AMSC663/664 Final Report: Analysis of the Adjoint Euler Equations as used for Gradient-based Aerodynamic Shape Optimization

Dylan Jude

May 18, 2017

Abstract

Adjoint methods are often used in gradient-based optimization because they allow for a significant reduction of computational cost for problems with many design variables. The proposed project focuses on the use of adjoint methods for two-dimensional airfoil shape optimization using Computational Fluid Dynamics to model the Euler equations.

1 Background

Airfoil shape optimization is the process of improving aerodynamic properties of an airfoil by altering its shape. The lift, drag, or pressure distribution are all examples of aerodynamic properties that can be used to analyze airfoil performance. Aerodynamic properties can be formalized mathematically as a cost function. Semantically this is introduced with the goal of minimizing the "cost" of an airfoil shape, and therefore minimizing the cost function.

Both design variables can also be mathematically defined to parameterize the shape of an airfoil. As an example, figure 1 shows an airfoil whose general shape can be altered using two variables α and c.

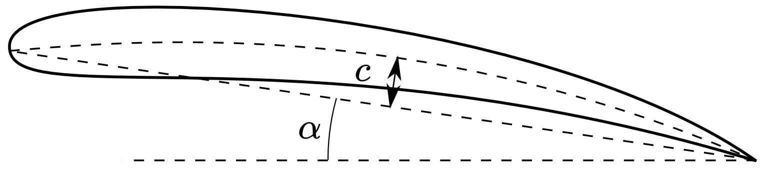


Figure 1: Example Airfoil Design Variables

1.1 Choosing a Cost Function

Assuming an existing airfoil shape and corresponding two-dimensional mesh, Computational Fluid Dynamics (CFD) can be used to solve the Euler equations over this mesh. From the airfoil, the flow solution is obtained using CFD, and the flow solution returns a pressure at every point on the airfoil. A simple cost function typically chosen in airfoil shape optimization compares the pressure distribution obtained from a flow solution to a desired pressure distribution. This is illustrated in figure 2. The x-axis follows the chord of the airfoil and therefore the two lines of each color represent the pressure at that location along the airfoil on the top and bottom surfaces. Other examples of cost functions could be to maximize the airfoil lift, or minimize the airfoil drag.

A mathematical formulation of a cost function for this comparison could be:

$$I_c(\alpha) = \oint_{airfoil} \frac{1}{2}(P - P_d)^2 ds \tag{1}$$

where α is a set of design variables used to obtain the airfoil shape, P is the resulting pressure distribution along the airfoil, and P_d is a desired or target pressure distribution. For an airfoil defined by a set of N discrete points, for convenience we can force the X-coordinates of each airfoil to be the same. This would result in the X-coordinate of each pressure curve to also be the same so that we can simplify the cost function:

$$I_c(\alpha) = \sum_{i=0}^{N} \frac{1}{2}(P_i - P_{d,i})^2 \tag{2}$$

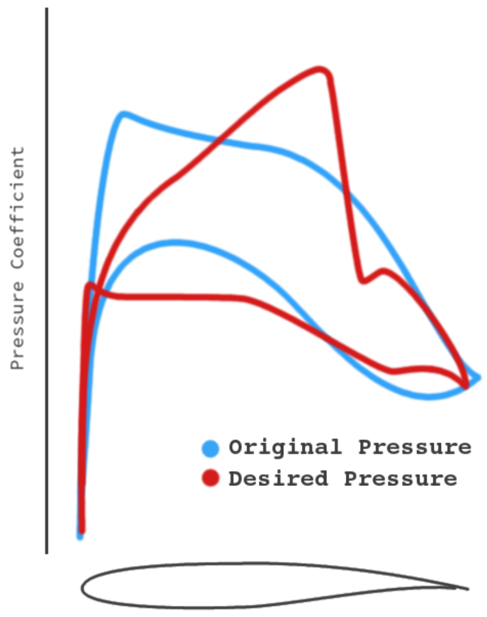


Figure 2: Comparison of a pressure distribution with the desired distribution [1]

where there are N points on the airfoil and i is the ith point.

