Viscous Airfoil Optimization Using Conformal Mapping Coefficients as Design Variables

by

Thomas Mead Sorensen

B.S. University of Rochester (1989)

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science
at the
Massachusetts Institute of Technology

June 1991

Signature of Author [illegible]
Department of Aeronautics and Astronautics
May 3, 1991

Certified by
Professor Mark Drela
Thesis Supervisor, Department of Aeronautics and Astronautics

Accepted by [illegible]
Professor Harold Y. Wachman
Chairman, Department Graduate Committee

Viscous Airfoil Optimization Using Conformal Mapping Coefficients as Design Variables

by

Thomas Mead Sorensen

Submitted to the Department of Aeronautics and Astronautics
on May 3, 1991
in partial fulfillment of the requirements for the degree of
Master of Science in Aeronautics and Astronautics

The XFOIL viscous airfoil design code was extended to allow optimization using conformal mapping coefficients as design variables. The optimization technique employed was the Steepest Descent method applied to an Objective/Penalty Function. An important idea of this research is the manner in which gradient information was calculated. The derivatives of the aerodynamic variables with respect to the design variables were cheaply calculated as a by-product of XFOIL's viscous Newton solver. The speed of the optimization process was further increased by updating the Newton system boundary layer variables after each optimization step using the available gradient information. It was determined that no more than 10 of each of the symmetric and anti-symmetric modes should be used during the optimization process due to inaccurate calculations of the higher mode derivatives. Both constant angle of attack and constant lift coefficient optimizations are possible. Multi-point optimization was also developed, although it works best for angle of attack design points. Four example optimizations are documented at the end of this thesis.

Thesis Supervisor: Mark Drela,
Assistant Professor of Aeronautics and Astronautics

Acknowledgments

I would like to express my sincerest appreciation for the guidance and counsel of my advisor, Mark Drela. His technical brilliance and selfless support have made a difficult job easier. He never failed to shed light on a stumbling block. I look forward to the chance of learning from him for several more years.

To everyone in the CFD lab - Thank you. Without the help and advice from everyone in the cluster I might still be coding, or writing, or ...

On a personal side, my thanks to my parents for never pushing but always being close enough to fall back on. And, for Dorene. For all you have endured for this day to be possible, I love you.

To my best friend, Roger Osmun, I want to dedicate this work. Without Roger's help I never would have finished my first fluids lab way back when. Due to circumstances beyond our control he could not have a direct hand in this thesis. But, I know that if he were nearby he would have done anything to help. For this, I thank him.

My thanks to the Department of Aeronautics and Astronautics' Fellowship and the National Science Foundation's PYI Program for their financial support of this research.

Contents

List of Figures

Nomenclature

Optimization Variables

F	Objective function
$\mathbf{x}$	General design variables
L	Lagrange multiplier function
$\mathbf{y}$	Lagrange multipliers
P	Penalty function
g_j	Inequality constraints
h_k	Equality constraints
K_j	Inequality constraint switches
n	Number of design variables
m	Number of inequality constraints
l	Number of equality constraints
κ	Cost parameter
r	Optimization step index
ϵ	Optimization step parameter
$\mathbf{s}$	Optimization search direction
A_n	XFOIL thickness design variables (symmetric)
B_n	XFOIL camber design variables (anti-symmetric)
n	Mode number
N_A	Last symmetric design mode used in optimization
N_B	Last anti-symmetric design mode used in optimization
$\mathbf{e}_n^A$	Unit vector in A_n direction
$\mathbf{e}_n^B$	Unit vector in B_n direction
$[J]$	Newton system Jacobian matrix

$[A]$	Addition to Jacobian matrix
$\{\delta\}$	Newton system unknown vector
$\{\Delta\}$	Addition to unknown vector
$\{R\}$	Residual vector
$[D]$	Aerodynamic variables derivative matrix
N_p	Number of optimization design points
w_m	Weighting function of m^{th} design point
p_m	Local penalty function of m^{th} design point

Aerodynamic Variables

C_l	Coefficient of lift
C_d	Coefficient of drag
C_m	Coefficient of moment
M	Mach number
Re	Reynolds number based on airfoil chord
N	Number of airfoil nodes
N_w	Number of wake nodes
f_i, g_i, h_i	Node i boundary layer equations
C_τ	Most-amplified Tollmien-Schlichting wave (laminar regions)
	Maximum shear stress coefficient (turbulent regions)
θ	Momentum thickness
m	Boundary layer mass deficit
d_{kj}	Mass influence matrix
u_e	Viscous edge velocity
δ^*	Displacement thickness
x_{tran}	Transition point location
c	Airfoil chord
$C_n = A_n + iB_n$	Complex mapping coefficients
ϵ_{te}	Trailing edge angle parameter
U_∞	Freestream velocity

α	Angle of attack
q	Inviscid surface speed
$z = \frac{x}{c} + i\frac{y}{c}$	Complex airfoil-plane coordinate
$\zeta = re^{i\omega}$	Complex circle-plane coordinate
$F(\zeta)$	Complex circle-plane potential
Γ	Circulation
w^*	Conjugate velocity in circle-plane
u_x, u_y	Velocity components in airfoil-plane
H	Shape parameter

Superscripts and Subscripts

$(\ )^l$	Lower design variable bound
$(\ )^u$	Upper design variable bound
$(\ )^r$	Current optimization step
$(\ )^{r+1}$	New optimization step

Miscellaneous Symbols

ϵ_s	Small quantity
∇	Gradient with respect to design variables
Δ	Difference operator
$\delta(\ )$	Newton system perturbation
$\Re(\ )$	Real part of the quantity in the parenthesis
$\Im(\ )$	Imaginary part of the quantity in the parenthesis

NOTE: **bold face** denotes vector quantities

Chapter 1
Introduction

The design of efficient two-dimensional airfoils is important to the aerospace industry because the aerodynamic performance of most aircraft is strongly influenced by the airfoil section characteristics. Airfoil design is an iterative process: the designer successively modifies the design until it can no longer be improved. Accurate high speed computer codes have made it possible to design exceptional airfoils by allowing designers to experiment with various ideas and to test them immediately. This is a time consuming process, however, and the number of trials necessary to arrive at the final design is dependent upon the expertise of the designer.

The object of optimization is to replace the designer with a routine that will take a given airfoil (the seed airfoil) and modify it such that a specified characteristic (called the Objective Function) will be driven as small as possible while satisfying several constraints. The need for the designer is not eliminated by the optimizer, instead the designer is freed from having to do the tedious design iterations. With the optimizer doing the repetitive process the designer can concentrate on more important questions, such as: What should be optimized? and, What constraints should the airfoil satisfy? Also, the experience of the designer can reduce the time requirement of the optimization process by choosing the seed airfoil wisely. Obviously, if the seed airfoil is almost optimal the optimizer will not have much work to do.

Airfoil design is a complex problem requiring a host of compromises between conflicting requirements [2]. There are also off-design performance considerations to take into account. In such a situation a truly optimal airfoil is difficult or impossible to define, therefore the role of the optimizer will not be to produce the perfect airfoil, but to reduce the Objective Function on a more simply constrained airfoil. In this manner the designer can evaluate the trends that develop for reducing the Objective Function on this simple airfoil and apply these trends, when warranted, to the real airfoil. The optimizer will become just another tool available to the designer rather than rendering the designer obsolete.

Several optimization packages are on the market. This author has experience with COPES/ADS [4, 10]. This particular code is utilized as a black box and is extremely general, allowing many problems to be optimized by linking an analysis code, that evaluates the Objective Function, to the optimization code. This has distinct advantages for one time only optimizations, but is highly inefficient for problems that will be done often. A better solution in these cases is to write a dedicated optimizer that utilizes the unique traits of the analysis code to increase optimization efficiency. For example, COPES/ADS uses finite difference techniques to calculate gradient information to be used in the optimization process. If the analysis code calculates this gradient information concurrently with the Objective Function, it should be used by the optimizer. To the author's knowledge this is something that COPES/ADS does not do.

G. N. Vanderplaats has tried many optimization problems and techniques, including optimizing airfoils by building optimal airfoils from combinations of several "Basis Airfoils" [9, 11]. A possible shortcoming of this technique is that, to a certain extent, the designer assumes the shape of the optimal airfoil by the choice of basis airfoils. In other words, a completely general airfoil cannot be constructed by this technique.

The object of the present research is to modify an existing 2D airfoil design code to perform optimization using a general set of design variables. The code used is Drela's XFOIL code [1]. XFOIL has several design routines, and includes both viscous and inviscid analysis routines. Principles from both the design and viscous analysis routines were combined to allow fully viscous optimizations. The optimizer was written explicitly to make the most of XFOIL's attributes to speed the optimization process, however, the emphasis was on developing the ingredients for the optimization: design variables and gradient information. To this end, the actual optimization technique was chosen to be simple and robust, but not necessarily efficient. Combining the ingredients with an efficient optimization technique is a good topic for follow-up work.

The outline for the remainder of this thesis is to first present general optimization theory and terminology, with emphasis placed on those techniques used in the XFOIL optimizer. An excellent reference on the theory of optimization is the book by G. N. Vanderplaats [11]. Once the prerequisite tools are in place, the adaptation to XFOIL will be discussed. Results of several test cases will then be presented, followed by the conclusions of this research.

Chapter 2
Optimization Theory

2.1 Constrained Problem Statement

Airfoil optimization is classified as a constrained optimization problem. The equations governing such a problem are listed in Table 2.1 [11].

Minimize:	$F(\mathbf{x})$		Objective Function
Subject to:	$x_i^l \leq x_i \leq x_i^u$	$i = 1, n$	Side Constraints
	$g_j(\mathbf{x}) \geq 0$	$j = 1, m$	Inequality Constraints
	$h_k(\mathbf{x}) = 0$	$k = 1, l$	Equality Constraints
Where:	$\mathbf{x} = \left\{ \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right\}$		Design Variables

Table 2.1: Constrained Optimization Equations

The Objective Function is the function that the optimizer will drive to the lowest possible value, subject to the stated constraints. For airfoil optimization the Objective Function could be simply the drag coefficient or a combination of several airfoil characteristics such as the negative of the range parameter, $-MC_l/C_d$.

The constraints come in three fundamental forms and serve to limit how the airfoil can be changed. Side constraints are the least restrictive; their sole purpose is to put bounds on the design variables. This type of constraint is enforced directly onto the design variables and, in effect, cuts a block out of the n-dimensional space within which to search for the optimum. Inequality constraints are the next least restrictive constraint. Vanderplaats [11] defines inequality constraints as $g_j(\mathbf{x}) \leq 0$ but in this thesis the definition will be $g_j(\mathbf{x}) \geq 0$. The difference in terminology was used to ensure that specified quantities remain positive. For example, an airfoil cannot have negative thickness. In this case the airfoil thickness is the quantity that must remain positive. This type of constraint cannot in general be enforced directly onto the design variables but is instead a function of the design variables. Notice that the side constraints are degenerate inequality constraints. Each side constraint can be written as two inequality constraints. An inequality constraint is analogous to a fence separating areas in the design space which allow permissible designs from those areas where the constraint is violated. Equality constraints are the last and generally most restrictive constraints. These constraints restrict the location of the optimum to only those regions of the design space where the constraint is satisfied.

Figure 2.1 shows how a general two degree of freedom (DOF) constrained optimization space might look [11]. The unconstrained optimum is located at the cross. However, assume that in this example the two inequality constraints prevent solutions above the dotted lines while the equality constraint forces the optimum to be located along the dashed line. Then the solution to this optimization problem is the lowest point along the dashed line, but which is also under the two dotted lines. This point is marked by the square.

A constraint is referred to as satisfied if the constraint equation is true at the current design iteration. For an optimum to be found all constraints must be satisfied. If a constraint is not satisfied it is referred to as unsatisfied or violated. A constraint is considered active if it affects the optimization process. By definition, all equality constraints are active since they must always be satisfied. An inequality constraint is only active if it is less than or equal to zero.

2.2 Equivalent Unconstrained Problem

Any constrained optimization problem can be converted into an unconstrained problem. This is beneficial because, in general, optimization is simpler to perform on an unconstrained problem. Two methods of conversion were investigated: the Lagrange Multiplier Method [7] and the Penalty Function Method [11]. Both methods involve defining a new optimization function that combines the Objective Function with the constraints.

Lagrange Multipliers work well when optimizing an analytic quadratic function constrained only by linear equality constraints. The new function to be optimized is defined as

$$L(\mathbf{x},\mathbf{y}) = F(\mathbf{x}) + \mathbf{y}\cdot\mathbf{h}(\mathbf{x}), \tag{2.1}$$

where $\mathbf{y}$ are referred to as the Lagrange multipliers. The optimum of the constrained function can be found by taking the derivatives of L with respect to $\mathbf{x}$ and $\mathbf{y}$, and setting them equal to zero. By doing this a system of equations is obtained for $\mathbf{x}$ and $\mathbf{y}$, however, this system is linear only if L is quadratic. For higher-order problems the resulting system of equations is non-linear and would be solved numerically using a Newton-Raphson scheme. Generating the Newton system can be computationally expensive for non-analytic problems since second derivative information will be needed. A further drawback of the Lagrange Multiplier Method is that only equality constraints can be treated.

In contrast, Penalty Functions work better for numerical problems and either inequality or equality constraints can be handled. Using this technique it is necessary to write all side constraints as inequality constraints, which as mentioned previously creates no difficulties. The penalty function is defined as

$$P(\mathbf{x}) = F(\mathbf{x}) + \frac{1}{2}\sum_{j=1}^{m} K_j\,(g_j(\mathbf{x}))^2 + \frac{1}{2}\sum_{k=1}^{l} \kappa\,(h_k(\mathbf{x}))^2, \tag{2.2}$$

where,

$$K_j = \begin{cases} 0 & g_j(\mathbf{x}) \geq 0 \\ \kappa & g_j(\mathbf{x}) < 0 \end{cases}, \tag{2.3}$$

is the switch that turns inequality constraints off if they are not active. The cost parameter, κ, is a large positive quantity, the purpose of which is shown below.

The one dimensional function depicted in Fig. 2.2 can be used to illustrate this technique. Consider the inequality constraint, $g(x) = 2 - x \geq 0$. Notice that the unconstrained optimum (the optimum of the Objective Function) cannot be allowed because the constraint is not satisfied. To the left of $x = 2$, $K = 0$, thus the optimizer works on the Objective Function. To the right of $x = 2$, $K = \kappa$, therefore the optimizer works on the Objective Function with an added quadratic term. This added term forces the optimum of the Penalty Function to lie between the true optimum and the unconstrained optimum. The problem with this procedure is obvious: as long as $\kappa \neq \infty$ the calculated optimum will always violate the constraint. However, for this case it was found that $100 \leq \kappa \leq 1000$ provides an acceptable constraint violation. Of course, different values of κ will be appropriate for different problems.

From here on it will be assumed that the optimization process uses the Penalty Function. It will also be referred to as the optimization function.

2.3 Iterative Optimization Technique

The optimization technique employed in the present development is iterative. The design variables of each new airfoil are calculated from the current airfoil using

$$\mathbf{x}^{r+1} = \mathbf{x}^r + \epsilon^r \mathbf{s}^r. \tag{2.4}$$

In Eq. (2.4), $\mathbf{x}^r$ are the current airfoil design variables, $\mathbf{x}^{r+1}$ are the new airfoil design variables, $\epsilon^r > 0$ is the step parameter, and $\mathbf{s}^r$ is the search direction (thus, $\epsilon^r \mathbf{s}^r$ is the step size). The optimization iteration index is r, where r = 0 for the seed airfoil. From Eq. (2.4) it is obvious that for an iterative optimization approach there are only two parameters, ϵ^r and $\mathbf{s}^r$, that need to be calculated for each optimization step. Figure 2.3 schematically represents the terms in Eq. (2.4) in a 2 DOF optimization space.

Using the iterative optimization approach, the following 7 stages constitute one optimization iteration (or step):

1. Start with a known airfoil geometry ($\mathbf{x}^r$ known).
2. Do the design analysis.
3. Has an optimum been found? If yes, then stop.
4. Choose a search direction, $\mathbf{s}^r$.
5. Choose a step parameter, ϵ^r.
6. Calculate $\mathbf{x}^{r+1}$ from Eq. (2.4).
7. Increment r and return to Stage 1.

The stages are repeated until the optimum has been found. Stages 1, 6, and 7 are self-explanatory. Stage 2 is performed by the analysis portion of the code and is unaffected by the optimizer. All information related to the current design is calculated in this stage; for example, calculation of the Objective Function and any gradient information is done here. The remaining stages will be explained in more detail in the following paragraphs.

2.3.1 Optimum Convergence (Stage 3)

It is not possible to know whether or not a global optimum has been found, the best that can be done is to determine if a local optimum has been found. Vanderplaats [11] has outlined a routine to check for convergence using three criteria. The optimizer in XFOIL is stopped if any of the following are satisfied:

$$r \geq 50, \tag{2.5}$$

$$P(\mathbf{x}^r) - P(\mathbf{x}^{r-1}) \leq 0.001P(\mathbf{x}^0), \tag{2.6}$$

or

$$P(\mathbf{x}^r) - P(\mathbf{x}^{r-1}) \leq 0.001P(\mathbf{x}^r). \tag{2.7}$$

Equation (2.5) puts a ceiling on how many iterations are allowed to prevent the possibility of an infinite loop if a problem occurs with the optimization process. Equation (2.6) stops the optimizer if the change in P between the last iteration and the current iteration slows to a small fraction of the starting value of P. Equation (2.7) stops the optimizer if the change in P is smaller than a small fraction of the current P. The former is referred to as the absolute convergence check and the latter as the relative convergence check. For the optimization to be stopped a convergence criteria must be triggered on 3 consecutive iterations. This is to prevent the optimizer from being punished if, on any one cycle, it slows down but then speeds up again.

A fourth condition that Vanderplaats suggests using is the Kuhn-Tucker Condition. This is nothing more than checking that $\nabla P(\mathbf{x}) \leq \epsilon_s$ at the optimum, where ϵ_s is a value sufficiently close to zero. This condition was not used as it was deemed unnecessary. However, a fourth condition was found to be needed due to the nature of XFOIL's viscous solver. This condition required the optimization to be stopped at the end of any iteration in which the Newton solver did not converge. This was necessary due to the ever present possibility of creating an airfoil that XFOIL for one reason or another could not evaluate. Usually when the Newton system did not converge for one iteration it would disrupt further iterations until the code crashed. By stopping the optimizer before this happened the results up to the current iteration could be saved.

2.3.2 Choosing the Search Direction (Stage 4)

Many techniques are available to pick $\mathbf{s}^r$ [5, 11]. Optimization efficiency can be greatly improved with a wise choice of search direction. The method utilized in XFOIL to find $\mathbf{s}^r$ is the Steepest Descent approach. It was chosen for its simplicity, but any method consistent with an iterative, unconstrained optimization technique could be used. The Steepest Descent method picks

$$\mathbf{s}^r = -\nabla P(\mathbf{x}^r), \tag{2.8}$$

i.e. the direction opposite to which the function is experiencing the greatest change. A reduction in the optimization function is guaranteed since the change in P due to a change in $\mathbf{x}$ is

$$\Delta P = \nabla P \cdot \Delta \mathbf{x}, \tag{2.9}$$

where, $\Delta(\ ) = (\ )^{r+1} - (\ )^r$. Substituting Eqs. (2.4 & 2.8) into Eq. (2.9) implies

$$\Delta P = -\epsilon^r |\nabla P|^2. \tag{2.10}$$

A sufficiently small positive ϵ^r thus ensures a reduction in P. Referring to Fig. 2.3, $\epsilon^r \mathbf{s}^r$ would be perpendicular to the contour through $\mathbf{x}^r$ when using the Steepest Descent technique.

Flowing water follows the path of least resistance, i.e. the path of steepest descent. As it flows, water is at every point flowing in the steepest descent direction. This is not practicable in an optimization problem, since this would require the step parameter to be infinitesimal. For a real problem the hope is to find the optimum with the lowest expenditure of resources, therefore, the optimizer should take as large a step as possible before picking a new search direction. The size of the step is controlled by the step parameter, ϵ^r and is discussed below.

2.3.3 Choosing a Step Parameter (Stage 5)

As stated above the step parameter should be as large as possible. The literature [5, 7, 11] suggests that once a search direction is picked, the step parameter should be chosen to minimize the optimization function in that direction. This requires that a one-dimensional optimization be performed along the search direction. The XFOIL optimizer does not do this, instead a constant step parameter is used. There were two distinct advantages with letting ϵ^r = constant. The first was simplicity. Since the emphasis was not on the optimization technique, it was deemed unnecessary to write and debug a 1D optimization routine. The second and more important advantage involves program robustness. XFOIL is not very forgiving of non-realistic airfoils which are possible when trying to find 1D minima. However, if the steepest descent direction is chosen after each small step rather than after a large minimizing step, the airfoil will have a better chance to remain in a region of XFOIL compatible airfoils.

A constant step parameter implies that the step size will be directly proportional to the gradient of the Penalty Function. When the gradient is large the step size is large, when the gradient is small (as it is when an optimum is neared) the step size is small. This system is very inefficient in terms of the number of steps to find the optimum. In fact the exact optimum will never be obtained due to the slow response as the optimum is neared. However, this system has proven to be more robust than when using the accepted procedure of changing the search direction only after the 1D optimum is found. In addition, as stated in the introduction, the exact optimum is relatively unimportant so the slow response as it is neared is inconsequential.

Since the step size was directly related to the gradient, in regions of large slopes the step size could become quite large. To prevent extremely large changes in the airfoil that might cause XFOIL to crash, the changes in the symmetric design variables were limited to 10% of their values at the current airfoil by reducing ϵ^r. Only the symmetric modes were considered since the the anti-symmetric design variables can be very small and will thus trigger this restriction unnecessarily.

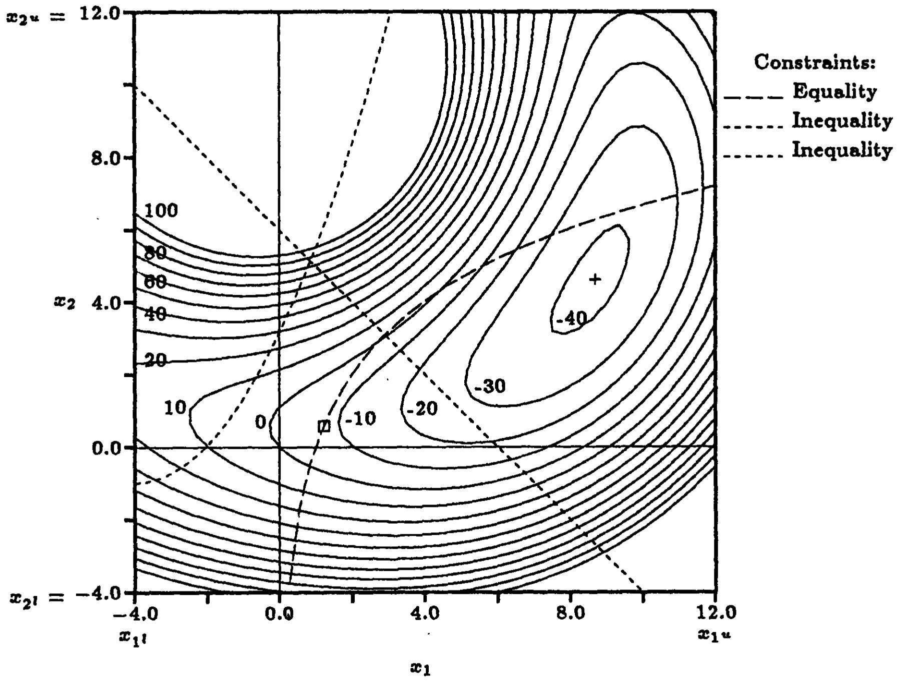

Figure 2.1: 2 DOF Optimization Space, Contours of $F(x_1, x_2)$

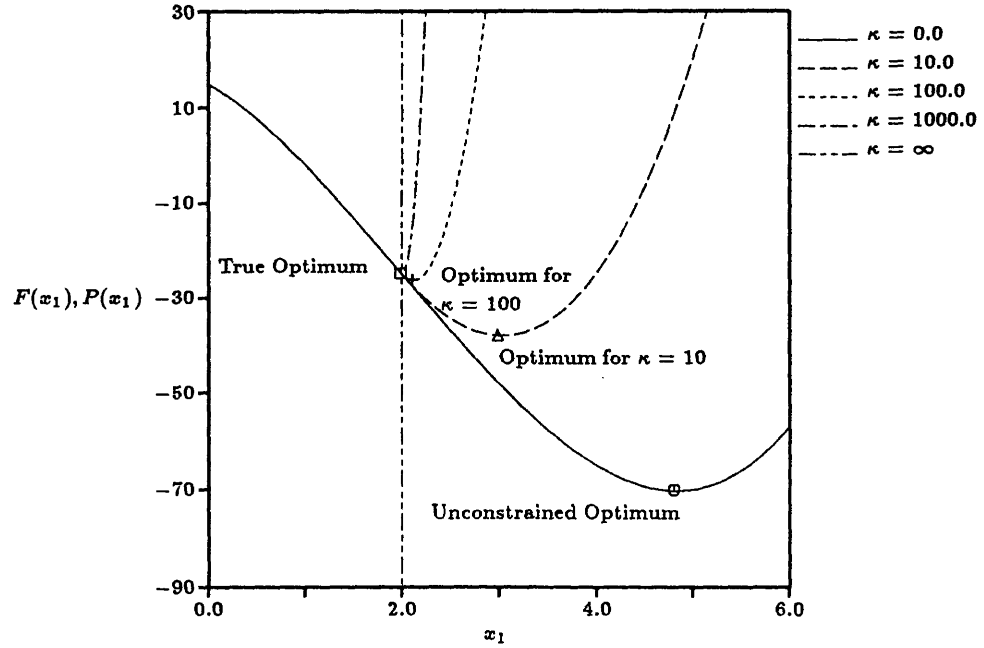

Figure 2.2: 1 DOF Penalty Function Constraint Example

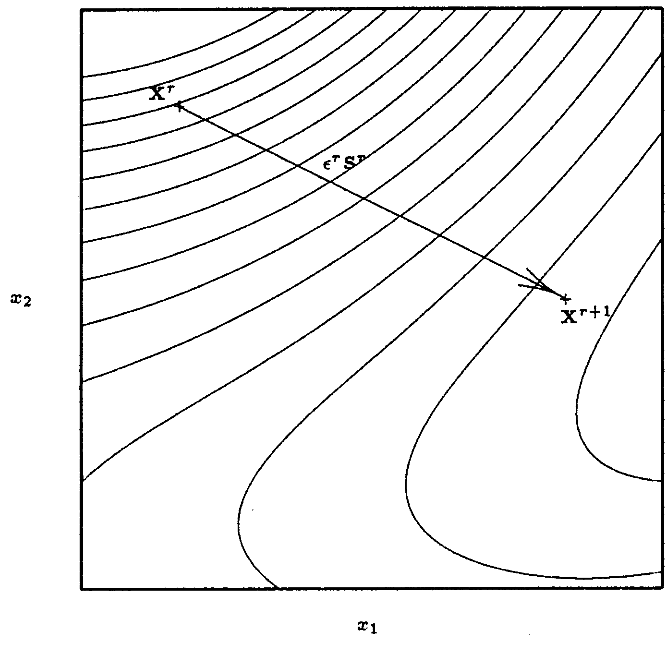

Figure 2.3: Schematic Diagram of Iterative Optimization Step

Chapter 3
XFOIL Optimization

3.1 Design Variables

The first step in modifying XFOIL to perform optimizations is to specify the optimization design variables, **x**. This is not an easy task. Completely general airfoils should be able to be created with the minimum number of design variables. As more design variables are used the optimization problem will tend to become ill-posed [11]. Typically fewer than 20 design variables is best.

