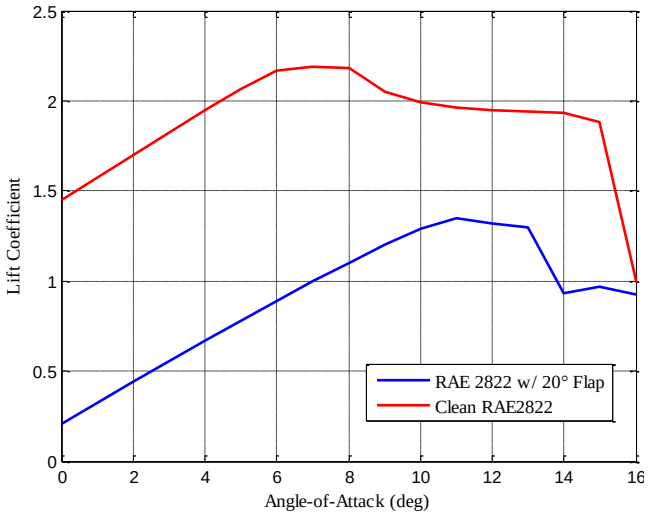

Figure 8 - Lift Curves of Base and Flapped Airfoils

V. Conclusion

While the short conclusion is that the optimization process does in fact improve $C_{L,\max}$, there is still much work to be done. Certainly the simple robustness tests need to be performed by attempting to use the methodology on different base airfoils, at different flap deflections, and so on. There are a few more particular questions that need to be answered, though.

The result that all of the cases run at 8° angle-of-attack experience some separation has called into question the decision to run the optimizers there. A course of action is unclear, however. Should the optimizer be run at 6°, where the airfoil does not seem to be experiencing separation? Or should the highest $C_{L,\max}$ be sought by running them at 11°, closer to $C_{L,\max}$ of the original airfoil? There is also the possibility that the optimizer could be designed to try to avoid these separated cases, possibly attempting to maximize C_L/C_D or something more unusual like C_L^2/C_D.

The flap leading edge geometry definitions remain somewhat unproven. It is clear that the methodology can create a flap that subjectively looks like a flap should, but whether the method is general enough to describe the best possible flap for an airfoil is unknown. Future tests should include cases for which known flaps have been developed and tested, so that there can be a comparison between optimizer results and the traditional methods of developing a flap geometry. Additionally, all of the optimization cases run so far have used 4th-order Bezier curves to define the flaps. This was chosen simply because it is the simplest case. It is certainly possible that the greater degree of freedom created by adding more control points would allow for better solutions.

Acknowledgments

The author would like to thank:

- Dr. Stephen Ruffin of Georgia Tech for support as Undergraduate Research Advisor.
- George Hicks and Guil Oliveira at Gulfstream Aerospace for initial motivation and numerous consultations on this project.
- Mike Foster and Andy Slater at Gulfstream Aerospace for facilitating the use of Gulfstream computer time to run these cases.

References

[1]Stevens, W.A., Goradia, S.H., Braden, J.A., Morgan, H.L., "Mathematical Model for Two-Dimensional Multi-Component Airfoils in Viscous Flow," AIAA 72-2, 10th AIAA Aerospace Sciences Meeting, 17-19 January 1972.

[2]Smith, A.M.O., "High-Lift Aerodynamics," *Journal of Aircraft*, Vol. 10, No. 6, Jun. 1975, pp. 501-530.

[3]Kulfan, B.M., "A Universal Parametric Geometry Representation Method - 'CST,'" AIAA 2007-62, 45th AIAA Aerospace Sciences Meeting and Exhibit, 8-11 January 2007.

[4]Millman, R.S. and Parker, G.D., *Elements of Differential Geometry*, Prentice-Hall Inc., Upper Saddle River, NJ, 1975, pp. 42.

[5]Engwirda, D. "MESH2D - Automatic Mesh Generation (v2.4)," *Matlab Central* [online code database], URL: http://www.mathworks.com/matlabcentral/fileexchange/25555-mesh2d-automatic-mesh-generation/ [cited 27 Feb 2010].

[6]*Fun3d Manual.* NASA Langley Research Center website, URL: http://fun3d.larc.nasa.gov [cited 27 Feb 2010].

[7]Anderson, W.K., Bonhaus, D.L., McGhee, R., and Walker, B., "Navier-Stokes Computations and Experimental Comparisons for Multielement Airfoil Configurations," AIAA 93-0645, AIAA Aerospace Sciences Meeting, 1993.