1.2 Finding Sensitivities

For an example problem with two design variables α_1 and α_2, the sensitivity of the cost function I_c to these design variables is

$$\frac{\partial I_c}{\partial \alpha_1}, \quad \frac{\partial I_c}{\partial \alpha_2} \tag{3}$$

Each of these partial derivatives could be approximated using a finite-difference, or "brute-force" approach where for each variable

$$\frac{\partial I_c}{\partial \alpha_1} = \frac{I_c(\alpha_1 + \Delta\alpha_1) - I_c(\alpha_1)}{\Delta\alpha_1} \tag{4}$$

For two design variables, this requires three CFD calculations for $I_c(\alpha_{1,2})$, $I_c(\alpha_1 + \delta\alpha_1)$, and $I_c(\alpha_2 + \delta\alpha_2)$. Especially for complex, three-dimensional flow problems, obtaining the solution using CFD can take on the order of hours or days. Using brute-force finite differences to find the sensitivities of many design variables would therefore be a long and painstaking process.

The goal of using adjoint methods, as presented in the following section, is to eliminate the dependence of the cost function I_c on the flow solution so that all design variable sensitivities can be solved at once.

2 Approach

Since the adjoint Euler equations are derived from the Euler equations, this section will start with an overview of the Euler equations as solved by an in-house CFD solver. This overview will be followed by a brief derivation of the Adjoint Euler equations for the interior domain and boundary. For this project, only steady flow is considered; the time-derivative terms are kept in the initial derivation of the Euler and adjoint terms for completeness.

2.1 Euler Equations

The Euler equations are a simplification of the compressible Navier-Stokes equations for inviscid flow. For two-dimensional, these equations consist of four equations: one for the conservation of mass, two for the conservation of momentum in x and y, and one for the conservation of energy. In the following description of the flow equations, standard usage of flow variables are used. ρ is the fluid density, $\vec{u}$ is the fluid velocity composed of $u_1 and u_2$, E is total energy (internal and kinetic), and p is pressure.

2.1.1 Conservation of mass

Over a volume Ω, the conservation of mass in integral form is

$$\frac{d}{dt}\int_\Omega \rho d\Omega = 0 \tag{5}$$

which using the Reynolds Transport Theorem can be re-written as

$$\int_\Omega \left[\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \vec{u})\right] d\Omega = 0 \tag{6}$$

2.1.2 Conservation of Momentum

The time rate of change of momentum in volume Ω in the direction x_i is

$$\frac{d}{dt}\int_\Omega \rho u_i d\Omega = \int_S F_i dS$$

where F_i represents the stresses acting on the surface S of the domain. Again the Reynolds Transport Theorem can be used to simplify the equation to

$$\frac{d}{dt}\int_\Omega \rho u_i d\Omega = \int_\Omega \left[\rho \frac{Du_i}{Dt}\right] d\Omega \tag{7}$$

where $\frac{D}{Dt}$ is the material derivative, sometimes called the convective derivative.

For inviscid flow, there are no viscous stresses and the only force acting on the surface of the domain is pressure p. Since pressure acts inward,

$$\int_S F_i dS = \int_S -p\delta_{ki} n_k dS$$

Using Gauss' theorem, the surface integral is converted to a volume integral

$$\int_S -p\delta_{ki} n_k dS = \int_\Omega -\frac{\partial}{\partial x_k}(p\delta_{ki}) d\Omega$$

and combining with equation (7), we obtain the integral form of the momentum equation:

$$\int_\Omega \left[\rho \frac{Du_i}{Dt} + \frac{\partial p}{\partial x_i} = 0\right] d\Omega \tag{8}$$

2.1.3 Conservation of Energy

The conservation of energy in a control volume without body forces and without a heat source is related to the thermodynamic work done on the surface of the control volume from pressure and the rate of heat loss through the surface.

$$\frac{d}{dt}\int_\Omega \rho E d\Omega = \int_S p\delta_{ki} u_k n_k dS - \int_S -q_k n_k dS \tag{9}$$

In the above equation, q_k is defined by the Fourier law of heat conduction as $q_k = -\kappa(\partial T/\partial x_k)$. Again using the Reynolds Transport theorem, Gauss' theorem, and the definition of the material derivative, the energy equation can be simplified to:

$$\int_\Omega \left[\rho \frac{D}{Dt}(E) + \frac{\partial}{\partial x_k}(-p_{ki} u_i + q_k)\right] d\Omega = 0 \tag{10}$$