XFOIL uses a panel method for its analysis, so the first idea that may come to mind is to use the airfoil's y-coordinates at the paneling nodes as the design variables. This is soon rejected due to the number of nodes involved in a typical problem (usually between 80 and 100). A second idea is to choose a small number (5 to 10) of standard airfoils and to use these shapes, or modes as they are sometimes called, as the design variables. This was done by Vanderplaats as already mentioned. In this way the optimal airfoil is constructed by a weighted sum of the standard airfoils. This has the advantage of using only a small number of design variables but these cannot be added up to construct a completely general airfoil.

The design variables employed in XFOIL's optimizer are the real and imaginary parts of the complex coefficients of the mapping function

$$\frac{\partial z}{\partial \zeta} = (1 - \frac{1}{\zeta})^{(1-\epsilon_{te})} \exp\left\{ \sum_{n=0}^{\infty} C_n \zeta^{-n} \right\}, \tag{3.1}$$

where,

$$C_n = A_n + iB_n, \qquad n = 0, 1, 2, \cdots. \tag{3.2}$$

The trailing edge angle is $\pi\epsilon_{te}$. The unit circle is mapped from the ζ-plane to an airfoil in the z-plane by this transformation [3, 6, 8]. The design variables are,

$$\mathbf{x} = \{A_2,\ A_3, \cdots\ A_{N_A},\ \ B_2,\ B_3, \cdots\ B_{N_B}\}^T, \tag{3.3}$$

and are similar to the latter technique described in the above paragraph, since they too can be thought of as separate shapes, or design modes, summed to produce the optimal airfoil. Each design variable corresponds to a single design mode. A particular convenience of using these design variables is that the A_n's control the thickness distribution of the airfoil (symmetric modes) and the B_n's control the camber distribution (anti-symmetric modes). The first usable design modes are A_2 and B_2 since A_0, A_1, B_0, and B_1 are constrained by Lighthill's constraints [6] and therefore are not available as design variables. The last symmetric design variable mode number is N_A and the last anti-symmetric design variable mode number is N_B.

The A_n and B_n coefficients completely control the airfoil geometry with the exception of the trailing edge angle and gap. For a typical airfoil only the first twenty or so C_n's are required to define the airfoil. The value of the design variables for a DAE11 airfoil are plotted in Fig. 3.1 as an indication of their magnitudes for a typical airfoil. Notice how quickly the higher frequency modes become unimportant. In both cases, only approximately the first 15 modes are important. The higher modes, however, become important for airfoils with small leading edge radii. An attempt was made to reduce the higher modes by placing the factor, $(1 - \frac{\zeta_0}{\zeta})$, in Eq. (3.1). The idea was to better model the leading edge mapping before using the correcting terms, $\exp\{\sum C_n\zeta^{-n}\}$. Unfortunately, this did not work. In fact it would either reduce the values of the lower modes to the point where all modes were of the same order of magnitude, or it would magnify the higher modes while reducing the lower modes. Due to these results, no other attempts to dampen the higher frequency modes were made.

The first 3 symmetric and anti-symmetric design modes are shown in Figs. 3.2 and

3.3, respectively. The solid lines in Fig. 3.2 indicate the airfoil surface for one value of A_n. The dashed lines show how the surface (i.e. the thickness) changes as another value of A_n is used. In Fig. 3.3, the lines are not the airfoil surface, but the camber lines.

3.2 Constraints

The XFOIL implementation allows optimizations to be carried out either with a constant angle of attack or a constant lift coefficient. These are actually unwritten equality constraints. They are not included in the constraint summations of Eq. (2.2) because they are an integral part of the Objective Function.

To the designer C_l is more relevant than α, however, the optimization process is more efficient for constant α problems. The reason is that α is independent of the design variables while C_l is not. Thus, when holding C_l constant the steepest decent technique picks the search direction based on the current α and after stepping to the new point α is adjusted to keep C_l constant. This adjustment will, in effect, change the optimization space at the new point, therefore, the direction traveled will no longer be the Steepest Descent. However, in practice both methods can be used without any serious discrepancies.

3.3 Computing Gradients

3.3.1 Aerodynamic Quantities

Use of the Steepest Descent method to calculate the search direction requires that gradient information be calculated. The gradient information will also prove useful in making the analysis procedure run faster as will be shown shortly. The manner in which the aerodynamic gradients are calculated without resorting to costly finite differencing is the most novel feature of this research.

Before continuing, a matter of notation must be addressed. Considering the design variables in two groups, symmetric and anti-symmetric, makes the following definition of the gradient operator with respect to the design variables useful

$$\nabla(\,) = \sum_{n=2}^{N_A} \frac{\partial(\,)}{\partial A_n} \mathbf{e}_n^A + \sum_{n=2}^{N_B} \frac{\partial(\,)}{\partial B_n} \mathbf{e}_n^B, \tag{3.4}$$

where, $\mathbf{e}_n^A$ and $\mathbf{e}_n^B$ are unit vectors in the appropriate directions. When $\frac{\partial(\,)}{\partial A_n}$ is used alone it will imply the partial derivative of the function in parenthesis with respect to the single design variable A_n where n can take on any integer value between 2 and N_A. Similarly for $\frac{\partial(\,)}{\partial B_n}$. In the remaining text, all gradients will refer to Eq. (3.4).

In its unmodified configuration XFOIL solves a viscous flow around an airfoil by constructing 3 linearized boundary layer (BL) equations at each airfoil and wake node (N airfoil nodes, N_w wake nodes) and solving the resulting system using a Newton solver. For a viscous airfoil analysis all aerodynamic quantities of interest are functions of the five BL variables: C_τ, θ, m, u_e, and δ^*. In this text C_τ will represent two quantities: in laminar regions it will be the amplitude of the most-amplified Tollmien-Schlichting wave, and in turbulent regions it will be the maximum shear coefficient. The Newton system only solves for three of these variables, C_τ, θ, and m, since u_e and δ^* are related to the first three variables. For more details of XFOIL, see Drela [1].

As an example of the kind of information needed and how it is obtained consider the expression for C_d,

$$C_d = \frac{2}{c}\theta_\infty, \tag{3.5}$$

where θ_∞ is the momentum thickness of the wake far downstream and c is the airfoil chord. Therefore, the gradient of C_d with respect to the design variables is

$$\nabla C_d = \frac{2}{c}\nabla\theta_\infty, \tag{3.6}$$

where, Eq. (3.4) is employed. The gradients of all other aerodynamic quantities can be obtained in a similar manner if the gradients of the five BL variables are known.

To calculate the required BL variable gradients, consider the Newton System used in XFOIL

$$[J]\{\delta\} = -\{R\}. \tag{3.7}$$

This equation is a block matrix equation where the i^{th}-row, j^{th}-column block of the Jacobian Matrix is

$$[J_{i,j}] = \begin{bmatrix} \frac{\partial f_i}{\partial C_{\tau_j}} & \frac{\partial f_i}{\partial \theta_j} & \frac{\partial f_i}{\partial m_j} \\ \frac{\partial g_i}{\partial C_{\tau_j}} & \frac{\partial g_i}{\partial \theta_j} & \frac{\partial g_i}{\partial m_j} \\ \frac{\partial h_i}{\partial C_{\tau_j}} & \frac{\partial h_i}{\partial \theta_j} & \frac{\partial h_i}{\partial m_j} \end{bmatrix}. \tag{3.8}$$

The corresponding i^{th}-row block of the vectors are

$$\{\delta_i\} = \begin{Bmatrix} \delta C_{\tau_i} \\ \delta\theta_i \\ \delta m_i \end{Bmatrix}, \qquad \{R_i\} = \begin{Bmatrix} f_i \\ g_i \\ h_i \end{Bmatrix}. \tag{3.9}$$

Many of the terms in the Jacobian Matrix are zero, but the detailed structure is not important here.

Equation (3.7) is constructed using 3 BL equations at each node with the functional forms

$$\begin{aligned} f_i &= f_i(C_{\tau_{i-1}},\ C_{\tau_i},\ \theta_{i-1},\ \theta_i,\ m_1,\ m_2,\cdots,\ m_{N+Nw}), \\ g_i &= g_i(C_{\tau_{i-1}},\ C_{\tau_i},\ \theta_{i-1},\ \theta_i,\ m_1,\ m_2,\cdots,\ m_{N+Nw}), \\ h_i &= h_i(C_{\tau_{i-1}},\ C_{\tau_i},\ \theta_{i-1},\ \theta_i,\ m_1,\ m_2,\cdots,\ m_{N+Nw}), \end{aligned} \tag{3.10}$$

where the subscripts indicate which node is being considered. However, recognizing that f_i, g_i, and h_i are geometry dependent implies that they must also be functions of A_n and B_n. Consequently, a new Newton system is obtained in the form

$$[J \mid A]\left\{\begin{matrix}\delta \\ \hline \Delta\end{matrix}\right\} = -\{R\}. \tag{3.11}$$

The i^{th}-row block of the Jacobian addition, $[A]$, is

$$[A_i] = \begin{bmatrix} \frac{\partial f_i}{\partial A_2} & \frac{\partial f_i}{\partial A_3} & \cdots & \frac{\partial f_i}{\partial A_{N_A}} & \frac{\partial f_i}{\partial B_2} & \frac{\partial f_i}{\partial B_3} & \cdots & \frac{\partial f_i}{\partial B_{N_B}} \\ \frac{\partial g_i}{\partial A_2} & \frac{\partial g_i}{\partial A_3} & \cdots & \frac{\partial g_i}{\partial A_{N_A}} & \frac{\partial g_i}{\partial B_2} & \frac{\partial g_i}{\partial B_3} & \cdots & \frac{\partial g_i}{\partial B_{N_B}} \\ \frac{\partial h_i}{\partial A_2} & \frac{\partial h_i}{\partial A_3} & \cdots & \frac{\partial h_i}{\partial A_{N_A}} & \frac{\partial h_i}{\partial B_2} & \frac{\partial h_i}{\partial B_3} & \cdots & \frac{\partial h_i}{\partial B_{N_B}} \end{bmatrix}. \tag{3.12}$$

The added vector term contains the changes in the design variables

$$\{\Delta\} = \left\{ \Delta A_2, \ \Delta A_3, \ \cdots \ \Delta A_{N_A}, \quad \Delta B_2, \ \Delta B_3, \ \cdots \ \Delta B_{N_B} \right\}^T, \tag{3.13}$$

where, $\Delta(\ ) = (\ )^{r+1} - (\ )^r$. The modified Jacobian matrix, $[J \mid A]$, is no longer square, but during normal viscous calculations the geometry is fixed and thus the ΔA_n and ΔB_n's are known (i.e. they are zero). Therefore, rewriting Eq. (3.11) with all knowns on the right hand side and then pre-multiplying both sides by $[J]^{-1}$ the system reduces to

$$\{\delta\} = -[J]^{-1}\{R\} + [D]\{\Delta\}, \tag{3.14}$$

where,

$$[D] = -[J]^{-1}[A]. \tag{3.15}$$

The viscous solution is obtained when the residual, $\{R\}$, is zero. Thus, at convergence Eq. (3.14) will have the same form as a first order Taylor series expansion of the 3 BL equations in terms of the design variables. For example, the Taylor expansion for C_τ, θ, and m at the i^{th} node is

$$\begin{Bmatrix} \delta C_{\tau_i} \\ \delta\theta_i \\ \delta m_i \end{Bmatrix} = \sum_{n=2}^{N_A} \Delta A_n \begin{Bmatrix} \frac{\partial C_{\tau_i}}{\partial A_n} \\ \\ \frac{\partial\theta_i}{\partial A_n} \\ \\ \frac{\partial m_i}{\partial A_n} \end{Bmatrix} + \sum_{n=2}^{N_B} \Delta B_n \begin{Bmatrix} \frac{\partial C_{\tau_i}}{\partial B_n} \\ \\ \frac{\partial\theta_i}{\partial B_n} \\ \\ \frac{\partial m_i}{\partial B_n} \end{Bmatrix}. \tag{3.16}$$

The Taylor coefficients are the BL variable derivatives being sought and after close examination it can be seen that they are the columns of $[D]$. For example, the i^{th}-row block of $[D]$ is

$$[D_i] = \begin{bmatrix} \frac{\partial C_{\tau_i}}{\partial A_2} & \frac{\partial C_{\tau_i}}{\partial A_3} & \cdots & \frac{\partial C_{\tau_i}}{\partial A_{N_A}} & \frac{\partial C_{\tau_i}}{\partial B_2} & \frac{\partial C_{\tau_i}}{\partial B_3} & \cdots & \frac{\partial C_{\tau_i}}{\partial B_{N_B}} \\ \\ \frac{\partial\theta_i}{\partial A_2} & \frac{\partial\theta_i}{\partial A_3} & \cdots & \frac{\partial\theta_i}{\partial A_{N_A}} & \frac{\partial\theta_i}{\partial B_2} & \frac{\partial\theta_i}{\partial B_3} & \cdots & \frac{\partial\theta_i}{\partial B_{N_B}} \\ \\ \frac{\partial m_i}{\partial A_2} & \frac{\partial m_i}{\partial A_3} & \cdots & \frac{\partial m_i}{\partial A_{N_A}} & \frac{\partial m_i}{\partial B_2} & \frac{\partial m_i}{\partial B_3} & \cdots & \frac{\partial m_i}{\partial B_{N_B}} \end{bmatrix}. \tag{3.17}$$

The components of $\nabla\theta_\infty$, the gradient required for calculating ∇C_d, are therefore the $\frac{\partial\theta_i}{\partial A_n}$ and $\frac{\partial\theta_i}{\partial B_n}$ elements for $i = N + N_w$.

The elements of this matrix are found not by carrying out the matrix multiplication as indicated in Eq. (3.15) but by solving the original Newton system with the columns of $[A]$ added as extra right hand sides. Since a direct matrix solver is used, very little extra work is needed to calculate the required sensitivities. In addition, the extra right hand sides only have to be included *after* convergence of the system, *not* every time the system is solved.

The above derivation presents a scheme to compute the BL variable gradients if the gradients of the BL equations, Eqs. (3.10), are known (i.e. if the terms of $[A]$ are known). The derivation of the terms in $[A]$ is relatively straight forward and is therefore relegated to Appendix A. The results are

$$\frac{\partial R_i}{\partial A_n} = \left(\frac{\partial R_i}{\partial q_{i-1}}\right)\left(\frac{\partial q_{i-1}}{\partial A_n}\right) + \left(\frac{\partial R_i}{\partial q_i}\right)\left(\frac{\partial q_i}{\partial A_n}\right), \tag{3.18}$$

$$\frac{\partial R_i}{\partial B_n} = \left(\frac{\partial R_i}{\partial q_{i-1}}\right)\left(\frac{\partial q_{i-1}}{\partial B_n}\right) + \left(\frac{\partial R_i}{\partial q_i}\right)\left(\frac{\partial q_i}{\partial B_n}\right), \tag{3.19}$$

where,

$$\frac{\partial R_i}{\partial q_{i-1}} = \frac{\partial R_i}{\partial u_{e_{i-1}}} - \frac{\partial R_i}{\partial \delta^*_{i-1}} \frac{m_{i-1}}{u^2_{e_{i-1}}}, \tag{3.20}$$

$$\frac{\partial R_i}{\partial q_i} = \frac{\partial R_i}{\partial u_{e_i}} - \frac{\partial R_i}{\partial \delta^*_i} \frac{m_i}{u^2_{e_i}}. \tag{3.21}$$

In the above four equations R_i can be f_i, g_i, or h_i. At node i the derivatives depend only on the information at that node and the upstream node $i-1$. All the terms in Eqs. (3.20 & 3.21) are already available once XFOIL constructs the Newton system. The only remaining unknown sensitivities in Eqs. (3.18 & 3.19) are the derivatives of q. These can be calculated analytically from the expression for q obtained after the complex potential is mapped from the circle-plane to the airfoil-plane. At any point, ζ, in the circle-plane, the physical speed is

$$q = \exp\left\{\Re\left[\ln\left(\left(1-\frac{1}{\zeta}\right)^{\epsilon_{te}}\left(e^{-i\alpha}+e^{i\alpha}\zeta^{-1}\right)\right) - \sum_{n=0}^{\infty} C_n \zeta^{-n}\right]\right\}. \tag{3.22}$$

The derivatives of this equation are remarkably easy and cheap to compute:

$$\frac{\partial q}{\partial A_n} = -q\Re\left(\frac{1}{\zeta^n}\right), \tag{3.23}$$

$$\frac{\partial q}{\partial B_n} = +q\Im\left(\frac{1}{\zeta^n}\right). \tag{3.24}$$

The detailed derivations of the previous three equations are contained in Appendix B.

The gradients of several other aerodynamic variables of interest are derived in Appendix C. The gradients of the lift coefficient, the moment coefficient, and the airfoil area were calculated using the discrete equations. These derivatives are not formally derived in the text, but the final forms can be found in the subroutines 'FinddCl' and 'FindArea' of Appendix D.

3.3.2 Geometry Gradient

Now, all aerodynamic variables that depend on the flow solution have been differentiated, and only one further piece of gradient information is necessary; the geometry sensitivity. In theory, this can be found analytically using the integrated form of Eq. (3.1), however, in practice there is a complication. The difficulty arises due to the need for the geometry gradient for the unit chord airfoil. Equation (3.1), when integrated, does not produce a unit chord airfoil and therefore its gradient will not be for a unit chord. The geometry is subsequently normalized, however this is not completely satisfactory for the gradient due to movement of the leading edge. This is not a concern for symmetric airfoils and is a relatively small effect for cambered airfoils. Therefore, the movement of the leading edge point was ignored in calculations of ∇z. The gradient of z was derived by differentiating the discrete geometry equations in XFOIL. This derivation is not included in the text, but the final results can be found in the subroutines 'ZAcalc' and 'ZAnorm' in Appendix D. The first routine calculates the gradient and the second of these routines normalizes the results to a unit chord airfoil.

3.4 Updating BL Variables

The Newton system of XFOIL uses the BL variables of the previous solution as the starting point of the new solution, therefore, the speed of the optimization can be increased by simply approximating the BL variables of the new airfoil. This can be done by adding the following perturbations to the BL variables at the old optimization step:

$$\{\delta\} = [D]\{\Delta\}. \tag{3.25}$$

The ΔA_n's and ΔB_n's in the $\{\Delta\}$ vector of Eq. (3.25) are the changes in the design variables between the current and new optimization steps, and are calculated from Eq. (2.4). The remaining two perturbations, δu_e and $\delta\delta^*$, can be found using

$$\delta u_e = \sum_{n=2}^{N_A} \frac{\partial u_e}{\partial A_n} \Delta A_n + \sum_{n=2}^{N_B} \frac{\partial u_e}{\partial B_n} \Delta B_n, \tag{3.26}$$

and

$$\delta \delta^* = \sum_{n=2}^{N_A} \frac{\partial \delta^*}{\partial A_n} \Delta A_n + \sum_{n=2}^{N_B} \frac{\partial \delta^*}{\partial B_n} \Delta B_n. \tag{3.27}$$

For a reasonable optimization step size this linear extrapolation will give a good approximation to the new BL variables. Thus, the Newton system constructed during the analysis of the new design point will converge faster than if no updating were done since it will have a better initial condition.

There is, however, a difficulty with this simple approach. Movement of the upper and lower surface transition points from one panel to another will cause such severe changes in the BL variables that this linear extrapolation will not work near the transition points. Figure 3.4 shows C_τ and m plotted versus the BL nodes (the leading edge is on the left, the trailing edge on the right) along the upper surface of the airfoil. The solid lines are the functions at the current optimization step and are included for reference. The dotted lines are the functions for the new optimization step. The functions approximated using Eq. (3.25) are plotted as dashed lines. Notice how accurate the approximation is, except near the transition point. If not considered separately, the poor transition point approximations would be enough to negate the gains in efficiency promised by the updating.

The solution is to approximate the new location of the transition points and then to 'fudge' the BL variables at each panel the transition points have passed over. This 'fudging' process will only affect the rate at which the Newton system converges, it will not affect the converged solution. For C_τ, θ, and u_e the approximation across the transition point shift is a linear extrapolation from the previous two approximated points. The equations for C_τ are

$$C_{\tau_i} = 2C_{\tau_{i-1}} - C_{\tau_{i-2}}, \tag{3.28}$$

where i is a BL node the transition point has passed over. The equations for θ and u_e are similar. For the remaining two BL variables, m and δ^*, it was found to be a better approximation not to extrapolate over the transition point shift using a tangent to the two previous points, but instead to set $m_i = m_{i-1}$ and $\delta^*_i = \delta^*_{i-1}$. The success of these updating procedures are evident in plots of C_τ and m as shown in Fig. 3.5. All that remains to be able to use these transition point approximations is to determine how far the transition point has shifted. This is done using

$$\delta x_{tran} = \frac{\partial x_{tran}}{\partial C_\tau}\delta C_\tau + \frac{\partial x_{tran}}{\partial \theta}\delta\theta + \frac{\partial x_{tran}}{\partial \delta^*}\delta\delta^* + \frac{\partial x_{tran}}{\partial u_e}\delta u_e. \tag{3.29}$$

All the derivative terms in the above are calculated in XFOIL to construct the Newton system, so the derivation is complete.

The convergence histories for a simple test case with and without updating the BL variables is shown in Fig. 3.6. The number of iterations for the Newton solver to convergence is plotted versus the optimization step number. The amount of time saved is not extensive, but the low cost of updating makes it worthwhile. Notice that as the optimization continues the savings will be smaller since the step sizes are small.

3.5 Multi-point Optimization

The development so far has assumed a single point optimization, i.e. optimizing for one flight condition. However, the equations derived are general enough to allow multi-point calculations. The only addition necessary is to modify the function being optimized (the Penalty Function) to be a weighted sum of the Penalty Functions at individual design points. Let P be the global Penalty Function, p_m be the local Penalty Functions, w_m the weighting function, and Np the number of design points. The optimization function is

$$P = \sum_{m=1}^{Np} w_m p_m. \tag{3.30}$$

The local penalty functions are computed in the same manner as before. The global penalty must use either constant angle of attack local penalty functions, or constant lift coefficient local penalty functions. There are no other restrictions, however, constant lift coefficient multi-design point optimizations did not perform as well as constant angle of attack optimizations. A schematic representation of a multi-point design with $N_p = 7$ is shown in Fig. 3.7. This multi-point modification was included since optimizing an airfoil with only one design point in mind may lead to poor off-design performance. One unanswered question is how to choose the weighting function. This will be left to other investigators, but to show the idea of multi-point optimization an example that uses an arbitrary weighting is included in the next chapter.