2.1.4 Euler Equations

The Euler equations for the conservation of mass, momentum, and energy can be written in conservative form as

$$\frac{\partial \vec{Q}}{\partial t} + \frac{\partial \vec{F}_{c,i}}{\partial x_i} = 0 \quad \text{in domain } \Omega, \quad i = 1, 2 \tag{11}$$

$$\vec{Q} = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho E \end{bmatrix}, \quad \vec{F}_{c,1} = \begin{bmatrix} \rho u_1 \\ \rho u_1^2 + p \\ \rho u_1 u_2 \\ (\rho E + p) u_1 \end{bmatrix}, \quad \vec{F}_{c,2} = \begin{bmatrix} \rho u_2 \\ \rho u_1 u_2 \\ \rho u_2^2 + p \\ (\rho E + p) u_2 \end{bmatrix} \tag{12}$$

To close the equations, the pressure is defined by the equation of state

$$p = \rho(\gamma - 1)\left[E - \frac{1}{2}||\vec{u}||^2\right] \tag{13}$$

where γ is the ratio of specific heats. Using a transformation to a Cartesian grid of coordinates ξ_i, the Euler equations can be written as

$$\frac{\partial \vec{q}}{\partial t} + \frac{\partial \vec{f}_{c,i}}{\partial \xi_i} = 0 \tag{14}$$

$$\vec{q} = J^{-1} \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ e \end{bmatrix}, \quad \vec{f}_{c,1} = J^{-1} \begin{bmatrix} \rho V_1 \\ \rho u_1 V_1 + \xi_{1,1} p \\ \rho u_2 V_1 + \xi_{1,2} p \\ (e + p) V_1 \end{bmatrix} \tag{15}$$

where J is the Jacobian of the coordinate transformation and V_i is the contravariant velocity in the ξ_i direction:

$$V_i = u_1 \xi_{i,1} + u_2 \xi_{i,2} \tag{16}$$

The discretization and mapping between Cartesian and curvilinear domains is covered in detail in the work of J. Blazek [2].

For this project the steady Euler equations are solved by performing transient iterations to make the residual $\partial \vec{q}/\partial t = 0$. This is done using local-timestepping to allow for the solution in each cell to advance with a constant Courrant-Friedrichs-Lewy (CFL) number, defined by the cell spacing (Δx), timestep (Δt) and local wave speed (a):

$$CFL = \frac{\Delta t}{\Delta x} a \tag{17}$$

The Euler equations are discretized explicitly using a first-order difference in both space and time. In one-spacial dimension, the explicit discretization for finding time $n+1$ is

$$\frac{\vec{q}^{n+1} - \vec{q}^{n}}{\Delta t} = -\left(\vec{f}^{n}_{c,i+1/2} - \vec{f}^{n}_{c,i-1/2}\right) - D^{n} \tag{18}$$

where D is artificial dissipation from the Roe-flux difference splitting scheme [3] or scalar dissipation [1].

The Euler equations are also discretized implicitly in time

$$\frac{\vec{q}^{n+1} - \vec{q}^{n}}{\Delta t} = -\left(\vec{f}^{n+1}_{c,i+1/2} - \vec{f}^{n+1}_{c,i-1/2}\right) - D^{n+1} \tag{19}$$

which can be approximately factored into a Diagonal Alternating Direction Implicit (DADI) scheme, outlined in detail by Pulliam and co-authors [4].

2.1.5 Boundary Conditions

For a “O” mesh topology, a 2D grid is defined by coordinates `j` and `k`, illustrated in figure 3. The `jmin` and jmax boundaries are periodic boundaries and the far-field can be approximated by a Dirichlet boundary by setting free-stream conditions. At the airfoil wall, the flow tangency condition must be satisfied:

$$(\vec{u} \cdot \vec{n}_{wall}) = 0 \tag{20}$$

where $\vec{n}_{wall}$ is the outward pointing wall-normal vector.

2.2 Continuous Adjoint Euler Equations

The adjoint to a set of equations is usually defined in one of two ways. The first uses a linear algebra approach to define the problem and the other uses the method of Lagrange variables. Since both methods are equivalent and previous studies in computational aerodynamic design tend to prefer the Lagrangian multiplier approach [5], this section will also motivate the use of adjoint methods using Lagrangian multipliers. There are two ways of implementing the adjoint equations in a code. The first is to take the adjoint of the continuous Euler equations and discretize at the end. The second method takes the adjoint of the already discretized Euler equations. The continuous adjoint is presented first as it is a more general form for the equations.

2.2.1 General Derivation

As presented in a previous section, we can define a cost function to minimize during the design process. This cost function can be a defined over the whole domain and/or over the boundary of the domain. The cost function is a function of both the flow solution q and geometry X and can be written as

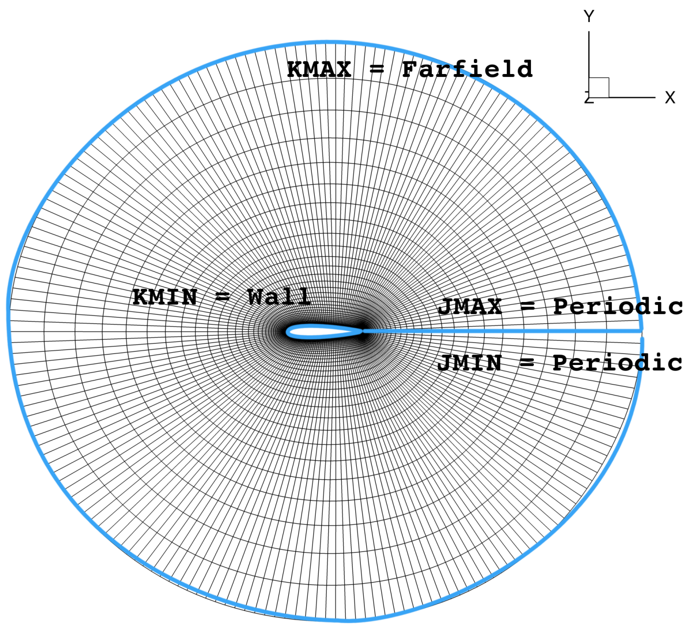


Figure 3: O-Mesh Topology with coordinates `j` and `k`

$$\delta I = \int_B \delta M(q, X) dB + \int_D \delta P(q, X) dD \tag{21}$$

where M is the contribution to the cost function from the boundary (B) and P is the contribution from the interior domain (D).

For simple problems, X could be the vector of design variables however more generally it represents the geometry of the grid. In CFD, the grid geometry consists of the cell volumes, face areas, and face vectors. These metrics appear directly in the flow equations as ξ_i, shown in equation (16), and J in equation (15).

Equation (21) can be further broken down into parts dependent on q and X:

$$\begin{aligned}\delta I &= \delta I_q + \delta I_X \\ &= \int_B \left[\frac{\partial M}{\partial q} \delta q + \frac{\partial M}{\partial X} \delta X \right] dB + \int_D \left[\frac{\partial P}{\partial q} \delta q + \frac{\partial P}{\partial X} \delta X \right] dD\end{aligned}$$

Now recalling the steady Euler equations, we can define the residual R and its dependence on q and X as:

$$\begin{aligned} R &= \left[\frac{\partial f_i}{\partial \xi_i}\right] &&= 0 \\ \partial R &= \left[\frac{\partial R}{\partial q}\right]\delta q + \left[\frac{\partial R}{\partial X}\right]\delta X &&= 0 \end{aligned}$$

Since this is equal to 0, we can subtract this from equation (21) with a Lagrangian multiplier ψ:

$$\delta I = \delta I_q + \delta I_X - \psi(\delta R_q + \delta R_X)$$

To eliminate dependence on q we focus on choosing ψ so that

$$\delta I_q + \psi(\delta R_q) = 0$$

$$\begin{aligned} R &= \frac{\partial f_i}{\partial \xi_i} &&= 0 \\ \frac{\partial R}{\partial q}\delta q &= \frac{\partial}{\partial q}\left[\frac{\partial}{\partial \xi_i}\delta f_i\right] &&= 0 \end{aligned}$$

As an integral over the whole domain, introducing the Lagrange multiplier ψ as the weak form variable:

$$\int_D \frac{\partial}{\partial \xi_i}\delta f_i = \int_D \psi^T \frac{\partial}{\partial \xi_i}\delta f_i = 0$$

integrating by parts

$$\int_B \left[n_i \psi^T \delta f_i\right] dB - \int_D \left[\frac{\partial \psi}{\partial \xi_i}\delta f_i\right] dD = 0$$

since this is zero, we can subtract it from the δI equation. ψ is then a Lagrangian multiplier for the optimization of I with constraint equation $R = 0$.

$$\begin{aligned} \delta I = &\int_B \delta M(q, X) dB + \int_D \delta P(q, X) dD \\ &- \int_B \left[n_i \psi^T \delta f_i\right] dB + \int_D \left[\frac{\partial \psi}{\partial \xi_i}\delta f_i\right] dD \end{aligned}$$

we then pick ψ to eliminate all dependence on δq.