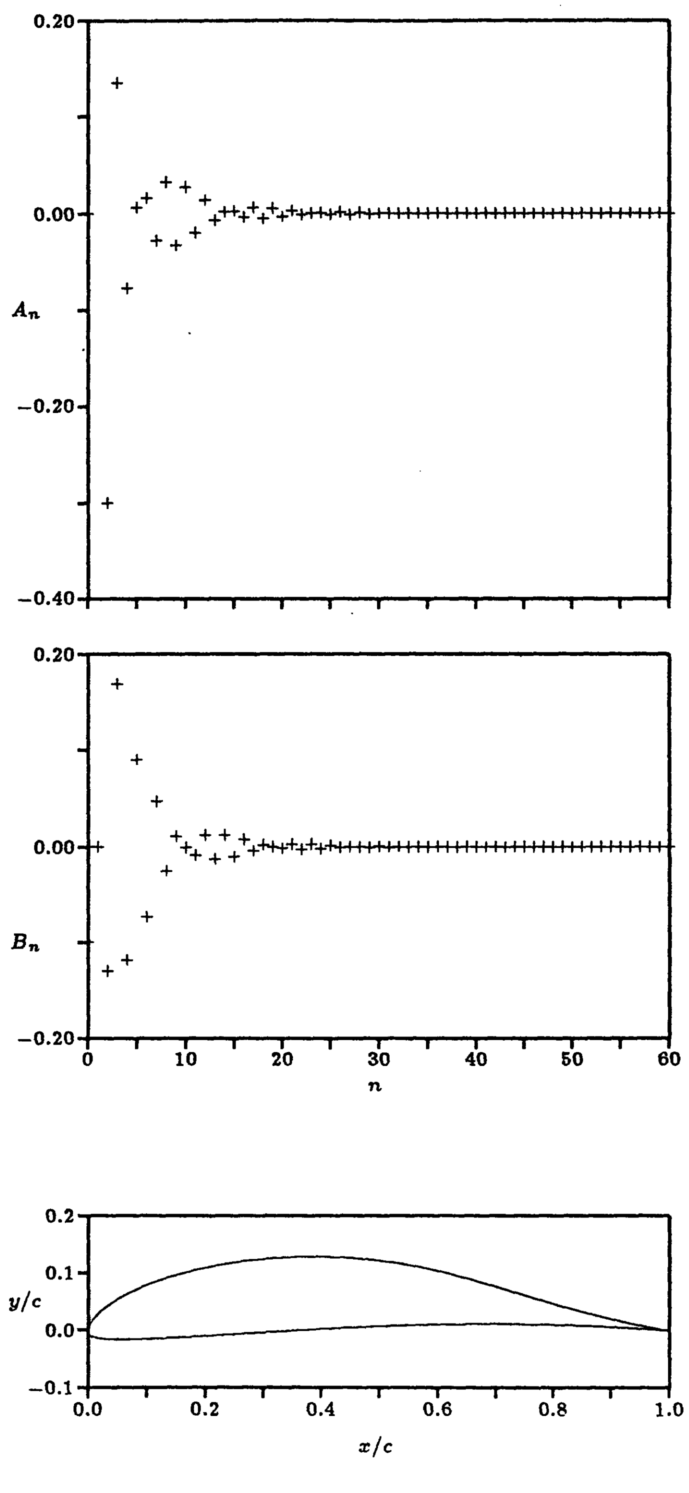

Figure 3.1: A_n and B_n Distributions for a DAE11 Airfoil

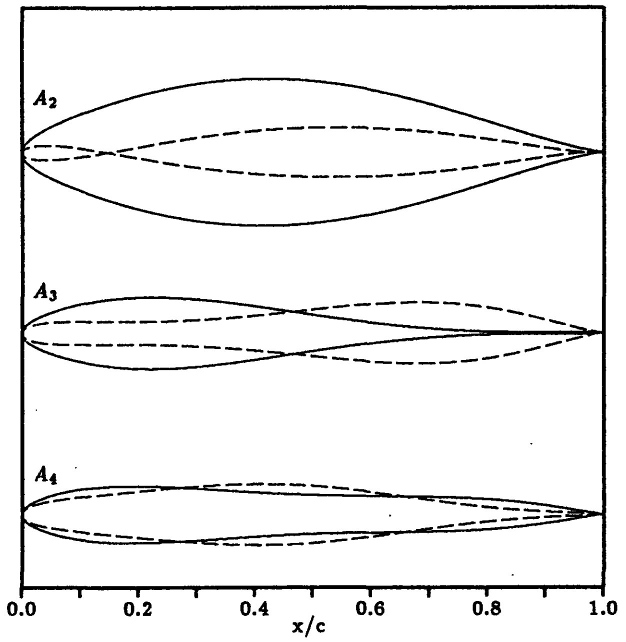

Figure 3.2: First Three Symmetric Design Modes

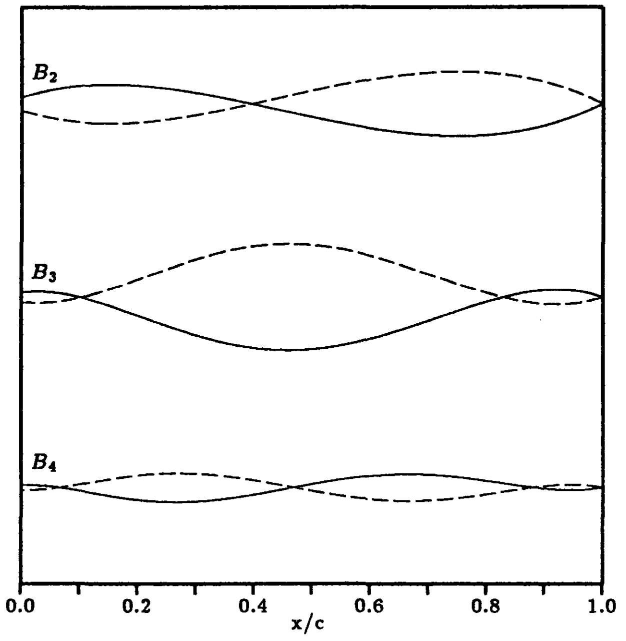

Figure 3.3: First Three Anti-Symmetric Design Modes

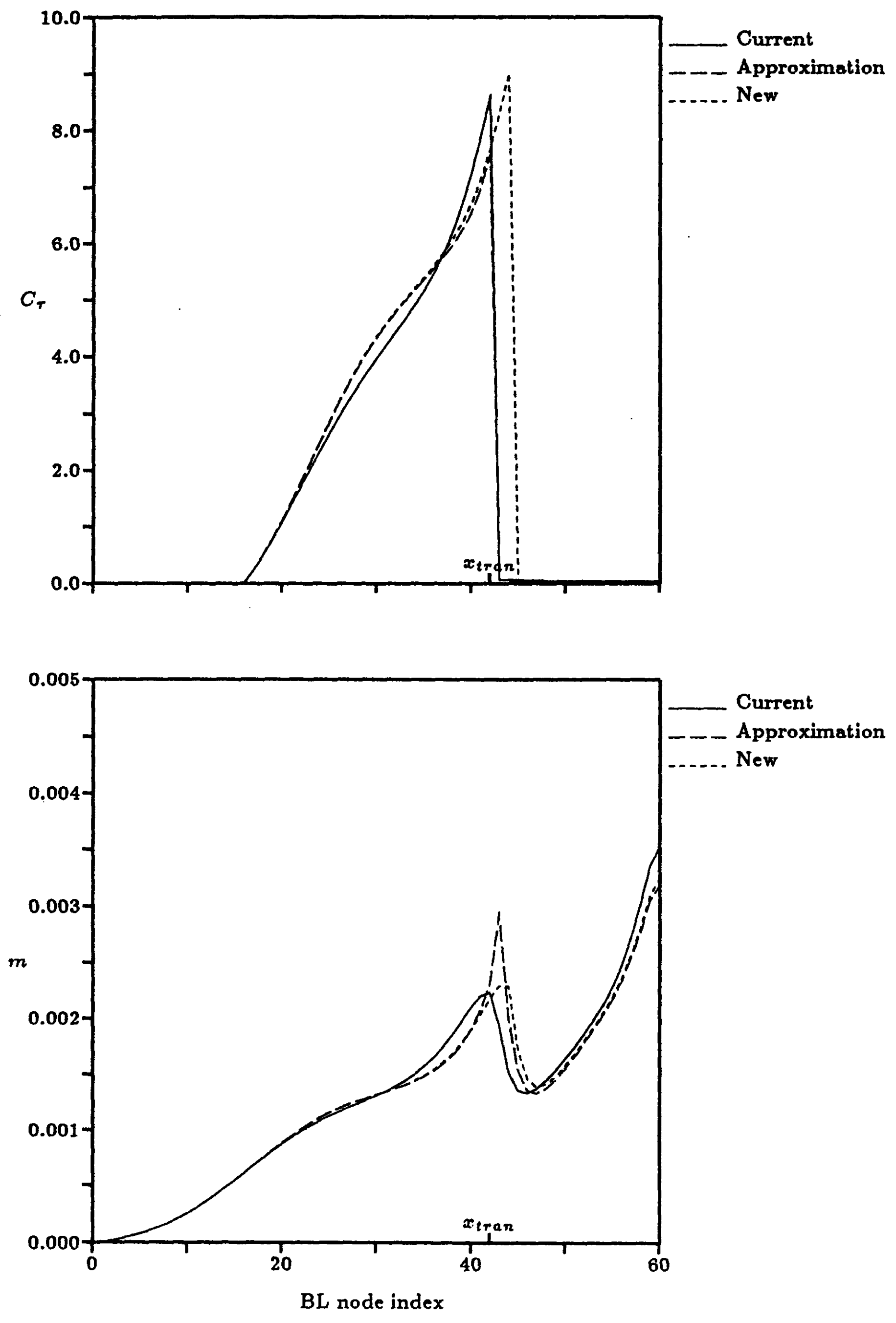

Figure 3.4: C_τ and m Over Upper Surface Ignoring Transition Point

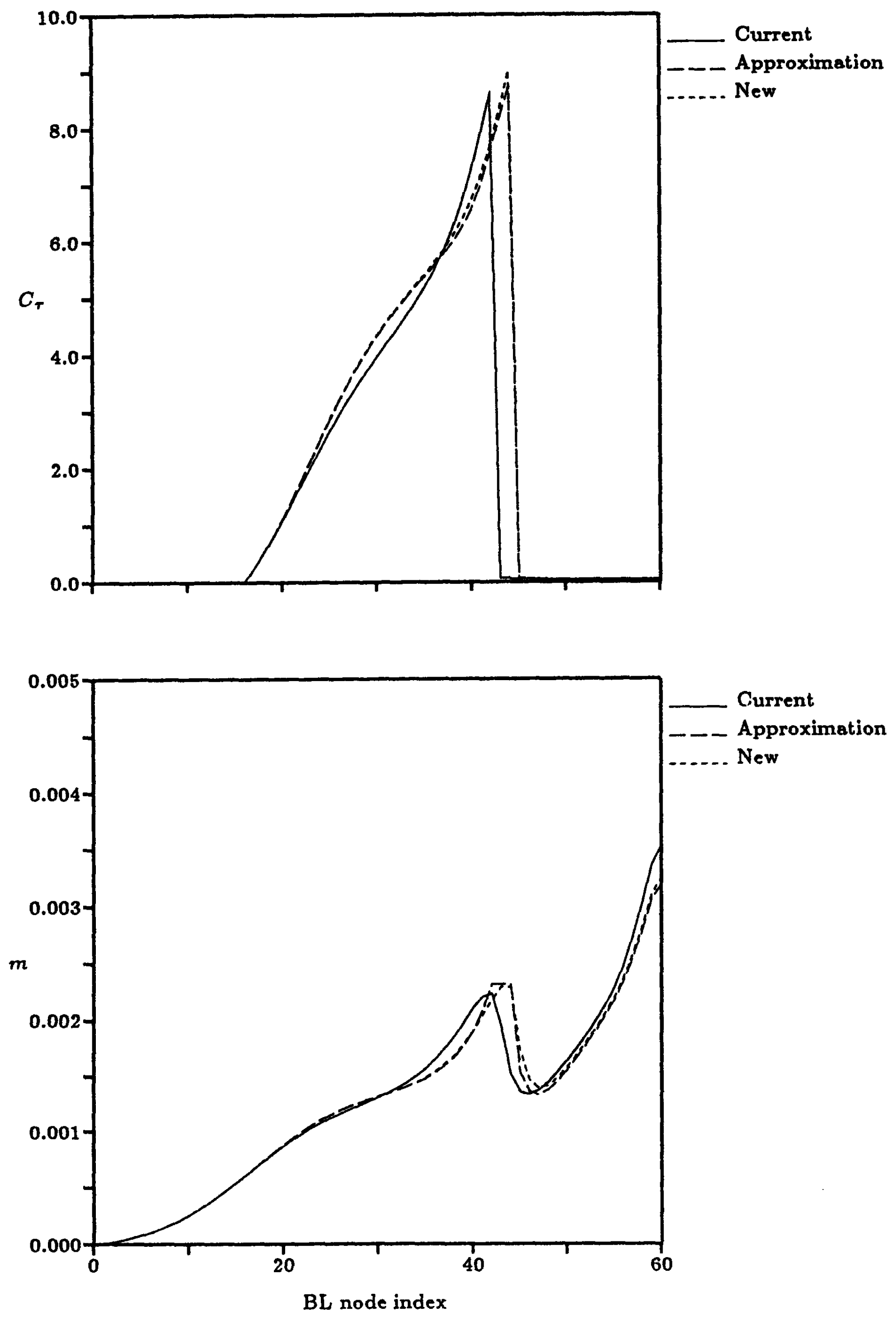

Figure 3.5: C_τ and m Over Upper Surface Taking Transition Point Into Account

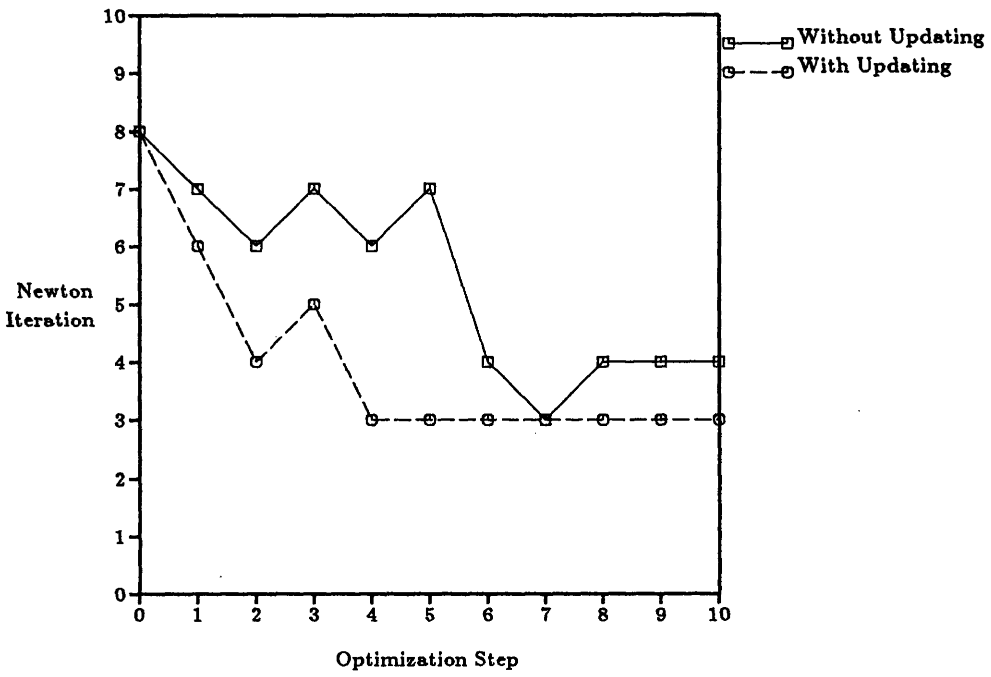

Figure 3.6: Convergence History With and Without Updating

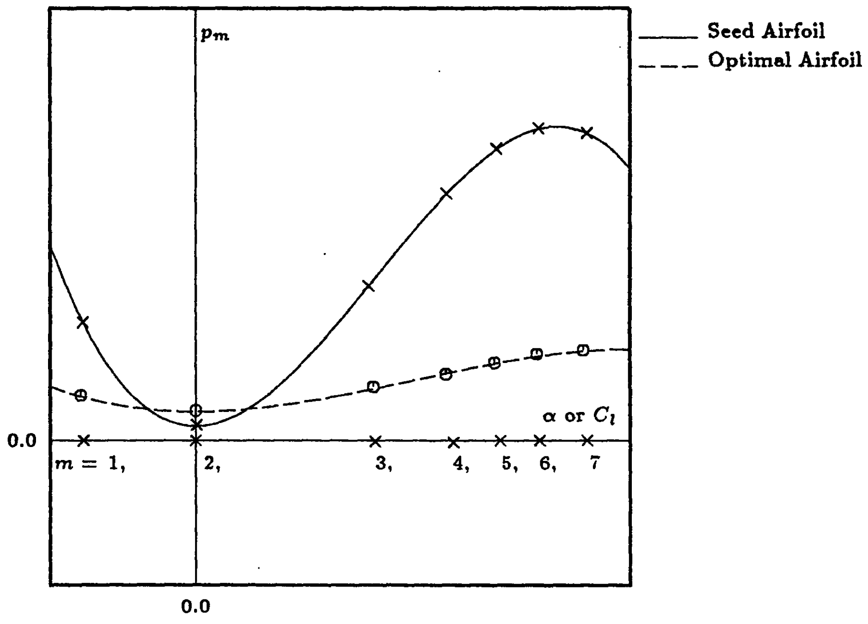

Figure 3.7: Multi-point Design Function Schematic

Chapter 4
Results

Four examples will be presented in this section. These examples were chosen to show the various properties of XFOIL's optimizer, they are not designed to be realistic design problems.

4.1 Example 1 - C_d minimization, $M = 0$, $\alpha = 0°$

The first test case was designed as a simple example to build faith in the optimization code. A NACA 0015 airfoil was used as the seed airfoil with C_d used as the Objective Function. The only constraint was to keep the angle of attack constant at 0°. The Reynolds Number based on the chord was 10^6. The two design variables used were A_2 and A_3. Using only two design variables will allow a pictorial representation of the optimization path to be constructed.

Figure 4.1 portrays the optimization space for this test case. The contours are of constant C_d and a local minimum is located in the upper left corner. The seed airfoil is located out of the picture in the lower right corner and the path taken by the optimizer is marked by the crosses. Convergence took 24 iterations and approximately 12 minutes on an 80 MFLOP machine. Figure 4.1 clearly shows the larger step sizes in the first five steps, i.e. in the region of large slope. The step directions are perpendicular to the contours, as they should be, where the gradients are large. As the optimum is neared the step directions start to parallel the contours. This is due to the approximations made in the gradient calculations. This is not a detriment since the exact mathematical optimum is relatively unimportant.

From Fig. 4.2 it is obvious that the largest drag reductions are produced in the first few iterations. This is a recurrent observation. Figure 4.3 compares the optimal airfoil to the seed airfoil. Because only two design modes were utilized, the possible change in the airfoil is small. However, large changes were made in C_d by modifying the airfoil such that the transition points were moved further aft.

4.2 Example 2 - C_d minimization, $M = 0$, $C_l = 0.5$

The second example optimized the C_d of an airfoil using 7 symmetric and 5 anti-symmetric design modes. The seed airfoil was an NACA 3412 and was constrained for a constant lift coefficient and a minimum allowed thickness at 95% of the chord. This constraint was necessary to prevent negative thickness airfoils. The cost parameter and the Reynolds number were $\kappa = 100$ and $Re = 5 \times 10^6$.

This example was stopped after a viscous Newton system was unconverged at the 38th optimization iteration. The Penalty Function and constraint histories are shown in Figs. 4.4 and 4.5. The drag reduction slows slightly after 20 iterations but is definitely still headed down when the optimizer was stopped. This indicates that the airfoil was not yet close to the optimum so the optimizer was restarted using the last airfoil generated before the Newton system failed as the seed airfoil. Optimization convergence was achieved after an additional 15 iterations. The drag was further lowered from $C_d = 0.00389$ to $C_d = 0.00380$. The reason for the unconverged Newton system is unexplained but it does not invalidate the results of the optimizer.

The inequality constraint history clearly shows the overshoot inherent in the Penalty Function technique. The small overshoot can easily be absorbed by adding a small amount of slack to the constraint.

The pressure plots of the seed and optimized airfoils are shown in Figs. 4.6 and 4.7, respectively. The dashed lines in the C_p curves are the inviscid solutions and the solid lines the viscous solutions. The waviness apparent in the C_p curve of the optimized airfoil is due to the fact that higher design modes were not used during the optimization.

4.3 Example 3 - Constrained Area

The third example uses both inequality and equality constraints. The optimization conditions are the same as Example 1 except that 10 symmetric design modes were used. The constraints were a minimum thickness near the trailing edge and a specified area. The area specified was 0.1 which is slightly less than the area of NACA 0015. The solution was obtained after 28 iterations.

The reduction in P is similar to the previous examples (see Fig. 4.8). The slowdown starts after the tenth iteration. Figure 4.9 shows how quickly the area equality constraint is satisfied. The thickness constraint remains inactive throughout the optimization process. The airfoils are compared, with the y/c-coordinate expanded, in Fig. 4.10. With the area constrained, and only the thickness modes used, there is not much room for change in the airfoil. However, the optimizer produces an airfoil with a "laminar-airfoil" shape, thus reducing the drag.

4.4 Example 4 - 5 Point Optimization

The final example was the optimization of an airfoil for minimum C_d at 5 angles of attack with $M = 0.4$ and $Re = 5 \times 10^6$. The seed airfoil was an NACA 3412 with 7 symmetric and 5 anti-symmetric design modes being utilized. The angle of attacks and weighting used were arbitrary and are listed in Table 4.1.

m	α	w_m
1	$-1.0°$	0.5
2	$0.0°$	0.5
3	$1.0°$	1.0
4	$3.0°$	3.0
5	$4.0°$	0.5

Table 4.1: Multi-point Optimization Weighting Distribution

For comparison, a one-point optimization was performed using the same conditions except that a single angle of attack of 3 degrees was used. The C_l vs. C_d polars are shown in Fig. 4.11 for the seed airfoil, the 5 point optimized airfoil, and the 1 point optimized airfoil. The only constraint used was a minimum thickness near the trailing edge. A pronounced low drag bucket has formed in both optimizations, however, the bucket is larger for the 5 point design, indicating better off-design performance for the multi-point design. The bucket formed in the region of the heaviest weighting, i.e. at $\alpha = 3°$. The multi-point optimization ended after a Newton system convergence failed at the 14^{th} optimization step. The single point design converged after 20 iterations. All three airfoils are shown in Fig. 4.12. The two optimized airfoils are very similar, but this is very much dependent upon the weighting functions, which, in these cases, were both chosen to be heavy at $\alpha = 3°$.

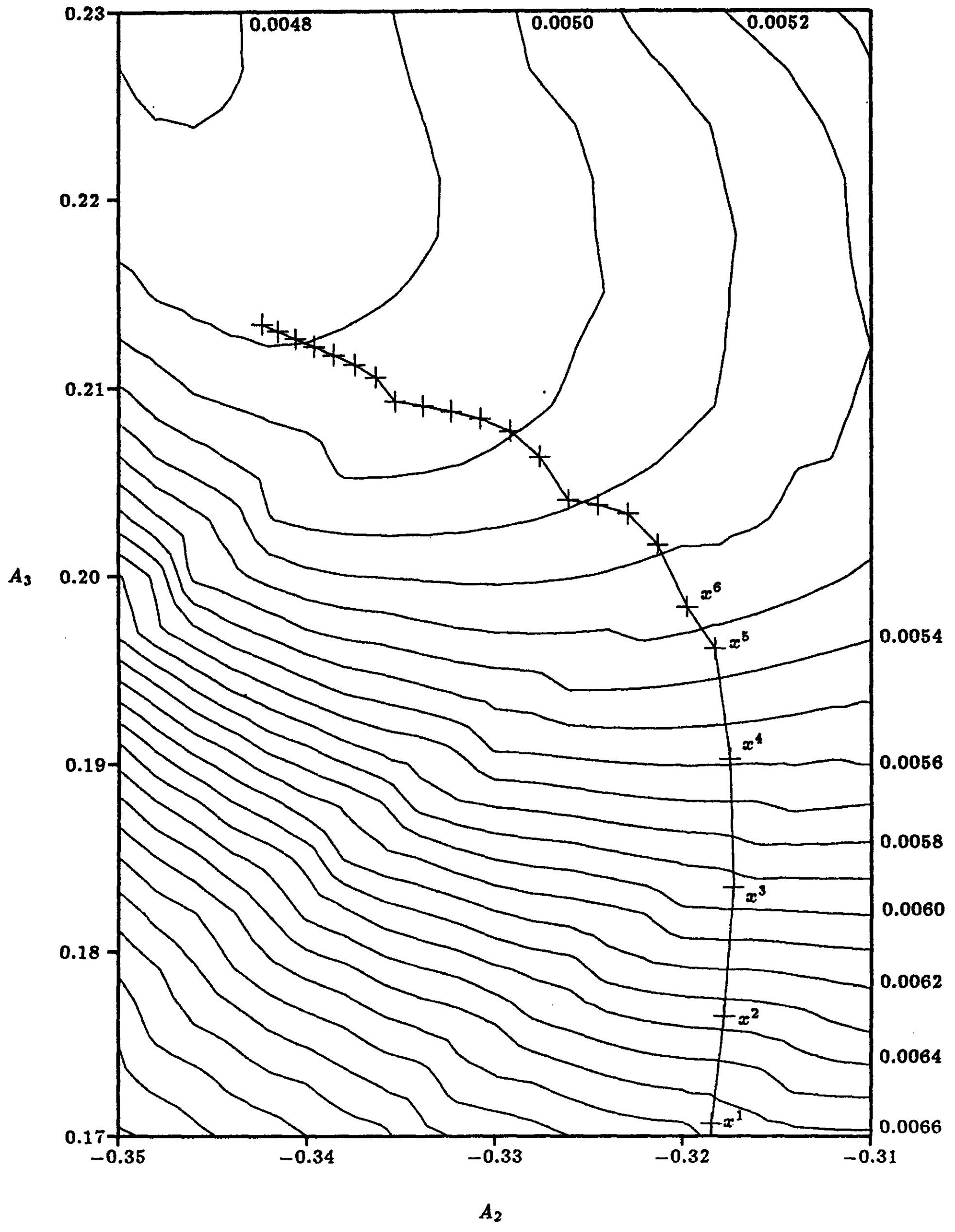

Figure 4.1: Example 1 - Optimization Path

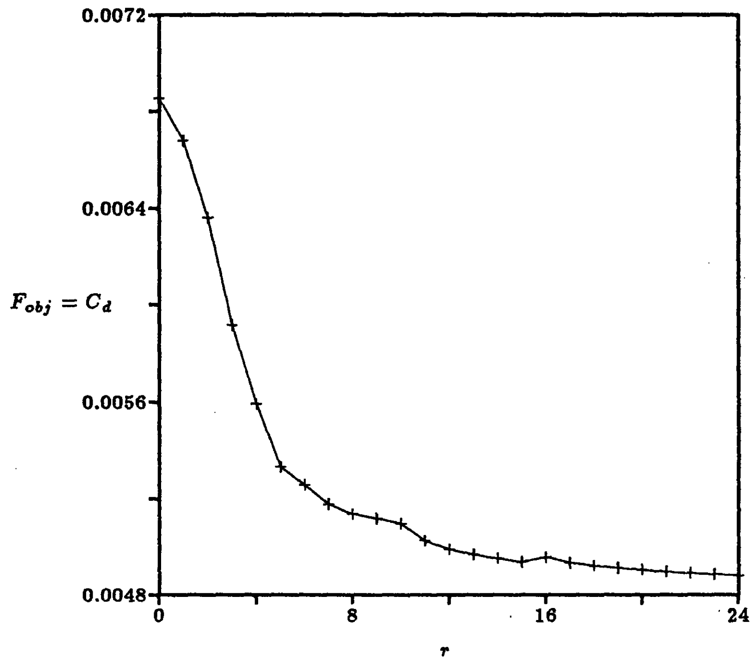

Figure 4.2: Example 1 - Optimization History

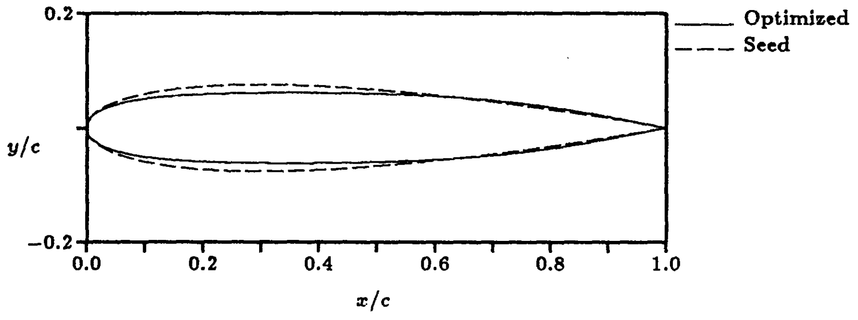

Figure 4.3: Example 1 - Airfoil Comparisons

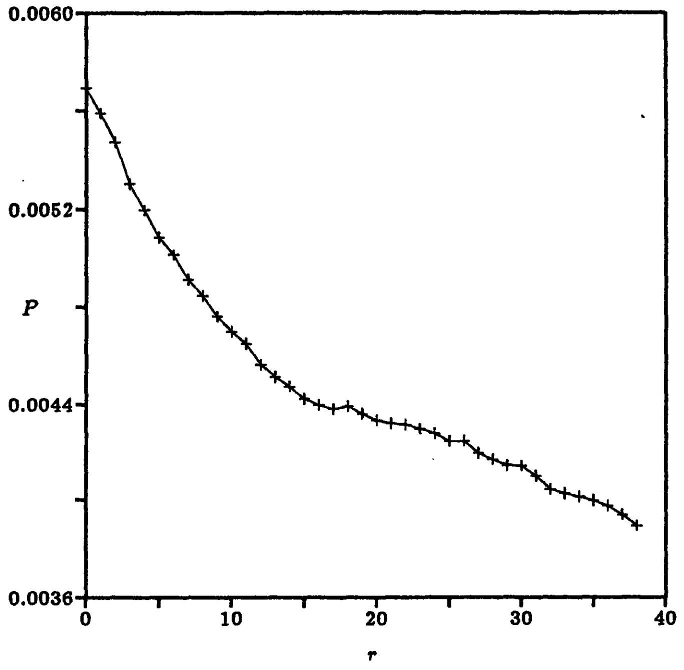

Figure 4.4: Example 2 - Optimization History

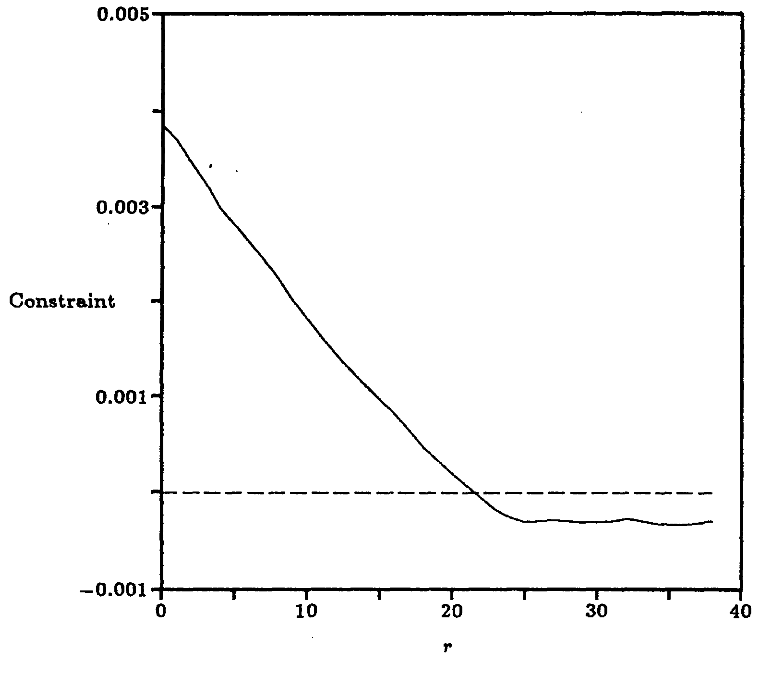

Figure 4.5: Example 2 - Thickness Constraint History

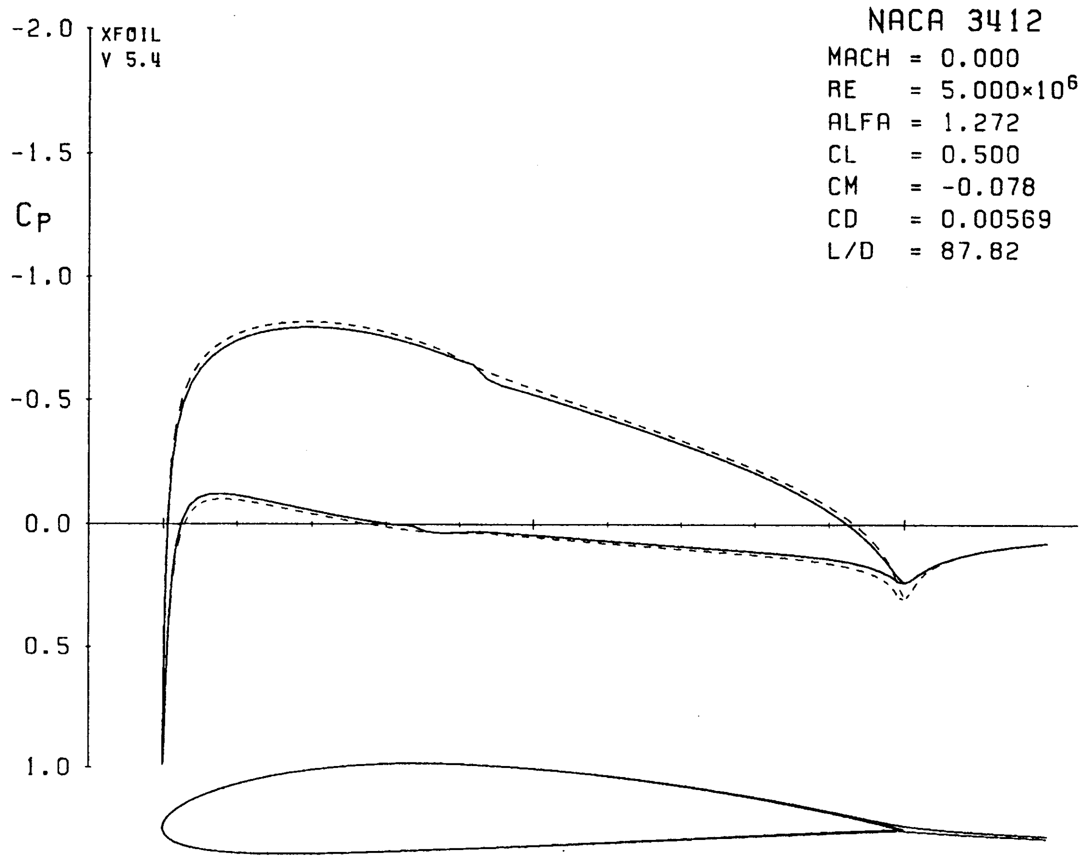

Figure 4.6: Example 2 - NACA 3412 C_p Plot

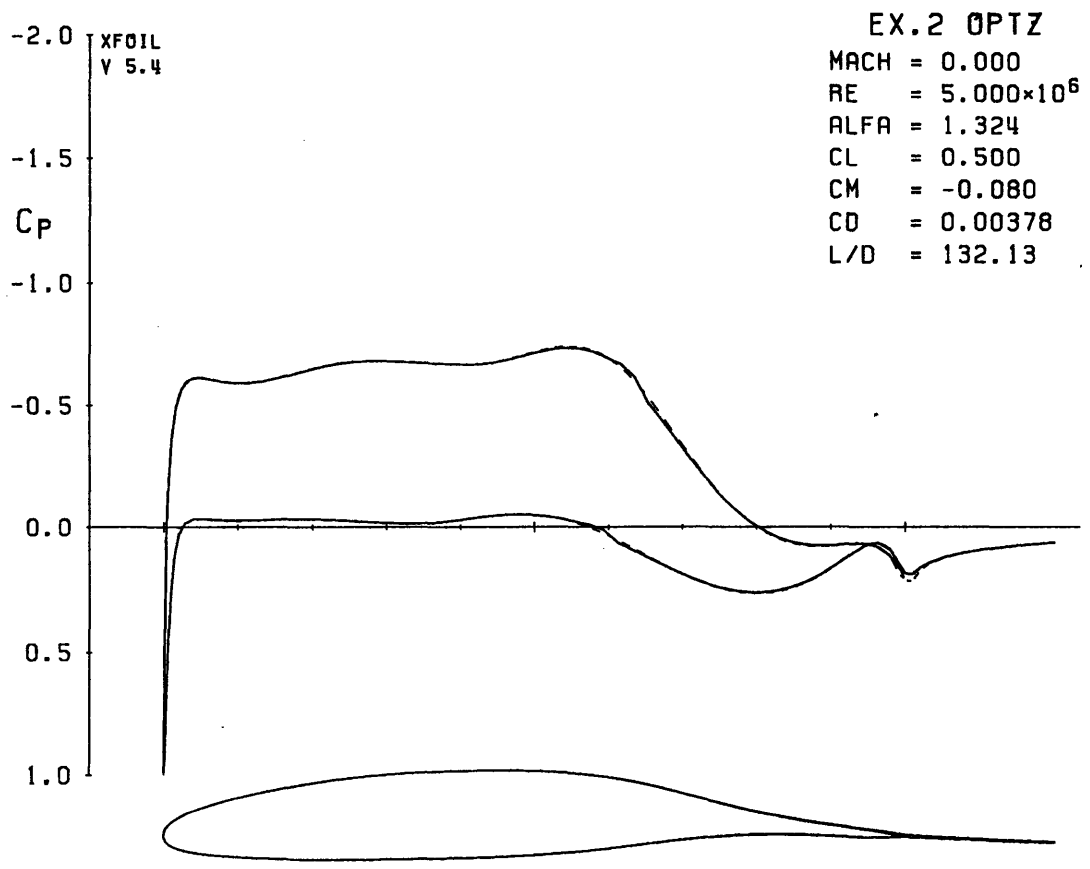

Figure 4.7: Example 2 - Optimized Airfoil C_p Plot

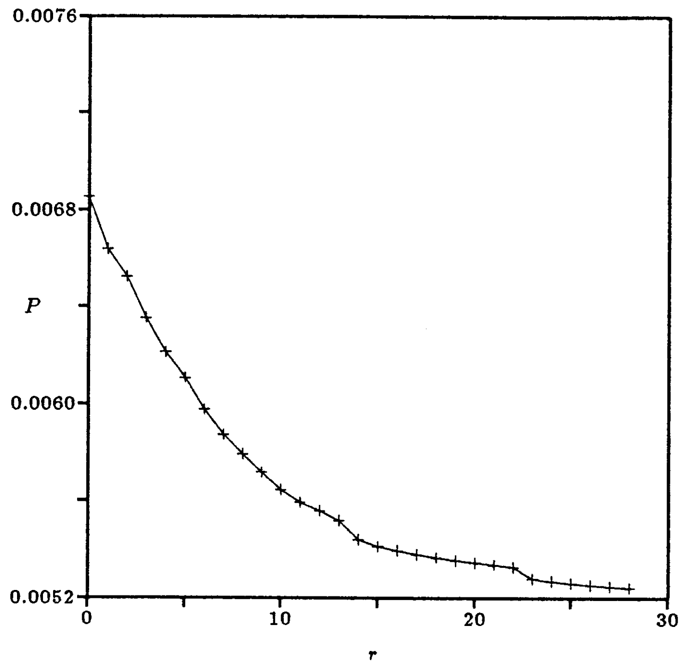

Figure 4.8: Example 3 - Optimization History

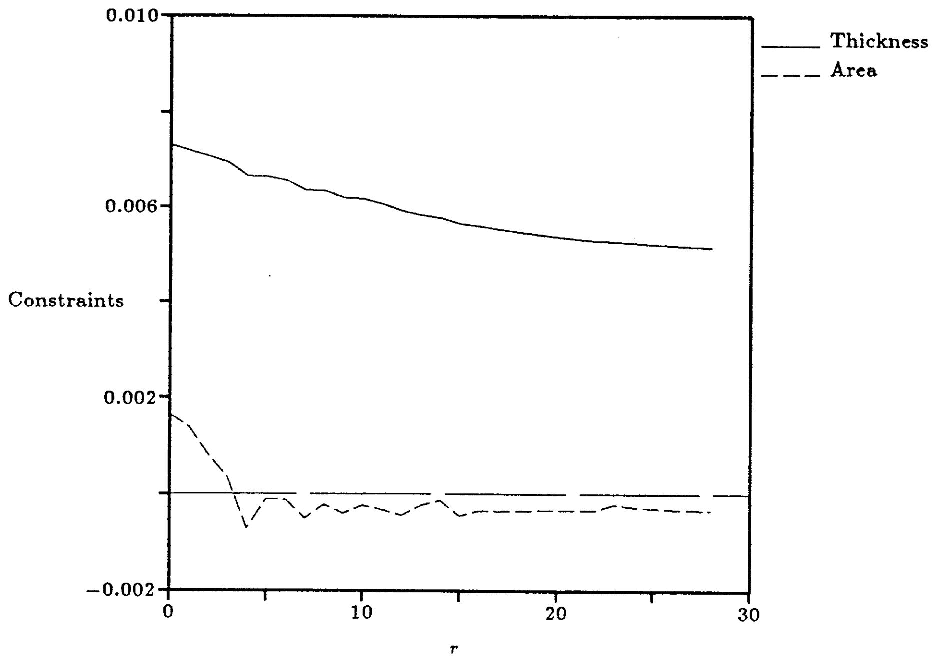

Figure 4.9: Example 3 - Constraint Histories

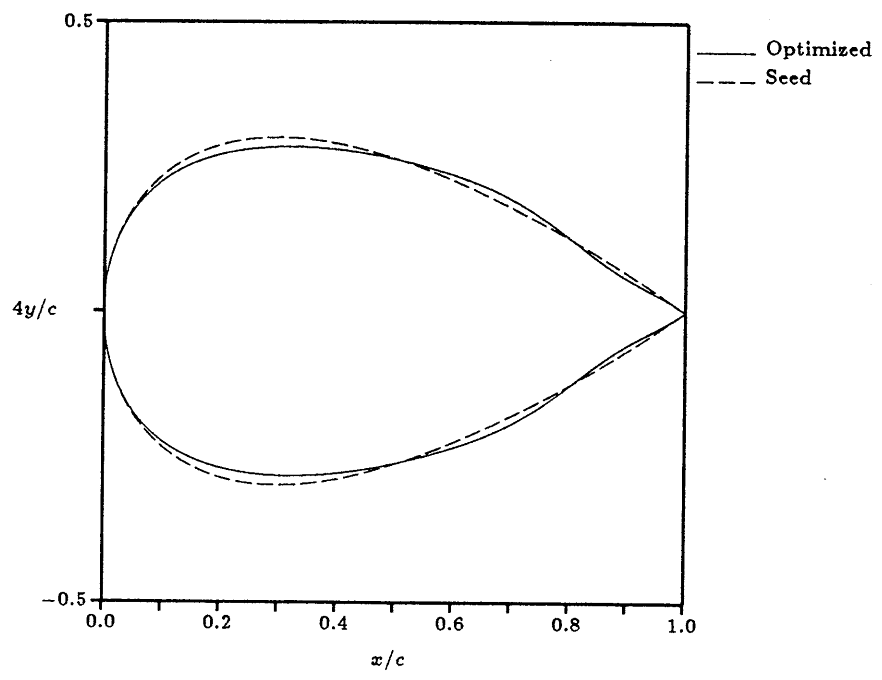

Figure 4.10: Example 3 - Airfoil Comparisons

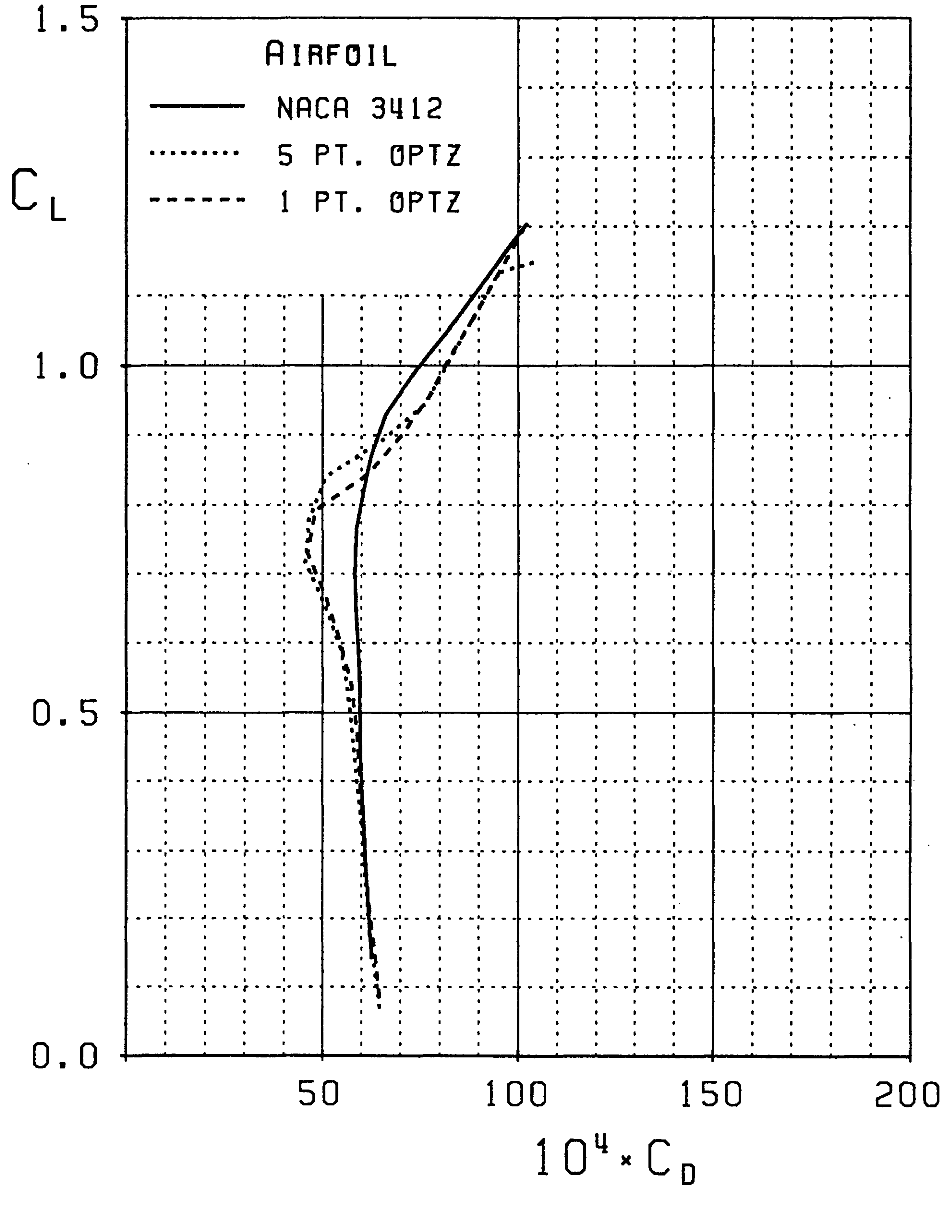

Figure 4.11: Example 4 - Drag Polars

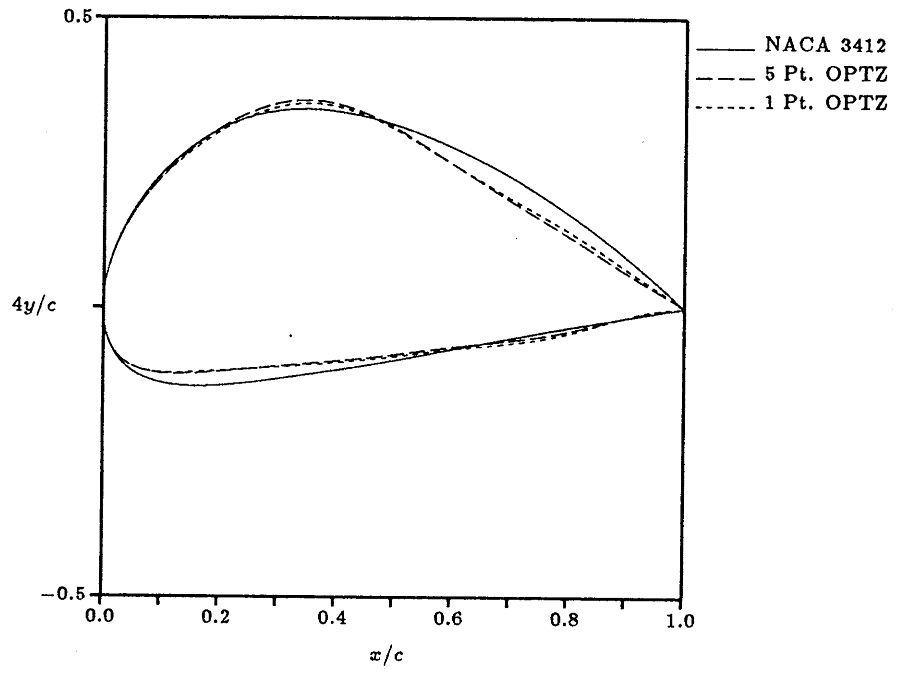

Figure 4.12: Example 4 - Airfoil Comparisons

Chapter 5
Conclusions

Modification of an airfoil design code to use mapping coefficients as the design variables was successfully implemented. Gradient information was calculated within the analysis portion of the code with a minimum of extra effort. The gradient information was shown to be accurate except as the optimum is neared. The low accuracy near the optimum is not considered to be vital since pushing an optimization to the mathematical optimum is probably fruitless due to the overconstrained nature of airfoil design.

When used in the proper way, the XFOIL optimizer can become a valuable design tool. The optimizer should not be used as a 'black box' to create perfect airfoils but as a designer's tool that will free the designer to become more creative and productive by reducing the time spent in iterative design modifications. The 'optimal' airfoils obtained should be used to give the designer ideas for what characteristics the real airfoil should have.

There were also several areas in which the XFOIL optimizer did not live up to expectations. The first is the limited number of design variables that could be utilized. It was found that the optimizer should be restricted to $N_A \leq 12$ and $N_B \leq 12$ because the higher mode derivatives became inaccurate. This does not allow the generation of completely general airfoils with the chosen design variables. This is a disappointment, however the cheap gradient calculations made possible by using the mapping coefficients as design variables make up for this deficiency. Another disappointment was the temperamental nature of XFOIL's Viscous Newton solver. This does not destroy the promise of the optimizer it only enforces that some care needs to be exercised when using the optimizer.

The multi-point optimization ability of the XFOIL optimizer may prove to be a valuable addition. However, the one piece of information needed to determine its value was not addressed in this thesis. The multi-point optimization is only as good as the weighting function used. This may be as simple as weighting the design points on the percentage of time the airfoil will be flying at each design point, or it may be a function based on a complex statistical scheme, or it may be based on the designers experience. The development of a systematic weighting function is an area requiring future research.

Another area for future research is the development of design variables that can also control the trailing edge angle and gap, and if possible, be completely general.

Bibliography

[1] M. Drela. XFOIL: An analysis and design system for low Reynolds number airfoils. In T.J. Mueller, editor, *Low Reynolds Number Aerodynamics*. Springer-Verlag, Jun 1989. Lecture Notes in Engineering, No. 54.

[2] M. Drela. Elements of airfoil design methodology. In P. Henne, editor, *Applied Computational Aerodynamics*, AIAA Progress in Aeronautics and Astronautics. AIAA, 1990.

[3] R. Eppler and D. M. Somers. A computer program for the design and analysis of low-speed airfoils. NASA TM 80210, Aug 1980.

[4] L. E. Madsen and G. N. Vanderplaats. COPES - A FORTRAN control program for engineering synthesis. NPS69-81-003 Naval Postgraduate School, March 1982.

[5] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes*. Cambridge University Press, Cambridge, 1986.

[6] M. S. Selig and M. D. Maughmer. A multi-point inverse airfoil design method based on conformal mapping. In *29th Aerospace Sciences Meeting*, Reno, Nevada, Jan 1991.

[7] G. Strang. *Introduction to Applied Mathematics*. Wellesley-Cambridge Press, Wellesley, 1986.

[8] J. L. Van Ingen. A program for airfoil section design utilizing computer graphics. In *AGARD-VKI Short Course on High Reynolds Number Subsonic Aerodynamics*, AGARD LS-37-70, April 1969.

[9] G. N. Vanderplaats. Efficient algorithm for numerical airfoil optimization. *Journal of Aircraft*, 16(12), Dec 1979.

[10] G. N. Vanderplaats. A new general-purpose optimization program. In *Structures, Structural Dynamics and Materials Conference*, Lake Tahoe, NV, May 1983.

[11] G. N. Vanderplaats. *Numerical Optimization Techniques for Engineering Design: with Applications.* McGraw-Hill, New York, 1984.

Appendix A
Derivation of $\frac{\partial R_i}{\partial A_n}$ and $\frac{\partial R_i}{\partial B_n}$

The BL equations, Eq. (3.10), can also be written in the form

$$R_i = R_i\left(\theta_{i-1}, \delta^*_{i-1}, u_{e_{i-1}}, C_{\tau_{i-1}}, x_{i-1}, \theta_i, \delta^*_i, u_{e_i}, C_{\tau_i}, x_i\right), \tag{A.1}$$

where, R_i can be f_i, g_i, or h_i. Using this functional formulation, a first-order Taylor series expansion of R_i is

$$\begin{aligned}\delta R_i &= \left(\tfrac{\partial R_i}{\partial \theta_{i-1}}\delta\theta_{i-1} + \tfrac{\partial R_i}{\partial \theta_i}\delta\theta_i\right) + \left(\tfrac{\partial R_i}{\partial \delta^*_{i-1}}\delta\delta^*_{i-1} + \tfrac{\partial R_i}{\partial \delta^*_i}\delta\delta^*_i\right)\\ &+ \left(\tfrac{\partial R_i}{\partial u_{e_{i-1}}}\delta u_{e_{i-1}} + \tfrac{\partial R_i}{\partial u_{e_i}}\delta u_{e_i}\right) + \left(\tfrac{\partial R_i}{\partial C_{\tau_{i-1}}}\delta C_{\tau_{i-1}} + \tfrac{\partial R_i}{\partial C_{\tau_i}}\delta C_{\tau_i}\right)\\ &+ \left(\tfrac{\partial R_i}{\partial x_{i-1}}\delta x_{i-1} + \tfrac{\partial R_i}{\partial x_i}\delta x_i\right).\end{aligned} \tag{A.2}$$

Using the definitions of u_e and δ^*:

$$u_{e_k} = q_k + \sum_j d_{kj} m_j, \tag{A.3}$$

$$\delta^*_k = \frac{m_k}{u_{e_k}}, \tag{A.4}$$

the following perturbations can be derived

$$\delta u_{e_k} = \delta q_k + \sum_j d_{kj}\delta m_j + \sum_j m_j \delta d_{kj}, \tag{A.5}$$

$$\delta\delta^*_k = \frac{\delta m_k}{u_{e_k}} - \frac{m_k}{u^2_{e_k}}\delta u_{e_k} = \frac{\delta m_k}{u_{e_k}} - \frac{m_k}{u^2_{e_k}}\left[\delta q_k + \sum_j d_{kj}\delta m_j\right]. \tag{A.6}$$

In the above, k can be i or $i-1$. The last term of Eq. (A.5) is insignificant compared to the other terms, so it is ignored. Substituting Eqs. (A.5 & A.6) into Eq. (A.2) and following only those terms with a δq, the equation has the form

$$\delta R_i = \left(\frac{\partial R_i}{\partial u_{e_{i-1}}} - \frac{\partial R_i}{\partial \delta^*_{i-1}} \frac{m_{i-1}}{u^2_{e_{i-1}}}\right) \delta q_{i-1} + \left(\frac{\partial R_i}{\partial u_{e_i}} - \frac{\partial R_i}{\partial \delta^*_i} \frac{m_i}{u^2_{e_i}}\right) \delta q_i + \cdots. \tag{A.7}$$

Therefore,

$$\frac{\partial R_i}{\partial q_{i-1}} = \left(\frac{\partial R_i}{\partial u_{e_{i-1}}} - \frac{\partial R_i}{\partial \delta^*_{i-1}} \frac{m_{i-1}}{u^2_{e_{i-1}}}\right), \tag{A.8}$$

$$\frac{\partial R_i}{\partial q_i} = \left(\frac{\partial R_i}{\partial u_{e_i}} - \frac{\partial R_i}{\partial \delta^*_i} \frac{m_i}{u^2_{e_i}}\right). \tag{A.9}$$

Using the chain rule the terms of $[A]$ are

$$\frac{\partial R_i}{\partial A_n} = \left(\frac{\partial R_i}{\partial q_{i-1}}\right)\left(\frac{\partial q_{i-1}}{\partial A_n}\right) + \left(\frac{\partial R_i}{\partial q_i}\right)\left(\frac{\partial q_i}{\partial A_n}\right), \tag{A.10}$$

$$\frac{\partial R_i}{\partial B_n} = \left(\frac{\partial R_i}{\partial q_{i-1}}\right)\left(\frac{\partial q_{i-1}}{\partial B_n}\right) + \left(\frac{\partial R_i}{\partial q_i}\right)\left(\frac{\partial q_i}{\partial B_n}\right). \tag{A.11}$$

As a reminder, R_i can be f_i, g_i, or h_i.