2.2.2 Interior Equations

Our cost function, as previously presented in equation (2), involves only an integral of pressure over the surface of the airfoil. Since the surface of the airfoil is along the boundary, the P for this case is 0. The interior equation then becomes:

$$\begin{aligned} \int_D \left[\frac{\partial \psi}{\partial \xi_i} \frac{\partial f_i}{\partial q} \right] dD =& 0 \\ \frac{\partial \psi}{\partial \xi_i} \frac{\partial f_i}{\partial q} =& \quad 0 \end{aligned}$$

using the definition of flux Jacobian $A_i = \partial f_i / \partial q$, the adjoint residual is

$$[A_i]^T \frac{\partial \psi}{\partial \xi_i} = 0 \tag{22}$$

This form looks very similar to the original Euler equation residual, which was

$$\frac{\partial f}{\partial \xi_i} = [A_i]^T \frac{\partial q}{\partial \xi_i} = 0$$

2.2.3 Boundary Equations

We also want to eliminate the dependence on q over the boundary. This is done very similarly to the interior equations however since $M \neq 0$, it remains in the formulation:

$$\begin{aligned} \int_B \frac{\partial M}{\partial q} \delta q dB - \int_B \left[n_i \psi^T \frac{\partial f_i}{\partial q} \delta q \right] dB = 0 \\ \frac{\partial M}{\partial q} = n_i \psi^T \frac{\partial f_i}{\partial q} \end{aligned}$$

2.3 Discrete Adjoint Euler Equations

The discrete adjoint is derived directly from the discrete Euler equations. This results in more terms for the final code however has the benefit of being consistent with sensitivities obtained from brute-force Euler computations.

2.3.1 Flux Terms

As a slight modification from the Euler derivation in section 2.1, let f denote flux in j-coordinate direction and g denote flux in k-coordinate direction.

$$\frac{\partial q}{\partial t} + \frac{\partial f}{\partial \xi_1} + \frac{\partial g}{\partial \xi_2} = 0$$

Let the residual of the steady Euler Equation be defined as:

$$R^n = \frac{q^{n+1} - q^n}{\Delta t} = 0 \tag{23}$$

The residual expanded in both dimensions j, k at time n is

$$R^n_{j,k} = -\left(f_{j+1/2,k} - f_{j-1/2,k}\right) - \left(g_{j,k+1/2} - g_{j,k-1/2}\right) \tag{24}$$

Similar to the continuous relation, the method of Lagrange multipliers is applied to the cost function subject to the Euler equation (residual R) as a constraint:

$$\delta I_q - \psi(\delta R_q) = 0$$

This becomes the new PDE to solve. The term $\psi(\delta R_q)$ can be auto-differentiated or derived by-hand:

$$\partial R^n_{j,k} = -\left(\partial f_{j+1/2,k} - \partial f_{j-1/2,k}\right) - \left(\partial g_{j,k+1/2} - \partial g_{j,k-1/2}\right)$$

$$\partial R^n_{j,k} = -\left(\frac{\partial f}{\partial q}\delta q\right)_{j+1/2,k} + \left(\frac{\partial f}{\partial q}\delta q\right)_{j-1/2,k} - \left(\frac{\partial g}{\partial q}\delta q\right)_{j,k+1/2} + \left(\frac{\partial g}{\partial q}\delta q\right)_{j,k-1/2} \tag{25}$$

Using the definition of the flux Jacobian A and B:

$$\left(\frac{\partial f}{\partial q}\delta q\right)_{j,k} = (A\delta q)_{j,k} \quad , \quad \left(\frac{\partial g}{\partial q}\delta q\right)_{j,k} = (B\delta q)_{j,k}$$

so that the equations simplify to

$$\partial R^n_{j,k} = -(A\delta q)_{j+1/2,k} + (A\delta q)_{j-1/2,k} - (B\delta q)_{j,k+1/2} + (B\delta q)_{j,k-1/2} \tag{26}$$

Recall from the definition of the flux in equation (15) that the flux f contains both information from the conservative vector q as well as the cell metrics ξ_1, ξ_2. The cell metrics can more easily be though of as the cell geometry and can be directly related to the cell face vectors $S_{j,k}$. The Euler equations require information at the cell face, for example $f_{j+1/2}$, which is a combination of face geometry and interpolated values of the cell-centered conservative information.