Appendix B
Derivation of $\frac{\partial q}{\partial A_n}$ and $\frac{\partial q}{\partial B_n}$

The derivation of q is presented in several of the references [3, 6, 8], but is included here for completeness. Refer to Fig. B.1. The complex potential for an inviscid flow around the unit circle in the ζ-plane is:

$$F(\zeta) = e^{-i\alpha}\zeta + e^{i\alpha}\zeta^{-1} - \frac{\Gamma}{2\pi i}\ln\zeta, \tag{B.1}$$

where,

$$\Gamma = 4\pi\sin\alpha = \frac{4\pi}{2i}\left(e^{i\alpha} - e^{-i\alpha}\right). \tag{B.2}$$

Differentiating Eq. (B.1) gives:

$$\frac{dF}{d\zeta} = (1 - \frac{1}{\zeta})\left(e^{-i\alpha} + e^{i\alpha}\zeta^{-1}\right). \tag{B.3}$$

The complex velocity conjugate is:

$$w^* = \frac{dF}{dz} = \frac{dF/d\zeta}{dz/d\zeta} = qe^{-i\theta}, \tag{B.4}$$

where,

$$q = \sqrt{u_x^2 + u_y^2} = |w|. \tag{B.5}$$

Taking the natural logarithm of the last two parts of Eq. (B.4) gives

$$\ln q - i\theta = \ln\left(\frac{dF}{d\zeta}\right) - \ln\left(\frac{dz}{d\zeta}\right). \tag{B.6}$$

Substituting in Eqs. (B.3 & 3.1) gives

$$\ln q - i\theta = \epsilon_{te} \ln\left(1 - \frac{1}{\zeta}\right) + \ln\left(e^{-i\alpha} + e^{i\alpha}\zeta^{-1}\right) - \sum_{n=0}^{\infty} C_n \zeta^{-n}. \tag{B.7}$$

To isolate q, first take the real part of Eq. (B.7) and then the inverse logarithm of both sides. The result is

$$q = \exp\left\{\Re\left[\ln\left(\left(1 - \frac{1}{\zeta}\right)^{\epsilon_{te}} \left(e^{-i\alpha} + e^{i\alpha}\zeta^{-1}\right)\right) - \sum_{n=0}^{\infty} C_n \zeta^{-n}\right]\right\}. \tag{B.8}$$

This result is valid everywhere in the flow field, and reduces on the airfoil, where $\zeta = e^{i\omega}$, to

$$q = 2\cos\left(\frac{\omega}{2} - \alpha\right)\left[2\sin\frac{\omega}{2}\right]^{\epsilon_{te}} e^{-P}, \tag{B.9}$$

where,

$$P + iQ = \sum_{n=0}^{\infty} C_n \zeta^{-n}. \tag{B.10}$$

The derivatives are most easily computed by differentiating the real part of Eq. (B.6). Doing the A_n derivatives:

$$\frac{d}{dA_n}(\ln q) = \Re\left\{\frac{\partial}{\partial A_n}\left[\ln\left(\frac{dF}{d\zeta}\right)\right] - \frac{\partial}{\partial A_n}\left[\ln\left(\frac{dz}{d\zeta}\right)\right]\right\}. \tag{B.11}$$

The first term is zero, so that

$$\frac{1}{q}\frac{\partial q}{\partial A_n} = -\Re\left\{\frac{1}{\frac{dz}{d\zeta}}\frac{\partial}{\partial A_n}\left(\frac{dz}{d\zeta}\right)\right\}. \tag{B.12}$$

Since $\frac{\partial(dz/d\zeta)}{\partial A_n} = \frac{dz}{d\zeta}\zeta^{-n}$ this reduces to

$$\frac{\partial q}{\partial A_n} = -q\Re\left\{\zeta^{-n}\right\}. \tag{B.13}$$

A similar process for $\frac{\partial}{\partial B_n}$ will give

$$\frac{\partial q}{\partial B_n} = q\Im\left\{\zeta^{-n}\right\}. \tag{B.14}$$

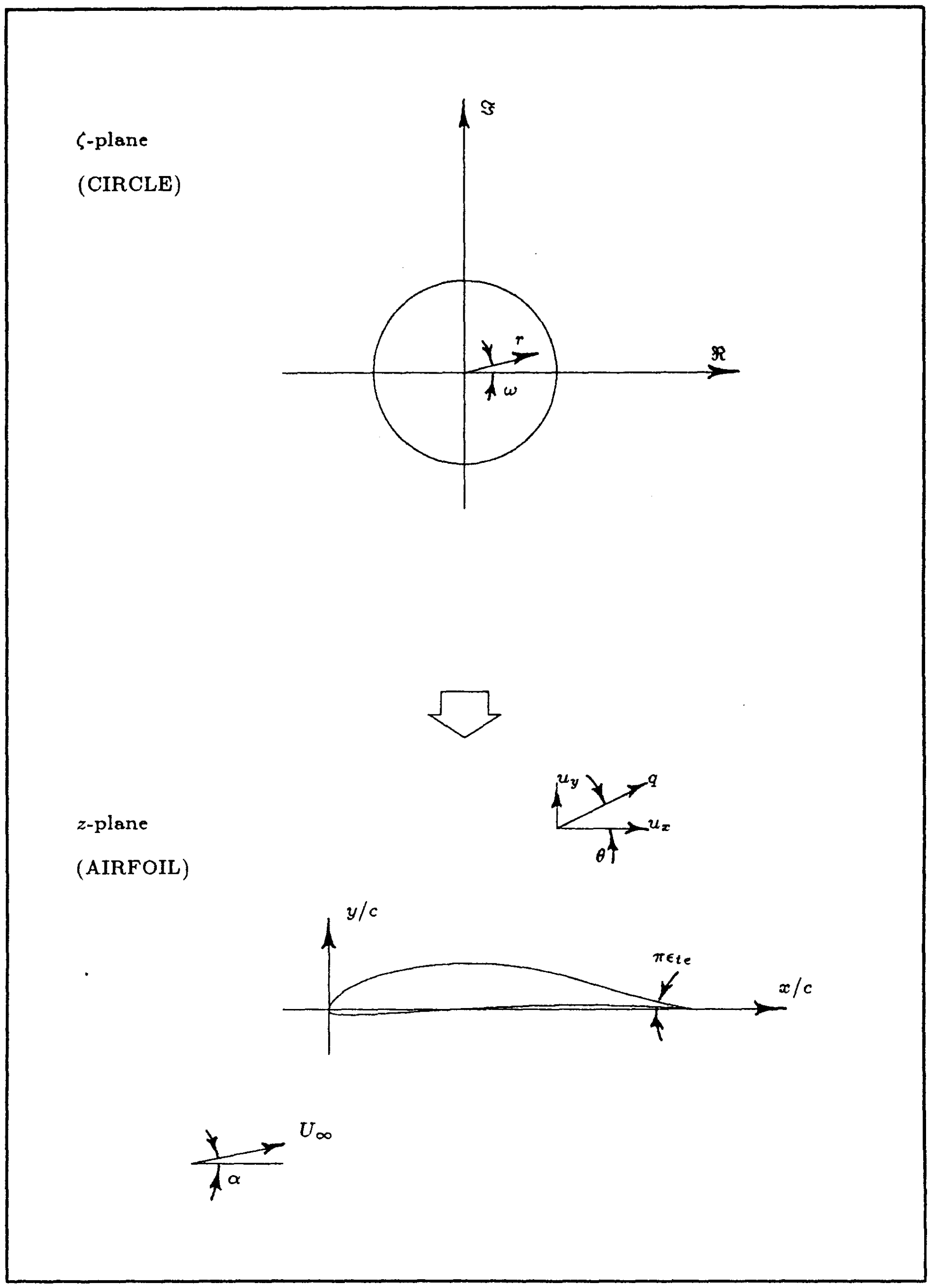

Figure B.1: Mapping Planes

Appendix C
Derivation of Gradients

The displacement thickness, δ^*, and the shape function, H, are calculated using

$$\delta^* = \frac{m}{u_e}, \tag{C.1}$$

and,

$$H = \frac{\delta^*}{\theta}. \tag{C.2}$$

Therefore, the derivatives are:

$$\frac{\partial \delta^*}{\partial A_n} = \frac{1}{u_e}\frac{\partial m}{\partial A_n} - \frac{m}{u_e^2}\frac{\partial u_e}{\partial A_n}, \tag{C.3}$$

$$\frac{\partial \delta^*}{\partial B_n} = \frac{1}{u_e}\frac{\partial m}{\partial B_n} - \frac{m}{u_e^2}\frac{\partial u_e}{\partial B_n}, \tag{C.4}$$

$$\frac{\partial H}{\partial A_n} = \frac{1}{\theta}\frac{\partial \delta^*}{\partial A_n} - \frac{\delta^*}{\theta^2}\frac{\partial \theta}{\partial A_n}, \tag{C.5}$$

$$\frac{\partial H}{\partial B_n} = \frac{1}{\theta}\frac{\partial \delta^*}{\partial B_n} - \frac{\delta^*}{\theta^2}\frac{\partial \theta}{\partial B_n}. \tag{C.6}$$

The derivatives of the C_l/C_d ratio are:

$$\frac{\partial}{\partial A_n}\left(\frac{C_l}{C_d}\right) = \frac{C_l}{C_d}\left(\frac{1}{C_l}\frac{\partial C_l}{\partial A_n} - \frac{1}{C_d}\frac{\partial C_d}{\partial A_n}\right), \tag{C.7}$$

$$\frac{\partial}{\partial B_n}\left(\frac{C_l}{C_d}\right) = \frac{C_l}{C_d}\left(\frac{1}{C_l}\frac{\partial C_l}{\partial B_n} - \frac{1}{C_d}\frac{\partial C_d}{\partial B_n}\right). \tag{C.8}$$

Appendix D
Computer Code

D.1 Code added to XFOIL for optimization

D.1.1 Added Declarations

The variables added to XFOIL for the optimization process are contained in 'xfoil.inc' and 'xbl.inc'. The following blocks of code are additions to the file 'xfoil.inc':

```
      parameter (NumOut   = 26,  NumMin   = 88, NumDump  = 30)
      parameter (MaxIt    = 50,  MaxPts   = 10, maxConst = 10)
      parameter (maxAn    = 30,  maxBn    = 30)
      parameter (ifirstAn = 2,   ifirstBn = 2)
      parameter (maxRHS   = 2 + maxAn + maxBn)
c
      real          kost, MinAbs, Mpts
      complex       za, za_an, za_bn, paq, paq_an, paq_bn
      complex       eaw, zle, zeta
      logical       Lthick, Lcamber, Loptz, LReStart, LTheEnd, Lconv
      character*50 OptzName
C
      common/dAn/ cl_an   (     ifirstAn:maxAn),
     &            cd_an   (     ifirstAn:maxAn),
     &            cm_an   (     ifirstAn:maxAn),
     &            za_an   (izx,ifirstAn:maxAn),
     &            ue_an   (izx,ifirstAn:maxAn),
     &            ds_an   (izx,ifirstAn:maxAn),
     &            qa_an   (izx,ifirstAn:maxAn),
     &            paq_an  (izx,ifirstAn:maxAn),
     &            area_an (     ifirstAn:maxAn),
     &            Fobj_An (     ifirstAn:maxAn),
     &            Fcost_An(     ifirstAn:maxAn)
c
      common/dBn/ cl_bn   (     ifirstBn:maxBn),
     &            cd_bn   (     ifirstBn:maxBn),
     &            cm_bn   (     ifirstBn:maxBn),
     &            za_bn   (izx,ifirstBn:maxBn),
     &            ue_bn   (izx,ifirstBn:maxBn),
     &            ds_bn   (izx,ifirstBn:maxBn),
     &            qa_bn   (izx,ifirstBn:maxBn),
     &            paq_bn  (izx,ifirstBn:maxBn),
     &            area_bn (    ifirstBn:maxBn),
     &            Fobj_Bn (    ifirstBn:maxBn),
     &            Fcost_Bn(    ifirstBn:maxBn)
c
      common/gem/ za(izx), wa(izx), paq(izx), eaw(izx,0:imx),
     &            zeta(izx), wcle, zle, snorm(izx)
c
      common/tof/ Lthick, Lcamber, Loptz, LReStart, LTheEnd, Lconv
c
      common/sup/ NumPts, weight(MaxPts), Apts(MaxPts),
     &            CLpts(MaxPts), Mpts(MaxPts), REpts(MaxPts)
c
      common/cst/ kost, Nconst, OnOff(MaxConst), Const(MaxConst),
     &            Const_an(MaxConst,ifirstAn:maxAn),
     &            Const_bn(MaxConst,ifirstBn:maxBn)
c
      common/add/ ilastAn, ilastBn, numAn, numBn, numRHS,
     &            IterConv, TheArea, TheLength, MinAbs,
     &            Fcubic, Fstart, grad2F(2), PercentMax,
     &            Fobj, Fcost(0:2), epsilon(0:2), jEnd(2:3),
     &            deltaAn(ifirstAn:maxAn), deltaBn(ifirstBn:maxBn),
     &            SdirAn (ifirstAn:maxAn), SdirBn (ifirstBn:maxBn),
     &            OptzName
c
c  **  Parameters
c      NumOut          file unit number of Optimization output file
c      NumMin                    "          dump file with minimum geometry
c      NumDump                   "          BL dump variable file
c      MaxIt           maximum allowed number of Optimization iterations
c      MaxPts          maximum            "         design points
c      maxConst        maximum            "         constraints
c      maxAn, Bn       maximum            "         An, Bn design variables
c      maxRHS          maximum            "         RHS's in Newton System
c      ifirstAn,Bn     = 2, first n value for all sensitivities
c
c  **  Sensitivities
c      cl_an, bn(.)    viscous d(Cl)/dAn, dBn of the airfoil   (n = 2,3,4...)
c      cd_an, bn(.)       "    d(Cd)/dAn, dBn           "
c      cm_an, bn(.)       "    d(Cm)/dAn, dBn           "
c      area_an, bn(.)     "    d(Area)/dAn, dBn         "
c
c      za_an, bn(..)   dz/dAn, dBn at each AIRFOIL PLANE node  (n = 2,3,4...)
c
c      ue_an, bn(..)   d(UEDG)/dAn, dBn at AIRFOIL PLANE nodes (n = 2,3,4...)
c      ds_an, bn(..)   d(DSTR)/dAn, dBn             "
c      qa_an, bn(..)   d(QC  )/dAn, dBn             "
c
c      Fobj_An,  Bn(.) d(Objective Function)/dAn, dBn          (n = 2,3,4...)
c      Fcost_An, Bn(.) d(Penalty   Function)/dAn, dBn                "
c
c  **  Geometry-mapping AT AIRFOIL NODES
c      za(.)           complex(x,y) at AIRFOIL nodes
c      wa(.)           CIRCLE plane w at AIRFOIL node mapped to a CIRCLE
```

```
c       paq(.)          P + iQ at AIRFOIL nodes
c       eaw(..)         exp(inw) at AIRFOIL nodes
c
c   **  Logical variables
c       Lthick          .true. if An terms to be used during optimization
c       Lcamber         .true. if Bn terms to be used during optimization
c       Loptz           .true. if in OPTZ routine (i.e. .true. if optimizing)
c       LReStart        .true. if Fletcher-Reeves Method is to be restarted
c       LTheEnd         .true. if optimum has been found
c       Lconv           .true. if ALL design point viscous sol'ns are converged
c
c   **  Multi-design point Optimization variables
c       NumPts          number of design points
c       weight(.)       relative weight of each design point
c       Apts(.)         angle of attack         "
c       CLpts(.)        Cl                      "
c       Mpts(.)         Mach number             "
c       REpts(.)        Reynolds number         "
c
c   **  Constraint variables
c       kost            cost parameter
c       Nconst          number of constraints (set by user)
c       OnOff(.)        0 => Active constraint, 1 => Inactive constraint
c       Const(.)        array of current constraint values
c       Const_an,bn(..) d(Const(.))/dAn, dBn                  (n = 2,3,4...)
c
c   **  Miscellaneous
c       ilastAn, Bn     = ifirstAn + numAn OR = ifirstBn + numBn
c       numAn, Bn       number of An, Bn terms to use during optimization
c       numRHS          = 2 + numAn + numBn (i.e. number of RHS in BL solver)
c       zeta(.)         zeta at each CIRCLE PLANE node
c       wcle            CIRCLE plane w of LE
c       zle             Complex coordinates of leading edge
c       snorm(.)        array holding normalized panel spacing
c       IterConv        number of newton iterations for viscous convergence
c       TheArea         Airfoil area
c       TheLength       Length of search direction vector before normalized
c       MinAbs          Absolute error for termination of optimization
c       Fstart          Objective function value of seed airfoil
c       grad2F(.)       grad(Fcost).grad(Fcost), 1 => previous iteration
c                                                2 => current  iteration
c       OptzName        Name of objective function, listed in output
c       PercentMax      Maximum allowed change in Cn's between optz iterations
c       Fobj            Objective Function
c       Fcost(1,2)      Penalty Function,        1 => previous iteration
c                                                2 => current  iteration
c       epsilon(.)      Step parameters
c       jEnd(.)         Number of consecutive matched end conditions
c       deltaAn,Bn(.)   An, Bn changes between optimization steps (n = 2,3,4...)
c       SdirAn, Bn(.)   Components of Search Direction Vector
```

The following block of code is an addition to the file 'xbl.inc':

```
      COMMON/XTRAN/ XT, XT_A1
     &    , XT_X1, XT_T1, XT_D1, XT_U1
     &    , XT_X2, XT_T2, XT_D2, XT_U2
     &    , xtt(2),    xtr_a1(2), xtr_x1(2)
     &    , xtr_t1(2), xtr_d1(2), xtr_u1(2)
```

D.1.2 Main Optimization Routines

The main optimization routine is contained in the new file 'xoptz.f' and is listed below, except for the data output routines:

```
      subroutine Optz
c
c   **********************************************************************
c   *  Written by Tom Sorensen, May 1991.                                *
c   *                                                                    *
c   *  This sub-program of XFOIL allows airfoil optimization to be       *
c   *  performed.  A seperate sub-program called 'XUSER' contains two    *
c   *  user  modifiable routines (for the objective function and         *
c   *  constraints).                                                     *
c   **********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      call SetOptz(2)          ! initialize optimization variables
      call XMDESinit           ! calculate Cn's & circle plane geometry
c
  500 CALL ASK ('.OPTZv^',1,COMAND)
c
      IF(COMAND.EQ.'    ') GO TO 10
      IF(COMAND.EQ.'RE  ') GO TO 20
      IF(COMAND.EQ.'MACH') GO TO 30
      IF(COMAND.EQ.'ALFA') GO TO 40
      IF(COMAND.EQ.'CL  ') GO TO 50
      IF(COMAND.EQ.'OBJ ') GO TO 60
      IF(COMAND.EQ.'OUT ') GO TO 70
      IF(COMAND.EQ.'DOIT') GO TO 80
      IF(COMAND.EQ.'OVAR') GO TO 90
      IF(COMAND.EQ.'?   ') WRITE(6,1050)
      IF(COMAND.EQ.'?   ') GO TO 500
c
      WRITE(6,1000) COMAND
      GO TO 500
c
c
c  ** Return to Top Routine
 10   call SetOptz(1)                ! nullify optimization variables
```

```
      close (unit = NumOut)          ! close output file
      close (unit = NumMin)          ! close min geometry file
      do 17 ipt = 1, NumPts
         close(unit = (NumDump+ipt)) ! close dump files
 17   continue
      return
c
c
c  ** Enter Reynolds Number
 20   do 25 ipt = 1, NumPts
         call ask ('Enter chord Reynolds number     ^',3,REINF)
         REpts(ipt) = REinf
 25   continue
      LVCONV = .FALSE.
      GO TO 500
c
c
c  ** Enter Mach Number
 30   do 35 ipt = 1, NumPts
 31      CALL ASK ('Enter Mach number                ^',3,MINF)
         Mpts(ipt) = Minf
         IF(MINF.GE.1.0) THEN
            WRITE(6,*) 'Supersonic freestream not allowed'
            GO TO 31
         ENDIF
         CALL COMSET
         IF(MINF.GT.0.0) WRITE(6,1300) CPSTAR, QSTAR/QINF
 35   continue
      LVCONV = .FALSE.
      GO TO 500
c
c
c  ** Enter Angle of Attack
 40   call ask ('Enter number of design points    ^',2,NumPts)
      if (NumPts.gt.MaxPts) then
         WRITE(6,*) 'Must use fewer points'
         go to 40
      endif
c
      do 45 ipt = 1, NumPts
         call ask ('Enter angle of attack (deg)      ^',3,Apts(ipt))
         call ask ('Enter weighting function         ^',3,weight(ipt))
         open (unit = (NumDump+ipt), status = 'scratch')
         if (REpts(ipt).le.1.0) REpts(ipt) = Reinf
 45   continue
      LALFA = .TRUE.
      GO TO 500
c
c
c  ** Enter Lift Coefficient
 50   call ask ('Enter number of design points    ^',2,NumPts)
      if (NumPts.gt.MaxPts) then
         WRITE(6,*) 'Must use fewer points'
         go to 50
      endif
c
      do 55 ipt = 1, NumPts
         call ask ('Enter lift coefficient          ^',3,CLpts(ipt))
         call ask ('Enter weighting function        ^',3,weight(ipt))
         open (unit = (NumDump+ipt), status = 'scratch')
         if (REpts(ipt).le.1.0) REpts(ipt) = Reinf
 55   continue
      LALFA = .FALSE.
      GO TO 500
c
c
c  ** Enter Objective Function
 60   write (6,3000)
      call ask ('Enter Optimization choice        ^',2,iType)
      if (iType.eq.1) OptzName = 'Cl'
      if (iType.eq.2) OptzName = 'Cd'
      if (iType.eq.3) OptzName = 'Cl/Cd'
      if (iType.eq.4) OptzName = 'other'
      GO TO 500
c
c
c  ** Enter output file names
 70   call  ask ('Enter Optz data output file name^',4,fname)
      open  (unit = NumOut, name = fname, type = 'new', err = 70)
c
 75   call  ask ('Enter min geometry  file name   ^',4,fname)
      open  (unit = NumMin, name = fname, type = 'new', err = 75)
      GO TO 500
c
c
c  ** Pick Optimization Method and start optimization
 80   call DoIt
      GO TO 500
c
c
c  ** Enter Optimization Variables
 90   call ask ('Enter number of An''s to use      ^',2,numAn)
      call ask ('Enter number of Bn''s to use      ^',2,numBn)
      call ask ('Enter maximum value of epsilon   ^', 3,epsilon(2))
      call ask ('Enter cost variable              ^', 3,kost)
c
      Lthick  = .false.
      Lcamber = .false.
      if (numAn.gt.0) Lthick  = .true.
      if (numBn.gt.0) Lcamber = .true.
      ilastAn = ifirstAn + numAn - 1
      ilastBn = ifirstBn + numBn - 1
      GO TO 500
C.....................................................................
C
 1000 FORMAT (1X,A4,' command not recognized.  Type a "?" for list')
 1050 FORMAT (/'   <cr>  Return to TOP LEVEL'
     &        /'   RE    Enter Reynolds Number'
     &        /'   MACH  Change Mach number'
     &        /'   ALFA  Prescribe alpha'
     &        /'   CL    Prescribe CL'
     &        /'   OBJ   Choose Object Function'
     &        /'   OUT   Choose Output file names'
     &        /'   DOIT  Start Optimization'
     &        /'   OVAR  Change Optimization Variables'/)

c
 1300 FORMAT (/'   Sonic Cp =', F12.2, '    Sonic Q/Qinf=', F12.3/)
c
 3000 format (/'   Optimize:'
     &        /'   1.) Cl'
     &        /'   2.) Cd'
     &        /'   3.) Cl/Cd'
     &        /'   4.) Other'/)
c
      end   !subroutine Optz




      subroutine SetOptz (icontrol)
c
c  ************************************************************************
c  *  This routine initializes variables needed for optimization.         *
c  *  icontrol = 1 => zero out variables for non-optimization use of XFOIL *
c  *  icontrol = 2 => assign variables values for optimization            *
c  ************************************************************************
c
      INCLUDE 'xfoil.inc'
c
      if (icontrol.eq.1) then
         numRHS     = 2
         numAn      = 0
         numBn      = 0
         Lthick     = .false.
         Lcamber    = .false.
         Loptz      = .false.
      else
         GREEK      = .FALSE.
         LReStart   = .true.
         Lconv      = .true.
c
c  **    set default values of airfoil parameters
         if (REinf.le.1.0) reinf = 1.0e6
         Apts(1)    = adeg
         Lalfa      = .true.
c
c  **    set default values of optimization control variables
         numAn      = 2
         numBn      = 0
         epsilon(2) = 0.1
         kost       = 10000
         ilastAn    = ifirstAn + numAn - 1
         ilastBn    = ifirstBn + numBn - 1
         PercentMax = 0.1
         Loptz      = .true.
         NumPts     = 1
         iType      = 2
         OptzName   = 'Cd'
c
         if (numAn.gt.0) Lthick  = .true.
         if (numBn.gt.0) Lcamber = .true.
c
c  **    zero delta's
         do 26 iAn = ifirstAn, ilastAn
            deltaAn(iAn) = 0.0
 26      continue
c
         do 88 iBn = ifirstBn, ilastBn
            deltaBn(iBn) = 0.0
 88      continue
c
c **     normalize aifoil paneling spline
         do 100 i = 1, N
            snorm(i) = s(i) / s(N)
 100     continue
      endif
c
      return
      end   !subroutine SetOptz




      subroutine XMDESinit
c
c  ********************************************************************
c  *  This routine computes Cn and other complex stuf from MDES.       *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
      INCLUDE 'xdes.inc'
C
C---- see if current Qspec, if any, didn't come from Mixed-Inverse
      IF(NSP.NE.NC)   LQSPEC = .FALSE.
      IF(.NOT.LSCINI) CALL SCINIT
C
C---- set initial Gamma for current alpha
      ALFGAM = ARAD
      CALL MAPGAM(1)
c
      return
      end   !subroutine XMDESinit
      subroutine DoIt
c
c  ********************************************************************
c  *  This routine performs the iterative optimization loop.          *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      iter     = 0              ! Optimization iteration counting variable
      jEnd(2)  = 0              ! number of iterations End Cond#2 satisfied
      jEnd(3)  = 0              ! number of iterations End Cond#3 satisfied
      LTheEnd  = .false.        ! .true. => Optimization finished
c
c  ** Evaluate SEED airfoil at all design points
      do 10 ipt = 1, NumPts
         if (Lalfa) then        ! optimize with constant angle of attack
            adeg   = Apts(ipt)
            arad   = dtor * adeg
         else                   ! optimize with constant lift coefficient
            arad   = 0.0
            CLSPEC = CLpts(ipt)
         endif
c
         REinf = REpts(ipt)     ! set Re at each design point
         Minf  = Mpts(ipt)      ! set M  at each design point
c
         call Analiz (ipt)      ! compute viscous solution
 10   continue
c
      Fstart = Fobj             ! save initial value of objective function
c
      call OutData  (iter)      ! print out relevant data
      call EndCheck (iter)      ! check ending conditions
c
c  ** Main optimization iteration loop
 26   if (.not.LTheEnd) then
         iter = iter + 1
c
         call Search            ! choose search direction
         call NextPt            ! choose and analize new optimization point
         call OutData  (iter)   ! print out relevant data
         call MinGeom           ! print out current optimal geom. to a file
         call EndCheck (iter)   ! check ending conditions
c
         go to 26
      endif
c
      return
      end   !subroutine DoIt
      subroutine Analiz (ipt)
c
c  *******************************************************************
c  *  This subroutine groups the routines needed to compute a viscous   *
c  *  solution for a given geometry.  In addition, it calls the         *
c  *  routines that find the sensitivities of the various variables.    *
c  *******************************************************************
c
      INCLUDE 'xfoil.inc'
c
      call Initializ          ! calculate wake coordinates
      call FindZeta           ! calculate zeta's     at airfoil nodes
      call FinddQA            ! calculate dq/dAn, Bn at airfoil nodes
      call FinddZ             ! calculate dz/dAn, Bn at airfoil nodes
      call VISCAL             ! calculate viscous solution
      call FinddUe            ! calculate dUe/dAn,Bn & dds/dAn,Bn
      call FinddCl            ! calculate dCl/dAn,Bn & dCm/dAn,Bn
      call FinddCd            ! calculate dCd/dAn,Bn
      call FinddArea          ! calculate dArea/dAn,Bn
      call constrain          ! calculate constraints
      call dFobj              ! calculate Fobj  and dFobj /dAn,dBn
      call dFcost (ipt)       ! calculate Fcost and dFcost/dAn,dBn
      call DumpOut(ipt)       ! dump BL variables to a scratch file
      CALL QVX                ! plot Cp distribution
c
      return
      end   !subroutine Analiz




      subroutine Initializ
c
c  ******************************************************************
c  *  This routine initializes airfoil and wake inviscid GAM and PSIO *
c  *  arrays.                                                         *
c  ******************************************************************
c
      INCLUDE 'xfoil.inc'
      INCLUDE 'xdes.inc'
C
c  ** computes wake coordinates
      LVISC  = .true.
      LVCONV = .FALSE.
      QINF   = 1.0
C
C---- calculate surface vorticity distributions for alpha = 0, 90 degrees
      IF(.NOT.LGAMU .OR. .NOT.LQAIJ) CALL GGCALC
c
      if (.not.Lalfa) then
C----    Newton loop for alpha to get specified inviscid CL
```

```
         DO 100 ITAL=1, 20
            COSA = COS(ARAD)
            SINA = SIN(ARAD)
C
C----       superimpose suitably weighted  alpha = 0, 90  distributions
            DO 50 I=1, N
               GAM(I) = COSA*GAMU(I,1) + SINA*GAMU(I,2)
   50       CONTINUE
            PSIO = COSA*GAMU(N+1,1) + SINA*GAMU(N+1,2)
C
C------     get corresponding CL and CL_alpha
            CALL CLCALC
C
            DARAD = (CLSPEC - CL) / CL_ALF
            ARAD  = ARAD + DARAD
C
            IF(ABS(DARAD).LE.1.0E-6) GO TO 110
  100    CONTINUE
         WRITE(6,*) 'SPECCL:  CL convergence failed'
  110    CONTINUE
      endif
c
C---- set final distribution (or distribution at specified alfa)
      COSA = COS(ARAD)
      SINA = SIN(ARAD)
      DO 40 I=1, N
        GAM(I) = COSA*GAMU(I,1) + SINA*GAMU(I,2)
   40 CONTINUE
      PSIO = COSA*GAMU(N+1,1) + SINA*GAMU(N+1,2)
C
      IF(ABS(ARAD-AWAKE) .GT. 1.0E-5) LWAKE  = .FALSE.
      IF(ABS(ARAD-AVISC) .GT. 1.0E-5) LVCONV = .FALSE.
c
      call XYWAKE
c
      return
      end   !subroutine Initializ
      subroutine findZeta
c
c ******************************************************************
c *  This subroutine computes zeta at the airfoil plane nodes       *
c *  added 4/26/90                                                  *
c ******************************************************************
c
      complex CnOHOLD, zleHOLD, zHOLD
c
      INCLUDE 'xfoil.inc'
c
c ** change Cn(0), Zle, and chordz only for finding zeta!
      CnOHOLD = Cn(0)
      zleHOLD = zle
      zHOLD   = chordz
c
      Cn(0)   = Cn(0) + cmplx(1.0 / (4.0 - real(zle)), 0.0)
      zle     = cmplx(0.0,0.0)
      chordz  = cmplx(1.0,0.0)
c
c ** assign zetas at trailing edge
      zeta(1) = cmplx(1.0,0.0)
      zeta(N) = cmplx(1.0,0.0)
c
c ** compute zetas corresponding to airfoil nodes
c ** do NOT compute zetas at trailing edge
      zeta(2) = cmplx(0.0,0.0)      ! initial guess for node 2
      do 26 ia = 2, N-1             ! ia = airfoil node number
         call zetaf (cmplx(x(ia), y(ia)), zeta(ia), ia)
         zeta(ia+1) = zeta(ia)      ! initial guess for next point
 26   continue
c
c ** compute zetas corresponding to wake nodes
      zeta(N+1) = cmplx(0.0,0.0)    ! initial guess for node N+1
      do 88 ia = N+1, N+Nw          ! ia = airfoil node number
         call zetaf (cmplx(x(ia), y(ia)), zeta(ia), ia)
         zeta(ia+1) = zeta(ia)      ! initial guess for next point
 88   continue
c
c ** change values back (these are global and used elsewhere as they were)
      Cn(0)  = CnOHOLD
      zle    = zleHOLD
      chordz = zHOLD
c
      return
      end   !subroutine findZeta
      subroutine finddQA
c
c  ********************************************************************
c  *  This subroutine computes dq/dAn and dq/dBn at the airfoil & wake  *
c  *  nodes added 4/19/90                                               *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
      real    qa(izx)
      complex CnSum, zetaInv, eAlfa, term1
c
C---- set alpha in the circle plane
      ALFCIR  = ALFgam - AIMAG(CN(0))
      eAlfa   = cexp (cmplx(0.0, ALFCIR))
c
c  ** calculate q at airfoil nodes,    ia = airfoil node number
      do 26 ia = 1, N + Nw
         zetaInv  = 1.0 / zeta(ia)
         term1    = ((1.0 / eAlfa) + eAlfa * zetaInv)
     &            * (1.0 - zetaInv)**AGTE
c
         CnSum    = Cn(mc)
         do 100 m = mc-1, 0, -1
            CnSum = Cn(m) + CnSum * zetaInv
 100     continue
c
         if ((ia.eq.1).or.(ia.eq.N)) then
c  **       q is zero at trailing edge
            qa(ia) = 0.0
         else
            qa(ia) = exp(real(clog(term1) - CnSum))
         endif
 26   continue
c
c  ** fudge TE q's to more closely approx. OPER answers
      call fudgeTE (qa(1), qa(2), qa(3), qa(N-2), qa(N-1), qa(N))
c
c  ** calculate dq/dAn,Bn at airfoil nodes,    ia = airfoil node number
      do 88 ia = 1, N + Nw
         zetaInv = 1.0 / zeta(ia)
c
         do 826 iAn = ifirstAn, ilastAn
            qa_an(ia,iAn) = -qa(ia) *  real(zetaInv**iAn)
            if ((iAn.eq.2).and.(ia.eq.2)) print*, qa_an(ia,ian)
 826     continue
c
         do 888 iBn = ifirstBn, ilastBn
            qa_bn(ia,iBn) =  qa(ia) * aimag(zetaInv**iBn)
            if ((iBn.eq.2).and.(ia.eq.2)) print*, qa_bn(ia,ibn)
 888     continue
 88   continue
c
      return
      end   !subroutine finddQA
      subroutine fudgeTE (qa1, qa2, qa3, qaN2, qaN1, qaN)
c
c  ******************************************************************
c  * This routine approximates the value of q at the trailing edge  *
c  * by a linear extrapolation of q near the trailing edge.  This   *
c  * is done to make q agree more closely with the XOPER results.   *
c  ******************************************************************
c
      INCLUDE 'xfoil.inc'
c
c  ** compute approximate qTE for suction side
      m     = (qa3 - qa2) / (x(3) - x(2))
      qTEs  = qa2 - m * (x(2) - x(1))
c
c  ** compute approximate qTE for pressure side
      m     = (qaN2 - qaN1) / (x(N-2) - x(N-1))
      qTEp  = qaN1 - m * (x(N-1) - x(N))
c
c  ** average qTEs and qTEp and assign to a(1) and q(N)
      qa1 = 0.5 * (qTEs + qTEp)
      qaN = qa1
c
      return
      end   !subroutine fudgeTE





      subroutine FinddZ
c
c  *********************************************************************
c  *  This routine groups the routines needed to compute               *
c  *  dz/dAn, Bn at the AIRFOIL nodes.                                 *
c  *********************************************************************
c
      include 'xfoil.inc'
c
      call eawSET        ! calculate exp(-inw)       at airfoil nodes
      call PaFilt(0.0)   ! calculate d(P+iQ)/dAn, Bn at airfoil nodes
      call zacalc        ! calculate dz/dAn, dBn     at airfoil nodes
      call zanorm        ! scale dx/dAn, Bn to a unit chord
c
      return
      end     !End FinddZ
      subroutine eawSET
c
c  *******************************************************************
c  * This routine saves exp(inw) for the AIRFOIL nodes to an array.    *
c  *******************************************************************
c
      include 'xfoil.inc'
c
      wa(1) = 0.0
      wa(N) = 2.0 * pi
c
c  ** compute w valuse at AIRFOIL nodes (not evenly spaced in circle plane)
      do 10 ia = 2, N-1
         wa(ia) = aimag(clog(zeta(ia)))
         if (wa(ia).lt.0.0) wa(ia) = wa(ia) + 2.0 * pi
   10 continue
c
c  ** calculate exp(inw)
      do 20 m = 0, mc
         do 30 ia = 1, N
            eaw(ia,m) = cexp(cmplx(0.0, float(m) * wa(ia)))
   30    continue
   20 continue
c
      return
      end     ! eawSET
```

```
      subroutine paFILT(FFILT)
c
c  *******************************************************************
c  *  This routine calculates d(P+iQ)/dAn, dBn at the AIRFOIL nodes    *
c  *  (with filtering).                                                *
c  *******************************************************************
c
      include 'xfoil.inc'
c
      real cwtx(0:imx)
C
C---- cut down higher transform coefficients with modified Hanning filter
      do 10 m = 0, mc
         freq = float(m) / float(mc)
c
         if (ffilt.gt.0.0) then
            cwtx(m) = (0.5 * (1.0 + cos(pi * freq)))**ffilt
         else
            cwtx(m) = (0.5 * (1.0 + cos(pi * freq)))
         endif
 10   continue
```

```
C
C---- Inverse-transform to get back modified speed function and its conjugate
      do 300 ia = 1, N
         paq(ia) = (0.0,0.0)
         do 310 m = 0, mc
            paq(ia) = paq(ia) + cwtx(m) * CN(M) * CONJG(eaw(ia,m))
  310    continue
c
         do 26 iAn = ifirstAn, ilastAn
            paq_an(ia,iAn) = cwtx(iAn) * conjg(eaw(ia,iAn))
 26      continue
c
         do 88 iBn = ifirstBn, ilastBn
            paq_bn(ia,iBn) = cwtx(iBn) * conjg(eaw(ia,iBn))
     &                    * cmplx(0.0,1.0)
 88      continue
  300 continue
C
      return
      end ! paFILT





      SUBROUTINE ZAcalc
c
c  *******************************************************************
c  *  This routine calculates dz/dAn, dBn at AIRFOIL nodes for        *
c  *  the airfoil of chord almost 4.                                  *
c  *******************************************************************
c
      INCLUDE 'xfoil.inc'
      COMPLEX dwa, DZDW1, DZDW2, DZ_paq1, DZ_paq2
C
c  ** compute d(za)/dAn,Bn (i.e. at airfoil plane nodes)
      ia     = 1
      za(ia) = (4.0,0.0)
c
c  ** zero out d(za)/dAn,Bn
      do 126 iAn = ifirstAn, ilastAn
         za_an(ia,iAn) = (0.0,0.0)
 126  continue
c
      do 188 iBn = ifirstBn, ilastBn
         za_bn(ia,iBn) = (0.0,0.0)
 188  continue
c
      SINW  = 2.0*SIN(0.5*wa(ia))
      SINWE = 0.
      IF(SINW.GT.0.0) SINWE = SINW**(1.0-AGTE)
C
```

```
      Hwa   = 0.5*(wa(ia)-PI)*(1.0+AGTE) - 0.5*PI
      DZDW1 = SINWE * CEXP( paq(ia) + CMPLX(0.0,Hwa) )
c
      DO 20 ia = 2, N
        SINW  = 2.0*SIN(0.5*wa(ia))
        SINWE = 0.
        IF(SINW.GT.0.0) SINWE = SINW**(1.0-AGTE)
C
        Hwa   = 0.5*(wa(ia)-PI)*(1.0+AGTE) - 0.5*PI
        DZDW2 = SINWE * CEXP( paq(ia) + CMPLX(0.0,Hwa) )
C
        dwa     = wa(ia) - wa(ia-1)
        za(ia)  = 0.5*(DZDW1+DZDW2)*dwa + za(ia-1)
        DZ_paq1 = 0.5*(DZDW1      )*dwa
        DZ_paq2 = 0.5*(      DZDW2)*dwa
C
        do 226 iAn = ifirstAn, ilastAn
           za_an(ia,iAn) = DZ_paq1 * paq_an(ia-1,iAn)
     &                   + DZ_paq2 * paq_an(ia  ,iAn)
     &                   + za_an(ia-1,iAn)
 226    continue
c
        do 288 iBn = ifirstBn, ilastBn
           za_bn(ia,iBn) = DZ_paq1 * paq_bn(ia-1,iBn)
     &                   + DZ_paq2 * paq_bn(ia  ,iBn)
     &                   + za_bn(ia-1,iBn)
 288    continue
C
        DZDW1 = DZDW2
   20 CONTINUE
C
      RETURN
      END ! ZAcalc




      SUBROUTINE ZAnorm
c
c     *******************************************************************
c     * This routine computes dz/dAn, dBn at the AIRFOIL nodes for the  *
c     * normalized airfoil.                                             *
c     *******************************************************************
c
      INCLUDE 'xfoil.inc'
c
      COMPLEX zaNEW, ZTE, za_ZTE
      complex zle_an(ifirstAn:maxAn), zle_bn(ifirstBn:maxBn)
      complex zte_an(ifirstAn:maxAn), zte_bn(ifirstBn:maxBn)
C
C---- find leftmost point
```

```
      XMIN = REAL(za(1))
      DO 30 ia = 1, N
        IF(REAL(za(ia)).LE.XMIN) THEN
         XMIN = REAL(za(ia))
         iaLE = ia
        ENDIF
   30 CONTINUE
C
C---- set restricted spline limits around leading edge
      ia1  = iaLE - 16
      ia2  = iaLE + 16
      Nia  = ia2 - ia1 + 1
      XTE  = 0.5* REAL(za(1) + za(N))
      YTE  = 0.5*AIMAG(za(1) + za(N))
      waLE = WCLE
C
c  ** compute effect of dAn on the LE
      do 126 iAn = ifirstAn, ilastAn
         do 26 ia = ia1, ia2
            i = ia - ia1 + 1
            w1(i) =  real(za_an(ia,iAn))
            w3(i) = aimag(za_an(ia,iAn))
 26      continue
c
c **     calculate spline near leading edge
         call spline (w1, w2, wa(ia1), nia)
         call spline (w3, w4, wa(ia1), nia)
c
c **     calculate leading edge zle_an derivative
         TheReal = seval(wale, w1, w2, wa(ia1), nia)
         TheImag = seval(wale, w3, w4, wa(ia1), nia)
c
         zle_an(iAn) = cmplx(TheReal, TheImag)
 126  continue
c
c  ** compute effect of dBn on the LE
      do 188 iBn = ifirstBn, ilastBn
         do 88 ia = ia1, ia2
            i = ia - ia1 + 1
            w1(i) =  real(za_bn(ia,iBn))
            w3(i) = aimag(za_bn(ia,iBn))
 88      continue
c
c **     calculate spline near leading edge
         call spline (w1, w2, wa(ia1), nia)
         call spline (w3, w4, wa(ia1), nia)
c
c **     calculate leading edge zle_bn derivative
         TheReal = seval(wale, w1, w2, wa(ia1), nia)
         TheImag = seval(wale, w3, w4, wa(ia1), nia)
c
         zle_bn(iBn) = cmplx(TheReal, TheImag)
 188  continue
c
```

```
C---- place leading edge at origin
      DO 60 ia = 1, N
         za(ia) = za(ia) - ZLE
         do 226 iAn = ifirstAn, ilastAn
            za_an(ia,iAn) = za_an(ia,iAn) - zle_an(iAn)
 226     continue
c
         do 288 iBn = ifirstBn, ilastBn
            za_bn(ia,iBn) = za_bn(ia,iBn) - zle_bn(iBn)
 288     continue
   60 CONTINUE
C
C---- set normalizing quantities and sensitivities
      ZTE = 0.5*(za(1) + za(N))
      do 326 iAn = ifirstAn, ilastAn
         zte_an(iAn) = 0.5*(za_an(1,iAn) + za_an(N,iAn))
 326  continue
c
      do 388 iBn = ifirstBn, ilastBn
         zte_bn(iBn) = 0.5*(za_bn(1,iBn) + za_bn(N,iBn))
 388  continue
C
C---- normalize sensitivities
      do 500 ia = 1, N
         zaNEW  = CHORDZ*za(ia)/ZTE
         za_ZTE = -zaNEW/ZTE
         za(ia) =  zaNEW
c
         do 426 iAn = ifirstAn, ilastAn
            za_an(ia,iAn) = chordz * za_an(ia,iAn) / zte
     &                    + za_zte * zte_an(iAn)
 426     continue
c
         do 488 iBn = ifirstBn, ilastBn
            za_bn(ia,iBn) = chordz * za_bn(ia,iBn) / zte
     &                    + za_zte * zte_bn(iBn)
 488     continue
 500  continue
C
      RETURN
      END ! ZAnorm
      subroutine FinddUe
c
c  ********************************************************************
c  *  this subroutine computes due/dAn, due/dBn, ddstar/dAn,Bn and     *
c  *  dH/dAn,Bn at the airfoil & wake nodes added 7/26/90              *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      do 15 is = 1, 2
         do 20 ibl = 2, nbl(is)
            i      = ipan(ibl,is)
            iv     = isys(ibl,is)
            uei    = uedg(ibl,is)
            mdi    = mass(ibl,is)
            thi    = thet(ibl,is)
            dsi    = mdi / uei
c
            coeff1 = mdi / uei**2
            coeff2 = dsi / thi**2
c
c  **        compute duedg/dAn at each airfoil node
            do 26 iAn = ifirstAn, ilastAn
               iRHS = 3 + iAn - ifirstAn
c
               sum = 0.0
               do 30 js = 1, 2
                  do 40 jbl = 2, nbl(js)
                     j   = ipan(jbl,js)
                     jv  = isys(jbl,js)
                     sum = sum + vti(ibl,is) * vti(jbl,js)
     &                         * dij(i,j) * vdel(3,iRHS,jv)
 40               continue
 30            continue
c
               ue_an(i,iAn) = qa_an(i,iAn) + sum
               ds_an(i,iAn) = -vdel(3,iRHS,iv) / uei
     &                       -  coeff1 * ue_an(i,iAn)
 26         continue
c
c  **        compute duedg/dBn at each airfoil node
            do 88 iBn = ifirstBn, ilastBn
               iRHS = (3 + numAn) + iBn - ifirstBn
c
               sum = 0.0
               do 90 js = 1, 2
                  do 100 jbl = 2, nbl(js)
                     j   = ipan(jbl,js)
                     jv  = isys(jbl,js)
                     sum = sum + vti(ibl,is) * vti(jbl,js)
     &                         * dij(i,j) * vdel(3,iRHS,jv)
 100              continue
 90            continue
c
```

```
              ue_bn(i,iBn) = qa_bn(i,iBn) + sum
              ds_bn(i,iBn) = -vdel(3,iRHS,iv) / uei
     &                         -  coeff1 * ue_bn(i,iBn)
 88         continue
 20       continue
 15     continue
c
  -     return
        end   !subroutine finddUe




        subroutine FinddCl
c
c    **********************************************************************
c    *  this subroutine computes dCl/dAn, dCl/dBn AND dCm/dAn, dCm/dBn at *
c    *  the airfoil & wake nodes added 7/26/90                            *
c    **********************************************************************
c
        INCLUDE 'xfoil.inc'
c
        real CPG1_an(ifirstAn:maxAn), CPG1_bn(ifirstBn:maxBn)
c
        SA = SIN(ARAD)
        CA = COS(ARAD)
C
        BETA = SQRT(1.0 - MINF**2)
        BFAC = 0.5*MINF**2 / (1.0 + BETA)
C
        I = 1
        CGINC = 1.0 - (GAM(I)/QINF)**2
        denom = beta + Bfac * CGINC
        CPG1  = CGINC / denom
C
        do 126 iAn = ifirstAn, ilastAn
c **      zeroing out sensitivities added 7/26/90
          cl_an(iAn)    = 0.0
          cm_an(iAn)    = 0.0
          CGINC_an      = -2.0 * ue_an(i,iAn) / qinf**2
          CPG1_an(iAn) = CGINC_an * (1.0 - CPG1 * Bfac) / denom
 126    continue
c
        do 188 iBn = ifirstBn, ilastBn
c **      zeroing out sensitivities added 7/26/90
          cl_bn(iBn)    = 0.0
          cm_bn(iBn)    = 0.0
          CGINC_bn      = -2.0 * ue_bn(i,iBn) / qinf**2
          CPG1_bn(iBn) = CGINC_bn * (1.0 - CPG1 * Bfac) / denom
 188    continue
C
```

```
      DO 10 I=1, N
        IP = I+1
        IF(I.EQ.N) IP = 1
C
        CGINC = 1.0 - (GAM(IP)/QINF)**2
        denom = beta + Bfac * CGINC
        CPG2  = CGINC / denom
C
        AX    = (0.5*(X(IP)+X(I)) - 0.25)*CA + 0.5*(Y(IP)+Y(I))*SA
        AY    = (0.5*(Y(IP)+Y(I))       )*CA - 0.5*(X(IP)+X(I))*SA
c
        DX    = (X(IP) - X(I))*CA + (Y(IP) - Y(I))*SA
        DY    = (Y(IP) - Y(I))*CA - (X(IP) - X(I))*SA
C
        AG    = 0.5 * (CPG2 + CPG1)
c
c **    compute An sensitivities
        do 226 iAn = ifirstAn, ilastAn
          CGINC_an    = -2.0 * ue_an(ip,iAn) / qinf**2
          CPG2_an     = CGINC_an * (1.0 - CPG2 * Bfac) / denom
c
          AG_an       = 0.5 * (CPG2_an + CPG1_an(iAn))
          DG_an       =        CPG2_an - CPG1_an(iAn)
c
          x1_an       =  real(za_an(i ,iAn))
          x2_an       =  real(za_an(ip,iAn))
          y1_an       = aimag(za_an(i ,iAn))
          y2_an       = aimag(za_an(ip,iAn))
c
          AX_an       = 0.5 * (x2_an + x1_an) * CA
     &                + 0.5 * (y2_an + y1_an) * SA
          AY_an       = 0.5 * (y2_an + y1_an) * CA
     &                - 0.5 * (x2_an + x1_an) * SA
c
          cl_an(iAn) = cl_an(iAn) + dx* AG_an
          cm_an(iAn) = cm_an(iAn) - dx*(AG_an * ax - DG_an * dx / 12.0)
     &                            - dy*(AG_an * ay - DG_an * dy / 12.0)
     &                            - ag*(AX_an * dx + AY_an * dy       )
c
          CPG1_an(iAn) = CPG2_an
 226    continue
c
c **    compute Bn sensitivities
        do 288 iBn = ifirstBn, ilastBn
          CGINC_bn    = -2.0 * ue_bn(ip,iBn) / qinf**2
          CPG2_bn     = CGINC_bn * (1.0 - CPG2 * Bfac) / denom
c
          AG_bn       = 0.5 * (CPG2_bn + CPG1_bn(iBn))
          DG_bn       =        CPG2_bn - CPG1_bn(iBn)
c
          x1_bn       =  real(za_bn(i ,iBn))
          x2_bn       =  real(za_bn(ip,iBn))
          y1_bn       = aimag(za_bn(i ,iBn))
          y2_bn       = aimag(za_bn(ip,iBn))
```

```
c
          AX_bn       = 0.5 * (x2_bn + x1_bn) * CA
     &                + 0.5 * (y2_bn + y1_bn) * SA
          AY_bn       = 0.5 * (y2_bn + y1_bn) * CA
     &                - 0.5 * (x2_bn + x1_bn) * SA
c
          cl_bn(iBn) = cl_bn(iBn) + dx* AG_bn
          cm_bn(iBn) = cm_bn(iBn) - dx*(AG_bn * ax - DG_bn * dx / 12.0)
     &                            - dy*(AG_bn * ay - DG_bn * dy / 12.0)
     &                            - ag*(AX_bn * dx + AY_bn * dy       )
c
          CPG1_bn(iBn) = CPG2_bn
 288    continue
c
        CPG1 = CPG2
  10 CONTINUE
c
      return
      end   !subroutine finddCl




      subroutine FinddCd
c
c  *********************************************************************
c  *  this subroutine computes dCd/dAn, dCd/dBn for the airfoil        *
c  *  added 7/26/90                                                    *
c  *********************************************************************
c
      INCLUDE 'xfoil.inc'
c
c **  compute dCd/dAn and dCd/dBn, w/o using Squire-Young relation
      do 268 iAn = ifirstAn, ilastAn
         iRHS       = 3 + iAn - ifirstAn
         cd_an(iAn) = -2.0 * vdel(2, iRHS, N+Nw)
 268  continue
c
      do 888 iBn = ifirstBn, ilastBn
         iRHS       = (3 + numAn) + iBn - ifirstBn
         cd_bn(iBn) = -2.0 * vdel(2, iRHS, N+Nw)
 888  continue
c
      return
      end   !subroutine finddCd
      subroutine FinddArea
c
c  ********************************************************************
c  *  This routine computes the airfoil area and area sensitivties     *
c  *  assuming counterclockwise ordering.                              *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      TheArea = 0.0
c
      do 5 iAn = ifirstAn, ilastAn
         area_an(iAn) = 0.0
 5    continue
c
      do 7 iBn = ifirstBn, ilastBn
         area_bn(iBn) = 0.0
 7    continue
c
      do 10 ia = 1, N-1
         rx      = x(ia+1) + x(ia)
         ry      = y(ia+1) + y(ia)
         dx      = x(ia+1) - x(ia)
         dy      = y(ia+1) - y(ia)
         da      = 0.25 * (rx * dy - ry * dx)
         TheArea = TheArea + da
c
c  **    compute sensitivities w.r.t An
         do 26 iAn = ifirstAn, ilastAn
            rx_an = real (za_an(ia+1,iAn)) + real (za_an(ia,iAn))
            ry_an = aimag(za_an(ia+1,iAn)) + aimag(za_an(ia,iAn))
            dx_an = real (za_an(ia+1,iAn)) - real (za_an(ia,iAn))
            dy_an = aimag(za_an(ia+1,iAn)) - aimag(za_an(ia,iAn))
c
            da_an        = 0.25 * (rx_an * dy + rx * dy_an
     &                         -  ry_an * dx - ry * dx_an)
            area_an(iAn) = area_an(iAn) + da_an
 26      continue
c
c  **    compute sensitivities w.r.t Bn
         do 88 iBn = ifirstBn, ilastBn
            rx_bn = real (za_bn(ia+1,iBn)) + real (za_bn(ia,iBn))
            ry_bn = aimag(za_bn(ia+1,iBn)) + aimag(za_bn(ia,iBn))
            dx_bn = real (za_bn(ia+1,iBn)) - real (za_bn(ia,iBn))
            dy_bn = aimag(za_bn(ia+1,iBn)) - aimag(za_bn(ia,iBn))
c
            da_bn        = 0.25 * (rx_bn * dy + rx * dy_bn
     &                         -  ry_bn * dx - ry * dx_bn)
            area_bn(iBn) = area_bn(iBn) + da_bn
 88      continue
   10 continue
C
      return
      end   !subroutine FinddArea
      subroutine dFcost(ipt)
c
c   *********************************************************************
c   *  This routine computes Fcost = Fobj + sum of constraints.  It also *
c   *  computes the gradient of Fcost.                                   *
c   *********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      if (ipt.eq.1) then
c  **    zero global Fcost and grad(Fcost)
         Fcost(2) = 0.0
c
         do 126 iAn = ifirstAn, ilastAn
            Fcost_an(iAn) = 0.0
 126      continue
c
         do 188 iBn = ifirstBn, ilastBn
            Fcost_bn(iBn) = 0.0
 188     continue
      endif
c
c
c  ** compute global Fcost based on fcost at each design point
      fipt     = Fobj  + 0.5 * kost * SumConst(1, 0)
      Fcost(2) = Fcost(2) + Weight(ipt) * fipt
c
c
c  ** compute gradient of global Penalty function (Fcost)
      do 26 iAn = ifirstAn, ilastAn
c  **    determine dFobj/dAn
         fipt_an       = Fobj_An(iAn)  + kost * SumConst(2, iAn)
         Fcost_an(iAn) = Fcost_an(iAn) + weight(ipt) * fipt_an
 26   continue
c
      do 88 iBn = ifirstBn, ilastBn
c  **    determine dFobj/dBn
         fipt_bn       = Fobj_Bn(iBn)  + kost * SumConst(3, iBn)
         Fcost_bn(iBn) = Fcost_bn(iBn) + weight(ipt) * fipt_bn
 88   continue
c
      return
      end   !subroutine dFcost
      function SumConst (i, m)
c
c  ***************************************************************************
c  *  This function:                                                         *
c  *     1.)  Sums constraints        if i = 1                               *
c  *     2.)  Sums d(constraints)/dAn if i = 2                               *
c  *     3.)  Sums d(constraints)/dBn if i = 3                               *
c  ***************************************************************************
c
      include 'xfoil.inc'
c
      SumConst = 0.0
c
      if     (i.eq.1) then
         do 10 j = 1, Nconst
            SumConst = SumConst + OnOff(j) * Const(j)**2
 10      continue
c
      elseif (i.eq.2) then
         do 20 j = 1, Nconst
            SumConst = SumConst + OnOff(j) * Const(j) * Const_An(j,m)
 20      continue
c
      elseif (i.eq.3) then
         do 30 j = 1, Nconst
            SumConst = SumConst + OnOff(j) * Const(j) * Const_Bn(j,m)
 30      continue
      endif
c
      return
      end     !function SumConst




      subroutine Search
c
c  ************************************************************************
c  *  This subroutine picks a search direction (SdirAn,Bn).               *
c  ************************************************************************
c
      INCLUDE 'xfoil.inc'
c
      TheLength = 0.0
c
c  ** Steepest Descent Methods (Sdir = -grad(Fcost))
      do 268 iAn = ifirstAn, ilastAn
         SdirAn(iAn) = -Fcost_An(iAn)
         TheLength   = TheLength + SdirAn(iAn)**2
 268  continue
c
```

```
      do 288 iBn = ifirstBn, ilastBn
         SdirBn(iBn) = -Fcost_Bn(iBn)
         TheLength   = TheLength + SdirBn(iBn)**2
 288  continue
c
      TheLength = sqrt(TheLength)
c
      return
      end   !subroutine Search




      function dot(An1, Bn1, An2, Bn2)
c
c  ***********************************************************************
c  *  This function computes the dot product of two vectors in which the  *
c  *  the components are grouped in terms of An's and Bn's.               *
c  ***********************************************************************
c
      INCLUDE 'xfoil.inc'
      real    An1(ifirstAn:maxAn), Bn1(ifirstBn:maxBn)
      real    An2(ifirstAn:maxAn), Bn2(ifirstBn:maxBn)
c
      dot = 0.0
c
      do 26 iAn = ifirstAn, ilastAn
         dot = dot + An1(iAn) * An2(iAn)
 26   continue
c
      do 88 iBn = ifirstBn, ilastBn
         dot = dot + Bn1(iBn) * Bn2(iBn)
 88   continue
c
      return
      end   !function dot
      subroutine NextPt
c
c  *********************************************************************
c  *  This routine computes the next optimization point based on the old *
c  *  current optimization point, epsilon, and the search direction.    *
c  *********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      real    dF_dE(0:2)
      logical EndIt
c
c  ** steepest descent with epsilon constant
      call PickEpsilon              ! prevent epsilon from being too big
      call NewCn                    ! compute new An's and Bn's
      call coords                   ! compute new geometry
c
      do 5 ipt = 1, NumPts
         call guessBL (ipt)         ! update BL variables
         call Analiz  (ipt)         ! compute viscous solution
 5    continue
c
      return
      end   !subroutine NextPt
```

```
      subroutine PickEpsilon
c
c  ************************************************************************
c  *  No matter what epsilon should be used, prevent it from being too large.*
c  ************************************************************************
c
      INCLUDE 'xfoil.inc'
c
      epsilon(1) = epsilon(2)
c
      do 26 iAn = ifirstAn, ilastAn
         delta      =   epsilon(1) *   SdirAn(iAn)    ! delta w/ const. epsilon
         deltaMax   =   PercentMax *  real(Cn(iAn))   ! max. allowed delta
         epsilonMax = abs(deltaMax /   SdirAn(iAn))   ! epsilon for max delta
c
         if (abs(delta).gt.abs(deltaMax)) then
            epsilon(1) = amin1(epsilonMax, epsilon(1))
         endif
 26   continue
c
      return
      end   !subroutine PickEpsilon
      subroutine EndCheck(iter)
c
c  ********************************************************************
c  *  This routine determines if an ending condition has been reached.   *
c  *  Routine liberated from (with small modifications):                  *
c  *  'Numerical Optimization Techniques for Engineering Design: with     *
c  *  Applications', Garret N. Vanderplaats, McGraw-Hill Book Company,    *
c  *  page 102.                                                           *
c  ********************************************************************
c
      parameter   (RelMin = 1.0e-3, Nstar = 3)
c
      INCLUDE     'xfoil.inc'
c
      if (iter.eq.0) then
c  **    cannot check changes before 1st iteration
c  **    compute absolute min. limit only
         AbsMin   = 0.001 * Fcost(2)
c
      else
c  **    CONDITION # 1: stop if number of iterations exceed a maximum
         if (iter.ge.MaxIt) then
            LTheEnd = .true.
            write (6,*)
     &      'OPTZ: stopped due to excessive number of iterations.'
            return
         endif
c
c
c  **    CONDITION # 2: stop if absolute change in Fcost <= AbsMin
c  **    must occur Nstar times in a row
         dFabs = abs(Fcost(2) - Fcost(1))
c
         if (dFabs.gt.AbsMin) then
            jEnd(2) = 0
         else
            jEnd(2) = jEnd(2) + 1
         endif
c
         if (jEnd(2).ge.Nstar) then
            LTheEnd = .true.
            write (6,26)
     &      'OPTZ: stopped due to absolute change in Fcost <= ', AbsMin
            return
         endif
c
c
c  **    CONDITION # 3: stop if relative change in Fcost <= RelMin
c  **    must occur Nstar times in a row
         dFrel = dFabs / amax1(abs(Fcost(2)),1.0e-10)
c
```

```
         if (dFrel.gt.RelMin) then
            jEnd(3) = 0
         else
            jEnd(3) = jEnd(3) + 1
         endif
c
         if (jEnd(3).ge.Nstar) then
            LTheEnd = .true.
            write (6,26)
     &      'OPTZ: stopped due to relative change in Fcost <= ', RelMin
            return
         endif
c
c
c  **    CONDITION # 4: stop if any design point viscous solution unconverged
         if (.not.Lconv) then
            LTheEnd = .true.
            write (6,*)
     &      'OPTZ: stopped due to non-convergence of viscous solution.'
            return
         endif
      endif
c
c  ** Assign previous Fcost values and reinitialize Lconv
      Fcost(1) = Fcost(2)
c
 26   format (a, f9.7)
      return
      end   !subroutine EndCheck





      subroutine NewCn
c
c  *******************************************************************
c  *  This routine computes the Cn's at the new optimization point.    *
c  *******************************************************************
c
      INCLUDE 'xfoil.inc'
c
      if (Lthick.or.Lcamber) then
c  **    compute changes needed in An's and add to Cn's
         do 126 iAn = ifirstAn, ilastAn
            deltaAn(iAn)   =  epsilon(1) * SdirAn(iAn)
            Cn(iAn)        =  Cn(iAn) + cmplx(deltaAn(iAn), 0.0)
 126     continue
c
c  **    compute changes needed in Bn's and add to Cn's
         do 188 iBn = ifirstBn, ilastBn
            deltaBn(iBn)   =  epsilon(1) * SdirBn(iBn)
            Cn(iBn)        =  Cn(iBn) + cmplx(0.0, deltaBn(iBn))
 188     continue
      endif
c
      return
      end   !subroutine NewCn
```

```
      subroutine coords
c
c  ***********************************************************************
c  *  This routine computes the coordinates of an airfoil geometry given  *
c  *  the Cn distribution.                                               *
c  ***********************************************************************
c
      INCLUDE 'xfoil.inc'
c
c  ** calculate new airfoil geometry
      call mapgen
C
c  ** convert new geometry to real coords and store in buffer
      NB = NC
      DO 95 I=1, NB
        XB(I) =  REAL(ZC(I))
        YB(I) = AIMAG(ZC(I))
   95 CONTINUE
C
C---- spline new buffer airfoil
      CALL SCALC (XB,YB, SB,NB)
      CALL SPLINE(XB,XBP,SB,NB)
      CALL SPLINE(YB,YBP,SB,NB)
c
c  ** repanel airfoil using old normalized arc length
      do 26 ia = 1, N
         x(ia) = seval(snorm(ia) * sb(NB), xb, xbp, sb, NB)
         y(ia) = seval(snorm(ia) * sb(NB), yb, ybp, sb, NB)
 26   continue
c
c  ** spline new current airfoil
      CALL SCALC(X,Y,S,N)
      CALL SPLINE(X,XP,S,N)
      CALL SPLINE(Y,YP,S,N)
      CALL LEFIND(SLE,X,XP,Y,YP,S,N)
C
C---- set various flags for new airfoil
      LGAMU  = .FALSE.
      LQINU  = .FALSE.
      LWAKE  = .FALSE.
      LQAIJ  = .FALSE.
      LADIJ  = .FALSE.
      LWDIJ  = .FALSE.
      LSCINI = .FALSE.
      LQSPEC = .FALSE.
      LIPAN  = .true.
      LBLINI = .true.
      LVCONV = .true.
C
C---- calculate panel angles for panel routines
      CALL APCALC
C
      CALL NCALC(X,Y,S,N,NX,NY)
C
      return
      end   !subroutine coords







      subroutine guessBL(ipt)
c
c   ************************************************************************
c   *  This routine approximates the new BL variables values with a linear *
c   *  approximation using the BL variable sensitivities.                  *
c   ************************************************************************
c
      INCLUDE 'xfoil.inc'
      INCLUDE 'xbl.inc'
c
      call dumpIN (ipt)
c
      DO 5 IS=1, 2
        DO 50 IBL=2, NBL(IS)
          I  = IPAN(IBL,IS)
          IV = ISYS(IBL,IS)
C
c   **    calculate change in BL variables
          dctau = 0.0
          dthet = 0.0
          dmass = 0.0
          duedg = 0.0
          ddstr = 0.0
c
c   **    add changes due to deltaAn
          do 26 iAn = ifirstAn, ilastAn
             iRHS   = 3 + iAn - ifirstAn
             dctau = dctau - deltaAn(iAn) * vdel(1,iRHS,iv)
             dthet = dthet - deltaAn(iAn) * vdel(2,iRHS,iv)
             dmass = dmass - deltaAn(iAn) * vdel(3,iRHS,iv)
             duedg = duedg + deltaAn(iAn) * ue_an(i,iAn)
             ddstr = ddstr + deltaAn(iAn) * ds_an(i,iAn)
```

```
 26       continue
c
c  **     add changes due to deltaBn
          do 88 iBn = ifirstBn, ilastBn
             iRHS   = (3 + numAn) + iBn - ifirstBn
             dctau = dctau - deltaBn(iBn) * vdel(1,iRHS,iv)
             dthet = dthet - deltaBn(iBn) * vdel(2,iRHS,iv)
             dmass = dmass - deltaBn(iBn) * vdel(3,iRHS,iv)
             duedg = duedg + deltaBn(iBn) * ue_bn(i,iBn)
             ddstr = ddstr + deltaBn(iBn) * ds_bn(i,iBn)
 88       continue
c
c
c  **     calculate movement of transition point
          if (ibl.eq.(itran(is)-1)) then
             dxtr   = xtr_a1(is) * dctau + xtr_t1(is) * dthet
    &                + xtr_d1(is) * ddstr + xtr_u1(is) * duedg
c
             xtrNEW = xtr(is) + dxtr
             ijump  = 0    ! number of panels xtran has shifted
c
c  **        determine which BL panel new trans. pt. lies on
             if (xtrNEW.ge.xtr(is)) then
c  **           search in downstream direction
 100            iNew = ipan(ibl + ijump + 1,is)
c
                if (xtrNEW.gt.x(iNew)) then
                   ijump = ijump + 1
                   go to 100
                endif
c
                ilow  = ibl
                ihigh = ibl + ijump
             else
c  **           search in upstream direction
 200            iNew = ipan(ibl - ijump - 1,is)
c
                if (xtrNEW.lt.x(iNew)) then
                   ijump = ijump - 1
                   go to 200
                endif
c
                ilow  = ibl - ijump
                ihigh = ibl
             endif
          endif
c
c
c  **     compute estimates of new BL variables
          if ((ibl.gt.ilow).and.(ibl.le.ihigh)) then
c  **        extrapolate over transition point movement area
c  **        linear extrapolation for CTAU, THET, and UEDG
             CTAU(IBL,IS) = 2.0 * CTAU(IBL-1,IS) - CTAU(IBL-2,IS)
             THET(IBL,IS) = 2.0 * THET(IBL-1,IS) - THET(IBL-2,IS)
```

```
            UEDG(IBL,IS) = 2.0 * UEDG(IBL-1,IS) - UEDG(IBL-2,IS)
c
c  **       straight across approximation for MASS and DSTR
            mass(ibl,is) = mass(ibl-1,is)
            DSTR(IBL,IS) = DSTR(IBL-1,IS)
         else
c  **       no change in panel of transition pt.
            CTAU(IBL,IS) = CTAU(IBL,IS) + DCTAU
            THET(IBL,IS) = THET(IBL,IS) + DTHET
            UEDG(IBL,IS) = UEDG(IBL,IS) + DUEDG
            mass(ibl,is) = mass(ibl,is) + dmass
            DSTR(IBL,IS) = DSTR(IBL,IS) + DDSTR
         endif
   50   CONTINUE
    5 CONTINUE
c
      return
      end   !subroutine guessBL




      subroutine dumpIN(ipt)
c
c  *******************************************************************
c  *  This routine reads in all variables from the scrathc file that  *
c  *  may haved changed between design points.                        *
c  *******************************************************************
c
      INCLUDE 'xfoil.inc'
      INCLUDE 'xbl.inc'
c
      num = NumDump + ipt
      rewind (unit = num)
c
c  ** read in variables for OUTDATA
      read (num,*)   adeg, Cl, Cm, Cd, xtr(1), xtr(2), IterConv
c
c  ** read in single variables
      read (num,*)  SST, SST_GO,SST_GP,ALFGAM,CLGAM,CMGAM,AGO,IST,
     &              SSPLE,SMAX,IQ1,IQ2,ALFQSP,CLQSP,CMQSP,CIRC,
     &              ARAD,AWAKE,AVISC,MINF,TKLAM,CPSTAR,QSTAR,WAKLEN,
     &              GAMTE,SIGTE,DXTE,DYTE,DSTE,ANTE,ASTE,
     &              CVPAR,CTERAT,CTRRAT,XSREF1,XSREF2,XPREF1,XPREF2,
     &              RETYP,REINF, Cn(0), Cn(1), nbl(1), nbl(2), clspec
c
c  ** read in BL variables
      do 10 is = 1, 2
         read (num,*)  XTR1(is),  XTR(is), XSSITR(is),
     &                 IBLTE(is), NBL(is), ITRAN(is),TFORCE(is),
     &                 xtr_a1(is), xtr_t1(is), xtr_d1(is),
     &                xtr_u1(is), xtr_x1(is)
c
         do 20 ibl = 2, nbl(is)
            read (num,*)  XSSI(ibl,is), CTAU(ibl,is), UEDG  (ibl,is),
     &                    THET(ibl,is), MASS(ibl,is), DSTR  (ibl,is),
     &                    TAU (ibl,is), UINV(ibl,is), UINV_A(ibl,is),
     &                    IPAN(ibl,is), ISYS(ibl,is), VTI   (ibl,is)
c
            iv = isys(ibl,is)
            do 30 iRHS = 1, numRHS
               read (num,*)  vdel(1,iRHS,iv), vdel(2,iRHS,iv),
     &                       vdel(3,iRHS,iv)
 30         continue
 20      continue
 10   continue
c
c  ** read in sensitivity stuf
      do 40 ia = 1, N+Nw
         do 50 iAn = ifirstAn, ilastAn
            read (num,*)  ue_an(ia,iAn), ds_an(ia,iAn)
 50      continue
c
         do 60 iBn = ifirstBn, ilastBn
            read (num,*)  ue_bn(ia,iBn), ds_bn(ia,iBn)
 60      continue
 40   continue
c
      return
      end   !subroutine dumpIN




      SUBROUTINE ZETAF(Z0, ZETO, ia)
c
C-----------------------------------------------------
C     Airfoil-to-circle mapping function.
C
C       Z0      airfoil plane point
C       ZETO    circle plane point
C
C      ZETO is assumed to initially contain the
C      circle-plane position of a neighboring point.
C      This is used as an initial guess for efficient
C      iteration to the new point.
C
C      If ZETO is passed in as (0,0), then a fresh
C      guess is made here.
C-----------------------------------------------------
c
      INCLUDE 'xfoil.inc'
```

```
c
      COMPLEX ZO, ZETO, zetinv, ZO_ZETO, DZETO, ZOITER, ZPLATE
C
C---- make initial guess for Zeta if passed in as (0,0)
      ZETABS = REAL(ZETO)**2 + AIMAG(ZETO)**2
c
      IF(ZETABS .EQ. 0.0) THEN
C-----   find position relative to a plate of chord 4 replacing the airfoil
         ZPLATE = 4.0*(ZO-ZLE)/CHORDZ - 2.0
C
C-----   set initial guess with inverse Joukowsky transform
         ZETO = 0.5*( ZPLATE + SIGN(1.0,REAL(ZPLATE))
     &                       * SQRT(ZPLATE**2 - 4.0))
      ENDIF
C
C---- solve  z(ZETO) - ZO = 0  by Newton-Raphson
      DO 10 ITER=1, 200
         zetinv = 1.0 / zeto
c
         CALL  ZF(ZETO, ZOITER, ia)
         CALL ZFD(ZETinv, ZO_ZETO)
C
         DZETO  = (ZO-ZOITER)/ZO_ZETO
         ZETO   = ZETO + DZETO
C
         IF(CABS(DZETO) .LE. 1.0E-5) RETURN
   10 CONTINUE
C
      WRITE(6,*) 'ZETAF: Convergence failed. Continuing ...'
      RETURN
      END




      SUBROUTINE ZF(ZETO, ZO, ia)
c
C-------------------------------------------------------
C     Circle-to-airfoil mapping function.
C
C       ZETO    circle plane point
C       ZO      airfoil plane point
C
C     The airfoil plane point ZO is calculated by
C     integrating dz/dZeta radially out in the circle
C     plane, starting on the circle where z(Zeta)
C     is already known.
C-------------------------------------------------------
c
      INCLUDE 'xfoil.inc'
c
```

```
      COMPLEX ZETO, ZO, ZLOG, ZETINV, ZET1, ZET2, Z_ZETA
C
C---- angle and log(r) of point in circle plane
      ZLOG = CLOG(ZETO)
      WCZ = AIMAG(ZLOG)
      GCZ =  REAL(ZLOG)
C
C---- make sure angle is between 0 and 2pi
      IF(WCZ.LT.0.0) WCZ = WCZ + 2.0 * 3.14159265
C
C---- set z on airfoil at the point with same circle plane angle as ZETO
      ICO = INT(WCZ/DWC) + 1
      WCO = DWC*FLOAT(ICO-1)
      IF(ICO.GE.NC) ICO = 1
      ZO = ZC(ICO) + (ZC(ICO+1)-ZC(ICO))*(WCZ-WCO)/DWC
C
C---- number of radial integration intervals uniform in log(r)
C     (i.e. exponential stretching in Zeta is used)
      NR = IABS(INT(GCZ/0.05)) + 1
C
C---- log(r) increment for radial integration
      DG = GCZ/FLOAT(NR)
C
      RRAT = EXP(DG)
C
C---- set Zeta on circle contour (airfoil surface) and first point off
      ZET1 = CMPLX( COS(WCZ) , SIN(WCZ) )
      ZET2 = ZET1*RRAT
C
C---- integrate radially out
      DO 100 IR=1, NR
        ZETINV = 2.0/(ZET1+ZET2)
        call zfd(zetinv, z_zeta)
c
C------ perform integration
        ZO = ZO + Z_ZETA*(ZET2-ZET1)
C
C------ increment zeta
        ZET1 = ZET2
        ZET2 = ZET1*RRAT
  100 CONTINUE
C
      RETURN
      END
      SUBROUTINE ZFD(ZETINV, ZO_ZETO)
c
C-------------------------------------------------------
C     Circle-to-airfoil mapping derivative function.
C
C       ZETINV  1.0/circle plane point
C       ZO_ZETO mapping derivative
C-------------------------------------------------------
c
      INCLUDE 'xfoil.inc'
c
      COMPLEX CNSUM, ZETINV, ZO_ZETO
C
      CNSUM = CN(MC)
      DO 10 M=MC-1, 0, -1
        CNSUM = CN(M) + CNSUM*ZETINV
   10 CONTINUE
C
      ZO_ZETO = CEXP(CNSUM) * (1.0 - ZETINV)**(1.0-AGTE)
C
      RETURN
      END
```