To be consistent with the discretization of the Euler equations, the interface flux jacobian must be defined as

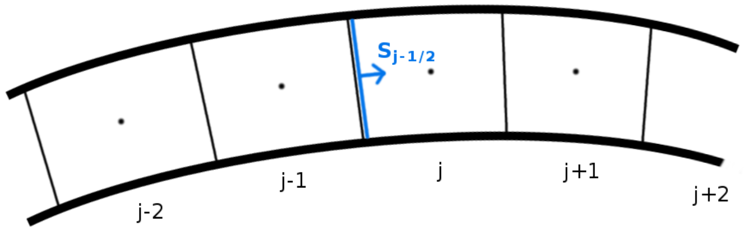


Figure 4: Illustration of cell-center location where conservative variables are stored and face location where geometry is well defined

$$A_{j+\frac{1}{2}} = \left[\frac{1}{2}(A_j + A_{j+1})\right]_{S@j+\frac{1}{2}} \tag{27}$$

2.3.2 Dissipation Terms

Numerical dissipation terms must be added to the Euler equations for stability. Typically a Roe Flux-difference splitting scheme results in very good numerical stability and accuracy in regions with shocks [3]. As a first-pass, however, scalar dissipation can be used to approximate the flux jacobian matrix as the magnitude of the largest eigenvalue. This methodology is outlined in detail in the work of Nadarajah [1]. The dissipation to the Euler equation is also applied at cell interfaces. In one-dimension, the dissipation h applied to the Euler residual appears as:

$$R_j = -\left(f_{j+\frac{1}{2}} + f_{j-\frac{1}{2}}\right) - \left(h_{j+\frac{1}{2}} + h_{j-\frac{1}{2}}\right)$$

Where the scalar dissipation term $h_{j+\frac{1}{2}}$ is

$$h_{j+\frac{1}{2}} = \epsilon\sigma\left(q_{j+1} - q_j\right)$$

,

$\epsilon \approx 0.25$ [1] and σ is the spectral radius scaled by the face area

$$\sigma = ||V|| + c||S_{j+\frac{1}{2}}||$$

and c is the speed of sound. Both V and c are functions of the flow q however according to Nadarajah [1], σ does not vary significantly and can be considered constant in deriving the adjoint dissipation terms.

$$\partial R^n_{j,k} = \quad \dots \quad - \left(\frac{\partial h}{\partial q}\delta q\right)_{j+1/2,k} - \left(\frac{\partial h}{\partial q}\delta q\right)_{j-1/2,k} \tag{28}$$

$$\frac{\partial h}{\partial q}_{j+1/2,k} = \frac{1}{2}\left[\left(\frac{\partial h}{\partial q}\right)_j + \left(\frac{\partial h}{\partial q}\right)_{j+1}\right]_{S@j+1/2} \tag{29}$$

$$\frac{\partial h}{\partial q}_j = -\epsilon\sigma \quad , \quad \frac{\partial h}{\partial q}_{j+1} = \epsilon\sigma \tag{30}$$

While σ may be held constant in the variation of the residual with respect to q, after the adjoint solution is found the residual is differentiated with respect to X to find the final sensitivity. For this differentiation, the variation of the residual with respect to the metrics X cannot be ignored.

2.4 Auto-Differentiation

The previous section deriving the by-hand discrete adjoint equations showed a term-by-term breakdown of the flux and dissipation differentiation to find the adjoint equations. This term-by-term differentiation can also be accomplished by automatic (or sometimes referred to as algorithmic) differentiation. Many research groups have shown successful implementations of adjoint methods using auto-differentiation [6]. These methods tend to be less efficient than by-hand adjoint solvers however have shown to produce accurate results [7].