D.1.3 User Modifiable Routines

Certain routines require that the user modify the code. These subroutines are contained in the file 'xuser.f' and are listed below:

```
c
c  ***********************************************************************
c  *  The following sensitivities are calculated and stored.  They can   *
c  *  used as building blocks:                                           *
c  *                                                                     *
c  *  Quantity                          Variable Names in XFOIL          *
c  *  ------------------------------    -----------------------          *
c  *  Lift   Coefficient                cl_an, cl_bn                     *
c  *  Moment Coefficient                cm_an, cm_bn                     *
c  *  Drag   Coefficient                cd_an, cd_bn                     *
c  *  Airfoil geometry                  za_an, za_bn                     *
c  *  Inviscid velocity on airfoil      qa_an, qa_bn                     *
c  *  Edge Velocity                     ue_an, ue_bn                     *
c  *  Displacement Thickness            ds_an, ds_bn                     *
c  *  Friction Coefficient              -vdel(1,iRHS,iv)                 *
c  *  Momentum Thickness                -vdel(2,iRHS,iv)                 *
c  *  Mass Defect                       -vdel(3,iRHS,iv)                 *
c  *  Airfoil Area                      area_an, area_bn                 *
c  *                                                                     *
c  ***********************************************************************
c
```

```
      subroutine constrain
c
c  ********************************************************************
c  *  This routine calculates sensitivities of the constraints         *
c  *                                                                   *
c  *  USER MODIFIABLE ROUTINE                                          *
c  ********************************************************************
c
      INCLUDE 'xfoil.inc'
c
      Nconst = 0    ! Total number of constraints
      Iconst = 0    ! Current constraint index
c
c
c
c  ** CONSTRAINT # 1: minimum thickness at x/c = 0.95 (inequality)
      tcmin  = 0.01                  ! minimum thickness
      Nconst = Nconst + 1
      Iconst = Iconst + 1
c
c  ** the following is a crude algorithm to find the
c  ** nodes surrounding x/c = 0.95
      xconst = 0.95
c
      do 10 ia = 2, N
         if     ((x(ia).lt.xconst).and.(x(ia-1).ge.xconst)) then
            iaa = ia-1
            iab = ia
         elseif ((x(ia).gt.xconst).and.(x(ia-1).le.xconst)) then
            iad = ia-1
            iae = ia
         endif
 10   continue
c
      coeff1 = (xconst - x(iaa)) / (x(iab) - x(iaa))
      coeff2 = (xconst - x(iae)) / (x(iad) - x(iae))
c
c  ** compute thickness at xconst
      yu = coeff1 * (y(iab) - y(iaa)) + y(iaa)
      yl = coeff2 * (y(iad) - y(iae)) + y(iae)
      tc = yu - yl
      print*, 'tmin = ', tc
c
c  ** define constraint
      Const(Iconst) = tc - tcmin
c
c  ** compute constraint gradient
      do 26 iAn = ifirstAn, ilastAn
         yu_an = coeff1 *
     &           (aimag(za_an(iab,iAn)) - aimag(za_an(iaa,iAn)))
     &          +  aimag(za_an(iaa,iAn))
c
         yl_an = coeff2 *
     &           (aimag(za_an(iad,iAn)) - aimag(za_an(iae,iAn)))
     &         +  aimag(za_an(iae,iAn))
c
         Const_An(Iconst,iAn) = yu_an - yl_an
 26   continue
c
      do 88 iBn = ifirstBn, ilastBn
         yu_bn = coeff1 *
     &           (aimag(za_bn(iab,iBn)) - aimag(za_bn(iaa,iBn)))
     &         +  aimag(za_bn(iaa,iBn))
c
         yl_bn = coeff2 *
     &           (aimag(za_bn(iad,iBn)) - aimag(za_bn(iae,iBn)))
     &         +  aimag(za_bn(iae,iBn))
c
         Const_Bn(Iconst,iBn) = yu_bn - yl_bn
 88   continue
c
c  ** is Constraint active?
      if (const(Iconst).lt.0.0) then
         print*, '******* CONSTRAINT VIOLATED ******'
         OnOff(Iconst) = 1.0   ! 1.0 => active constraint
      else
         OnOff(Iconst) = 0.0   ! 0.0 => inactive constraint
      endif
c
c
c
c
c  ** CONSTRAINT # 2: constrained area (equality)
      Aconst = 0.1                     ! allowed area
      Nconst = Nconst + 1
      Iconst = Iconst + 1
c
c  ** define constraint
      Const(Iconst) = TheArea - Aconst
c
c  ** compute constraint gradient
      do 126 iAn = ifirstAn, ilastAn
         Const_an(Iconst,iAn) = Area_an(iAn)
 126  continue
c
      do 188 iBn = ifirstBn, ilastBn
         Const_bn(Iconst,iBn) = Area_bn(iBn)
 188  continue
c
c  ** is Constraint active?
      OnOff(Iconst) = 1.0         ! equality constraints always active
      print*, 'AREA = ', TheArea
c
c
c **  CONSTRAINT # 3:
c
      return
      end   !subroutine constrain
      subroutine dFobj
c
c  *******************************************************************
c  *  This routine fills the Fobj (for local design point)           *
c  *  array with the objective function                              *
c  *                                                                 *
c  *  USER MODIFIABLE ROUTINE                                        *
c  *******************************************************************
c
      include 'xfoil.inc'
c
c  ** Assign Fobj the function to be optimized
      if (OptzName.eq.'Cl')    Fobj = -cl
      if (OptzName.eq.'Cd')    Fobj =  cd
      if (OptzName.eq.'Cl/Cd') Fobj = -(cl/cd)
      if (OptzName.eq.'other') Fobj =  cd                  ! User supplied
c
c  ** compute gradient of Fobj
      do 26 iAn = ifirstAn, ilastAn
         if (OptzName.eq.'Cl')    Fobj_An(iAn) = -cl_an(iAn)
         if (OptzName.eq.'Cd')    Fobj_An(iAn) =  cd_an(iAn)
         if (OptzName.eq.'Cl/Cd')
     &       Fobj_An(iAn) = -(Cl/Cd) * (cl_an(iAn)/Cl - cd_an(iAn)/Cd)
         if (OptzName.eq.'other') Fobj_An(iAn) =  cd_an(iAn) ! User supplied
 26   continue
c
      do 88 iBn = ifirstBn, ilastBn
         if (OptzName.eq.'Cl')    Fobj_Bn(iBn) = -cl_bn(iBn)
         if (OptzName.eq.'Cd')    Fobj_Bn(iBn) =  cd_bn(iBn)
         if (OptzName.eq.'Cl/Cd')
     &       Fobj_Bn(iBn) = -(Cl/Cd) * (cl_bn(iBn)/Cl - cd_bn(iBn)/Cd)
         if (OptzName.eq.'other') Fobj_Bn(iBn) =  cd_bn(iBn) ! User supplied
 88   continue
c
      if (OptzName.eq.'other') OptzName = 'Cd'               ! User supplied
      return
      end      !subroutine dFobj
```

D.2 Modified XFOIL Code

The following routines from XFOIL's file 'xbl.f' have minor additions marked by '##':

```
C
C     **********************************************
C     *             Copyright  1989                *
C     *   Massachusetts Institute of Technology    *
C     *            All Rights Reserved             *
C     *                                            *
C     *  Commercial use and/or reproduction without *
```

```
C     *  license prohibited. Licensing agent:       *
C     *                                            *
C     *       MIT Technology Licensing Office      *
C     *               (617) 253-6966               *
C     *                                            *
C     *  Academic and research use unrestricted by *
C     *  verbal permission from:                   *
C     *                                            *
C     *  Mark Drela   (617) 253-0067               *
C     *                                            *
C     **********************************************
C
c ##  NOTE: modifications for optimization routine are marked with: ##
c
      SUBROUTINE SETBL
C...........................................
C     Sets up the BL Newton system coefficients
C     for the current BL variables and the edge
C     velocities received from SETUP. The local
C     BL system coefficients are then
C     incorporated into the global Newton system.
C...........................................
      INCLUDE 'xfoil.inc'
      INCLUDE 'xbl.inc'
      REAL USAV(IVX,2)
      REAL U1_M(2*IVX), U2_M(2*IVX)
      REAL D1_M(2*IVX), D2_M(2*IVX)
      REAL ULE1_M(2*IVX), ULE2_M(2*IVX)
      REAL UTE1_M(2*IVX), UTE2_M(2*IVX)
C
C---- set gas constant (= Cp/Cv)
      GAMBL = GAMMA
      GM1BL = GAMM1
C
C---- set parameters for compressibility correction
      QINFBL = QINF
      TKBL = TKLAM
C
C---- stagnation density and 1/enthalpy
      RSTBL = (1.0 + 0.5*GM1BL*MINF**2) ** (1.0/GAMBL)
      HSTINV = GM1BL*(MINF/QINFBL)**2 / (1.0 + 0.5*GM1BL*MINF**2)
C
C---- Sutherland's const./To   (assumes stagnation conditions are at STP)
      HVRAT = 0.37
C
C---- set Reynolds number based on freestream density, velocity, viscosity
      HERAT = 1.0 - 0.5*QINFBL**2*HSTINV
      REY = REINF * SQRT((HERAT)**3) * (1.0+HVRAT)/(HERAT+HVRAT)
C
C---- set the CL used to define Reynolds number
      IF(LALFA) THEN
       CLREY = CL
      ELSE
       CLREY = CLSPEC
      ENDIF
C
      IF(LALFA .AND. RETYP.GT.1 .AND. CLREY.LT.0.05)
     & WRITE(6,*) 'SETBL warning:  CL < 0.05 ... Re(CL) may blow up.'
C
C---- set actual Reynolds number based on CL
      IF(RETYP.EQ.1) REYBL = REY
      IF(RETYP.EQ.2) REYBL = REY/SQRT(ABS(CLREY))
      IF(RETYP.EQ.3) REYBL = REY/ABS(CLREY)
C
      AMCRIT = ACRIT
C
C---- save TE thickness
      DWTE = WGAP(1)
C
      IF(.NOT.LBLINI) THEN
C----- initialize BL by marching with Ue (fudge at separation)
       WRITE(6,*) ' '
       WRITE(6,*) 'Initializing BL ...'
       CALL MRCHUE
       LBLINI = .TRUE.
      ENDIF
C
      WRITE(6,*) ' '
C
C---- march BL with current Ue and Ds to establish transition
      CALL MRCHDU
C
      DO 5 IS=1, 2
        DO 6 IBL=2, NBL(IS)
          USAV(IBL,IS) = UEDG(IBL,IS)
    6   CONTINUE
    5 CONTINUE
C
      CALL UESET
C
      DO 7 IS=1, 2
        DO 8 IBL=2, NBL(IS)
          TEMP = USAV(IBL,IS)
          USAV(IBL,IS) = UEDG(IBL,IS)
          UEDG(IBL,IS) = TEMP
          DUE = UEDG(IBL,IS) - USAV(IBL,IS)
    8   CONTINUE
    7 CONTINUE
C
      ILE1 = IPAN(2,1)
      ILE2 = IPAN(2,2)
      ITE1 = IPAN(IBLTE(1),1)
      ITE2 = IPAN(IBLTE(2),2)
C
      JVTE1 = ISYS(IBLTE(1),1)
      JVTE2 = ISYS(IBLTE(2),2)
C
      DULE1 = UEDG(2,1) - USAV(2,1)
      DULE2 = UEDG(2,2) - USAV(2,2)
C
C---- set LE and TE Ue sensitivities wrt all m values
      DO 10 JS=1, 2
        DO 110 JBL=2, NBL(JS)
          J  = IPAN(JBL,JS)
          JV = ISYS(JBL,JS)
          ULE1_M(JV) = -VTI(       2,1)*VTI(JBL,JS)*DIJ(ILE1,J)
          ULE2_M(JV) = -VTI(       2,2)*VTI(JBL,JS)*DIJ(ILE2,J)
          UTE1_M(JV) = -VTI(IBLTE(1),1)*VTI(JBL,JS)*DIJ(ITE1,J)
          UTE2_M(JV) = -VTI(IBLTE(2),2)*VTI(JBL,JS)*DIJ(ITE2,J)
  110   CONTINUE
   10 CONTINUE
C
      ULE1_A = UINV_A(2,1)
      ULE2_A = UINV_A(2,2)
C
C**** Go over each boundary layer/wake
      DO 2000 IS=1, 2
C
C---- there is no station "1" at similarity, so zero everything out
      DO 20 JS=1, 2
        DO 210 JBL=2, NBL(JS)
          JV = ISYS(JBL,JS)
          U1_M(JV) = 0.
          D1_M(JV) = 0.
  210   CONTINUE
   20 CONTINUE
      U1_A = 0.
      D1_A = 0.
C
      DUE1 = 0.
      DDS1 = 0.
C
C---- similarity station pressure gradient parameter  x/u du/dx
      IBL = 2
      BULE = 1.0
C
C---- set forced transition arc length position
      CALL XIFSET(IS)
C
      TRAN = .FALSE.
      TURB = .FALSE.
C
C**** Sweep downstream setting up BL equation linearizations
      DO 1000 IBL=2, NBL(IS)
C
      IV  = ISYS(IBL,IS)
C
      SIMI = IBL.EQ.2
      WAKE = IBL.GT.IBLTE(IS)
C
      I = IPAN(IBL,IS)
C
```

```
C---- set primary variables for current station
      XSI = XSSI(IBL,IS)
      AMI = CTAU(IBL,IS)        !  used if  IBL .LT. ITRAN(IS)
      CTI = CTAU(IBL,IS)        !  used if  IBL .GE. ITRAN(IS)
      UEI = UEDG(IBL,IS)
      THI = THET(IBL,IS)
      MDI = MASS(IBL,IS)
C
      DSI = MDI/UEI
c
c  ## Modifications made for Optimization study, April 6, 1990
      i1  = ipan(ibl-1,is)
      if (.not.simi) term1 = mass(ibl-1,is) / uedg(ibl-1,is)**2
                     term2 = mass(ibl  ,is) / uedg(ibl  ,is)**2
C
      IF(WAKE) THEN
       IW = IBL - IBLTE(IS)
       DSWAKI = WGAP(IW)
      ELSE
       DSWAKI = 0.
      ENDIF
C
      D2_M2 =  1.0/UEI
      D2_U2 = -DSI/UEI
C
      DO 30 JS=1, 2
        DO 310 JBL=2, NBL(JS)
          J  = IPAN(JBL,JS)
          JV = ISYS(JBL,JS)
          U2_M(JV) = -VTI(IBL,IS)*VTI(JBL,JS)*DIJ(I,J)
          D2_M(JV) = D2_U2*U2_M(JV)
  310   CONTINUE
   30 CONTINUE
      D2_M(IV) = D2_M(IV) + D2_M2
C
      U2_A = UINV_A(IBL,IS)
      D2_A = D2_U2*U2_A
C
C---- "forced" changes due to mismatch between UEDG and USAV=UINV+dij*MASS
      DUE2 = UEDG(IBL,IS) - USAV(IBL,IS)
      DDS2 = D2_U2*DUE2
C
C---- check for transition and set TRAN, XT, etc. if found
      IF((.NOT.SIMI) .AND. (.NOT.TURB)) CALL TRCHEK(IBL,IS)
C
C---- assemble 10x4 linearized system for dCtau, dTh, dDs, dUe, dXi
C     at the previous "1" station and the current "2" station
C
      IF(IBL.EQ.IBLTE(IS)+1) THEN
C
C----- define quantities at start of wake, adding TE base thickness to Dstar
       TTE = THET(IBLTE(1),1) + THET(IBLTE(2),2)
       DTE = DSTR(IBLTE(1),1) + DSTR(IBLTE(2),2) + ANTE
       CTE = ( CTAU(IBLTE(1),1)*THET(IBLTE(1),1)
     &       + CTAU(IBLTE(2),2)*THET(IBLTE(2),2) ) / TTE
       CALL TESYS(CTE,TTE,DTE)
C
       TTE_TTE1 = 1.0
       TTE_TTE2 = 1.0
       DTE_MTE1 =               1.0 / UEDG(IBLTE(1),1)
       DTE_UTE1 = -DSTR(IBLTE(1),1) / UEDG(IBLTE(1),1)
       DTE_MTE2 =               1.0 / UEDG(IBLTE(2),2)
       DTE_UTE2 = -DSTR(IBLTE(2),2) / UEDG(IBLTE(2),2)
       CTE_CTE1 = THET(IBLTE(1),1)/TTE
       CTE_CTE2 = THET(IBLTE(2),2)/TTE
       CTE_TTE1 = (CTAU(IBLTE(1),1) - CTE)/TTE
       CTE_TTE2 = (CTAU(IBLTE(2),2) - CTE)/TTE
C
C----- re-define D1 sensitivities wrt m since D1 depends on both TE Ds values
       DO 35 JS=1, 2
         DO 350 JBL=2, NBL(JS)
           J  = IPAN(JBL,JS)
           JV = ISYS(JBL,JS)
           D1_M(JV) = DTE_UTE1*UTE1_M(JV) + DTE_UTE2*UTE2_M(JV)
  350    CONTINUE
   35  CONTINUE
       D1_M(JVTE1) = D1_M(JVTE1) + DTE_MTE1
       D1_M(JVTE2) = D1_M(JVTE2) + DTE_MTE2
C
C----- "forced" changes from  UEDG --- USAV=UINV+dij*MASS  mismatch
       DUE1 = 0.
       DDS1 = DTE_UTE1*(UEDG(IBLTE(1),1) - USAV(IBLTE(1),1))
     &      + DTE_UTE2*(UEDG(IBLTE(2),2) - USAV(IBLTE(2),2))
C
      ELSE
C
       CALL BLSYS
C
      ENDIF
C
C---- Save wall shear for plotting output
      TAU(IBL,IS) = 0.5*R2*U2*U2*CF2
C
C---- set XI sensitivities wrt LE Ue changes
      IF(IS.EQ.1) THEN
       XI_ULE1 =  SST_GO
       XI_ULE2 = -SST_GP
      ELSE
       XI_ULE1 = -SST_GO
       XI_ULE2 =  SST_GP
      ENDIF
C
C---- stuff BL system coefficients into main Jacobian matrix
C
      DO 40 JV=1, NSYS
        VM(1,JV,IV) = VS1(1,3)*D1_M(JV) + VS1(1,4)*U1_M(JV)
     &              + VS2(1,3)*D2_M(JV) + VS2(1,4)*U2_M(JV)
     &              + (VS1(1,5) + VS2(1,5))
     &               *(XI_ULE1*ULE1_M(JV) + XI_ULE2*ULE2_M(JV))
   40 CONTINUE
C
      VB(1,1,IV) = VS1(1,1)
      VB(1,2,IV) = VS1(1,2)
C
      VA(1,1,IV) = VS2(1,1)
      VA(1,2,IV) = VS2(1,2)
C
      IF(LALFA) THEN
       VDEL(1,2,IV) = VSR(1)
      ELSE
       VDEL(1,2,IV) =
     &       (VS1(1,4)*U1_A + VS1(1,3)*D1_A)
     &     + (VS2(1,4)*U2_A + VS2(1,3)*D2_A)
     &     + (VS1(1,5) + VS2(1,5))*(XI_ULE1*ULE1_A + XI_ULE2*ULE2_A)
      ENDIF
C
      VDEL(1,1,IV) = VSREZ(1)
     &   + (VS1(1,4)*DUE1 + VS1(1,3)*DDS1)
     &   + (VS2(1,4)*DUE2 + VS2(1,3)*DDS2)
     &   + (VS1(1,5) + VS2(1,5))*(XI_ULE1*DULE1 + XI_ULE2*DULE2)
C
c  ## Modifications made for optimization study, April 6, 1990
c  ** Ri below refers to RCtau
      Ri_qi1 = vs1(1,4) - vs1(1,3) * term1    !dRi/dQi-1
      Ri_qi2 = vs2(1,4) - vs2(1,3) * term2    !dRi/dQi
c
c  ## computation of dRCtau/dAn and dRCtau/dBn storage in RHS
      call ExtraRHS (1, iv, ibl, i1, i, Ri_qi1, Ri_qi2)
C
      DO 50 JV=1, NSYS
        VM(2,JV,IV) = VS1(2,3)*D1_M(JV) + VS1(2,4)*U1_M(JV)
     &              + VS2(2,3)*D2_M(JV) + VS2(2,4)*U2_M(JV)
     &              + (VS1(2,5) + VS2(2,5))
     &               *(XI_ULE1*ULE1_M(JV) + XI_ULE2*ULE2_M(JV))
   50 CONTINUE
C
      VB(2,1,IV)  = VS1(2,1)
      VB(2,2,IV)  = VS1(2,2)
C
      VA(2,1,IV) = VS2(2,1)
      VA(2,2,IV) = VS2(2,2)
C
      IF(LALFA) THEN
       VDEL(2,2,IV) = VSR(2)
      ELSE
       VDEL(2,2,IV) =
     &       (VS1(2,4)*U1_A + VS1(2,3)*D1_A)
     &     + (VS2(2,4)*U2_A + VS2(2,3)*D2_A)
     &     + (VS1(2,5) + VS2(2,5))*(XI_ULE1*ULE1_A + XI_ULE2*ULE2_A)
      ENDIF
C
      VDEL(2,1,IV) = VSREZ(2)
     &    + (VS1(2,4)*DUE1 + VS1(2,3)*DDS1)
     &    + (VS2(2,4)*DUE2 + VS2(2,3)*DDS2)
     &    + (VS1(2,5) + VS2(2,5))*(XI_ULE1*DULE1 + XI_ULE2*DULE2)
C
c  ## Modifications made for optimization study, April 6, 1990
c  ** Ri below refers to Rtheta
      Ri_qi1 = vs1(2,4) - vs1(2,3) * term1    !dRi/dQi-1
      Ri_qi2 = vs2(2,4) - vs2(2,3) * term2    !dRi/dQi
c
c  ## computation of dRtheta/dAn and dRtheta/dBn storage in RHS
      call ExtraRHS (2, iv, ibl, i1, i, Ri_qi1, Ri_qi2)
C
      DO 60 JV=1, NSYS
        VM(3,JV,IV) = VS1(3,3)*D1_M(JV) + VS1(3,4)*U1_M(JV)
     &              + VS2(3,3)*D2_M(JV) + VS2(3,4)*U2_M(JV)
     &              + (VS1(3,5) + VS2(3,5))
     &               *(XI_ULE1*ULE1_M(JV) + XI_ULE2*ULE2_M(JV))
   60 CONTINUE
C
      VB(3,1,IV) = VS1(3,1)
      VB(3,2,IV) = VS1(3,2)
C
      VA(3,1,IV) = VS2(3,1)
      VA(3,2,IV) = VS2(3,2)
C
      IF(LALFA) THEN
       VDEL(3,2,IV) = VSR(3)
      ELSE
       VDEL(3,2,IV) =
     &       (VS1(3,4)*U1_A + VS1(3,3)*D1_A)
     &     + (VS2(3,4)*U2_A + VS2(3,3)*D2_A)
     &     + (VS1(3,5) + VS2(3,5))*(XI_ULE1*ULE1_A + XI_ULE2*ULE2_A)
      ENDIF
C
      VDEL(3,1,IV) = VSREZ(3)
     &    + (VS1(3,4)*DUE1 + VS1(3,3)*DDS1)
     &    + (VS2(3,4)*DUE2 + VS2(3,3)*DDS2)
     &    + (VS1(3,5) + VS2(3,5))*(XI_ULE1*DULE1 + XI_ULE2*DULE2)
C
c  ## Modifications made for optimization study, April 6, 1990
c  ** Ri below refers to Rm
      Ri_qi1 = vs1(3,4) - vs1(3,3) * term1    !dRi/dQi-1
      Ri_qi2 = vs2(3,4) - vs2(3,3) * term2    !dRi/dQi
c
c  ## computation of dRm/dAn and dRm/dBn storage in RHS
      call ExtraRHS (3, iv, ibl, i1, i, Ri_qi1, Ri_qi2)
C
      IF(IBL.EQ.IBLTE(IS)+1) THEN
C
C----- redefine coefficients for TTE, DTE, etc
       VZ(1,1)    = VS1(1,1)*CTE_CTE1
       VZ(1,2)    = VS1(1,1)*CTE_TTE1 + VS1(1,2)*TTE_TTE1
       VB(1,1,IV) = VS1(1,1)*CTE_CTE2
       VB(1,2,IV) = VS1(1,1)*CTE_TTE2 + VS1(1,2)*TTE_TTE2
```

```
C
        VZ(2,1)    = VS1(2,1)*CTE_CTE1
        VZ(2,2)    = VS1(2,1)*CTE_TTE1 + VS1(2,2)*TTE_TTE1
        VB(2,1,IV) = VS1(2,1)*CTE_CTE2
        VB(2,2,IV) = VS1(2,1)*CTE_TTE2 + VS1(2,2)*TTE_TTE2
C
        VZ(3,1)    = VS1(3,1)*CTE_CTE1
        VZ(3,2)    = VS1(3,1)*CTE_TTE1 + VS1(3,2)*TTE_TTE1
        VB(3,1,IV) = VS1(3,1)*CTE_CTE2
        VB(3,2,IV) = VS1(3,1)*CTE_TTE2 + VS1(3,2)*TTE_TTE2
C
      ENDIF
C
C---- turbulent intervals will follow if currently at transition interval
      IF(TRAN) TURB = .TRUE.
      TRAN = .FALSE.
C
      IF(IBL.EQ.IBLTE(IS)) THEN
C----- set "2" variables at TE to wake correlations for next station
C
       TURB = .TRUE.
       WAKE = .TRUE.
       CALL BLVAR(3)
      ENDIF
C
      DO 80 JS=1, 2
        DO 810 JBL=2, NBL(JS)
          JV = ISYS(JBL,JS)
          U1_M(JV) = U2_M(JV)
          D1_M(JV) = D2_M(JV)
  810   CONTINUE
   80 CONTINUE
C
      U1_A = U2_A
      D1_A = D2_A
C
      DUE1 = DUE2
      DDS1 = DDS2
C
C---- set BL variables for next station
      DO 190 ICOM=1, NCOM
        COM1(ICOM) = COM2(ICOM)
  190 CONTINUE
C
 1000 CONTINUE   ! with next streamwise station
C
      IF(TFORCE(IS)) THEN
       WRITE(6,9100) IS,XTR(IS),ITRAN(IS)
 9100  FORMAT(1X,'Side',I2,' forced transition at x/c = ',F7.4,I5)
      ELSE
       WRITE(6,9200) IS,XTR(IS),ITRAN(IS)
 9200  FORMAT(1X,'Side',I2,'  free  transition at x/c = ',F7.4,I5)
      ENDIF
C
```

```
 2000 CONTINUE   ! with next airfoil side
C
      RETURN
      END ! SETBL



      subroutine ExtraRHS (k, iv, ibl, i1, i, ri_qi1, ri_qi2)
c
c  ## ********************************************************************
c  *  This subroutine was added April 9, 1990 by Tom Sorensen.  It        *
c  *  provides the mechanism to fill RHS's with dR/dAn and dR/dBn.        *
c  ***********************************************************************
c
      integer k, iv, ibl, i1, i
      real    ri_qi1, ri_qi2
c
      INCLUDE 'xfoil.inc'
      INCLUDE 'xbl.inc'
c
c  ## NOTE: k = 1 => Ctau, k = 2 => theta, k = 3 => m
c  ** computation of dRi / dAn and storage in RHS# 2+(n+1)
      if (lthick) then
         do 26 iAn = ifirstAn, ilastAn
            iRHS = 3 + iAn - ifirstAn
c
            if (ibl.eq.2) then
c  **          must use stagnation quantities for station1: dq/dAn(Stag) = 0.0
               vdel (k,iRHS,iv) = Ri_qi2 * qa_an(i ,iAn)
            else
               vdel (k,iRHS,iv) = Ri_qi1 * qa_an(i1,iAn)
     &                          + Ri_qi2 * qa_an(i ,iAn)
            endif
 26      continue
      endif
c
c  ** computation of dRi / dBn and storage in RHS# (2+numAn)+(n+1)
      if (lcamber) then
         do 88 iBn = ifirstBn, ilastBn
            iRHS = (3 + numAn) + iBn - ifirstBn
c
            if (ibl.eq.2) then
c  **          must use stagnation quantities for station1: dq/dBn(Stag) = 0.0
               vdel (k,iRHS,iv) = Ri_qi2 * qa_bn(i ,iBn)
            else
               vdel (k,iRHS,iv) = Ri_qi1 * qa_bn(i1,iBn)
     &                          + Ri_qi2 * qa_bn(i ,iBn)
            endif
 88      continue
      endif
c
      return
      end     ! End of subroutine ExtraRHS
      SUBROUTINE TRCHEK(IBL,IS)
      INCLUDE 'xfoil.inc'
      INCLUDE 'xbl.inc'
C
      X1 = XSSI(IBL-1,IS)
      X2 = XSSI(IBL  ,IS)
C
C---- calculate AMPL2 value
      CALL DAMPL( HK1, T1, RT1, AX, AX_HK1, AX_T1 )
      AMPL2 = AMPL1 + AX*(X2-X1)
      AMI = AMPL2
C
C---- test for free or forced transition
      TRFREE = AMI.GE.AMCRIT
      TRFORC = XIFORC.GT.X1 .AND. XIFORC.LE.X2
C
C---- set transition interval flag
      TRAN = TRFORC .OR. TRFREE
C
      IF(.NOT.TRAN) THEN
       ITRAN(IS) = IBL+2
       RETURN
      ENDIF
C
C---- resolve if both forced and free transition
      IF(TRFREE .AND. TRFORC) THEN
       XT = (AMCRIT-AMPL1)/AX  +  X1
       TRFORC = XIFORC .LT. XT
       TRFREE = XIFORC .GE. XT
      ENDIF
C
      IF(TRFORC) THEN
C----- if forced transition, then XT is prescribed
       XT = XIFORC
       XT_A1 = 0.
       XT_X1 = 0.
       XT_T1 = 0.
       XT_D1 = 0.
       XT_U1 = 0.
       XT_X2 = 0.
       XT_T2 = 0.
       XT_D2 = 0.
       XT_U2 = 0.
      ELSE
C----- if free transition, XT is related to BL variables
       XT    =  (AMCRIT-AMPL1)/AX     + X1
       XT_AX = -(AMCRIT-AMPL1)/AX**2
C
       XT_A1 = -1.0/AX
       XT_X1 = 1.0
       XT_T1 = XT_AX*(AX_HK1*HK1_T1 + AX_T1)
```

```
        XT_D1 = XT_AX*(AX_HK1*HK1_D1         )
        XT_U1 = XT_AX*(AX_HK1*HK1_U1         )
        XT_X2 = 0.
        XT_T2 = 0.
        XT_D2 = 0.
        XT_U2 = 0.
       ENDIF
C
c ##    XTT's added 8/16/90
        XTT(is)    = xt
        XTR_A1(is) = xt_a1
        XTR_X1(is) = xt_x1
        XTR_T1(is) = xt_t1
        XTR_D1(is) = xt_d1
        XTR_U1(is) = xt_u1
c
C---- save transition location
       ITRAN(IS) = IBL
       TFORCE(IS) = TRFORC
       XSSITR(IS) = XT
C
C---- save info for user output
       IF(TFORCE(IS)) THEN
        XTR(IS) = XTR1(IS)
       ELSE
C----- interpolate airfoil geometry to find transition x/c
        SB1 = SST - X1
        SB2 = SST - X2
        IF(IS.EQ.2) THEN
         SB1 = SST + X1
         SB2 = SST + X2
        ENDIF
        XB1 = SEVAL(SB1,X,XP,S,N)
        XB2 = SEVAL(SB2,X,XP,S,N)
        XTR(IS) = XB1 + (XB2-XB1)*(XT-X1)/(X2-X1)
       ENDIF
C
       RETURN
       END ! TRCHEK
```

The modifications to allow multiple right hand side solutions of the Newton solver was straight forward and so is not included here.