As a first pass at implementing adjoint methods, it is convenient to rely upon auto-differentiation software such as *Tapenade* [8] to quickly develop an adjoint solver. This can be compared with sensitivities from finite-difference ("brute-force") gradients.

Auto-differentiation can be done in two directions: forward (sometimes called tangent) and backward (sometimes called adjoint or reverse). The forward-mode is the more mathematically intuitive way of defining sensitivities. Given design variables α, which affect the grid X, which affect the flow solution Q, which affects the cost function I_c:

$$\frac{\partial I_c}{\partial \alpha} = \frac{\partial I_c}{\partial Q}\frac{\partial Q}{\partial X}\frac{\partial X}{\partial \alpha} \tag{31}$$

Using "dot" notation to denote the partial derivative of a variable with respect to α, the above equation is executed in the order:

$$\dot{\alpha} \quad \rightarrow \quad \dot{X} \quad \rightarrow \quad \dot{Q} \quad \rightarrow \quad \dot{I}$$

In contrast the adjoint formulation is represented mathematically as:

$$\left(\frac{\partial I_c}{\partial \alpha}\right)^T = \left(\frac{\partial X}{\partial \alpha}\right)^T \left(\frac{\partial Q}{\partial X}\right)^T \left(\frac{\partial I_c}{\partial Q}\right)^T \tag{32}$$

Using "bar" notation to denote the partial derivative of the cost function I_c with respect to a variable, equation (32) is executed backwards:

$$\bar{\alpha} \quad \leftarrow \quad \bar{X} \quad \leftarrow \quad \bar{Q} \quad \leftarrow \quad \bar{I}$$

This methodology is outlined in much greater detail in the work of Giles, Ghate, and Duta [9]. One highlighted result of the authors' paper is that the "dot" and "bar" quantities can be combined at any step to recover the desired cost function sensitivity:

$$\dot{I}_c = \frac{\partial I_c}{\partial \alpha} = \bar{Q}^T \dot{Q} = \bar{X}^T \dot{X} = \bar{\alpha}^T \dot{\alpha} \tag{33}$$

This is especially convenient since the code developer may want to only auto-differentiate the computationally expensive routines and use intuitive forward differentiation for the rest. This procedure will be used for this project where auto-differentiation is only carried out to compute $\bar{X}$. Since grid generation is relatively cheap, $\dot{X}$ can be easily computed in forward mode through finite differences.

2.5 Gradient-based optimization

The previous section on auto-differentiation mentions the forward differentiation of the grid variation for each design variable ($\partial I/\partial X$). A very similar final step is required for the hand-derived adjoint Euler equations. From the solution to the adjoint Euler equations, the last step involves solving for the variation of the cost function with the geometry X but holding q constant:

$$\delta I = \left\{ \frac{\partial I^T}{\partial X} - \psi^T \left[\frac{\partial R}{\partial X} \right] \right\} \delta X$$

This equation depends on grid geometry X, which as shown in section (2.2) is not typically a simple array of the design variables. Instead the above sensitivities are found with respect to grid metrics X, and the variation of X with the design variables α_i can be found through brute-force finite-difference grid generation. Re-generating meshes for every design variable α is typically fast compared to a flow calculation, especially in 2D cases considered for this project.

Once the sensitivities are computed using this approach, each design variable can be altered in a gradient-based algorithm to approach a local minimum. The details and implementation of a gradient-based optimization algorithm are beyond the scope of this project. Many aerodynamic optimization codes use external libraries such as SNOPT [10]. For simplicity, since the Adjoint and Euler codes are wrapped in python for communication, the python SciPy optimization library was used with both the Sequential Least Squares Programming (SLSQP)[11] and Conjugate Gradient (CG) [12] optimization methods.

2.6 Hicks-Henne Bump Functions

The entire airfoil design process relies upon a chosen set of design variables to alter the airfoil shape. The design variables need to be defined by continuous functions in order for gradient-based optimization to work well. One such method of parameterizing airfoil perturbations was presented by Hicks and Henne in 1977, and is commonly referred to as "Hicks-Henne Bump Functions"[13]. These bump functions are sinusoidal perturbations applied at different locations along the airfoil. A commonly used form is

$$b(x) = a\left[sin\left(\pi x^{\frac{log(0.5)}{log(t_1)}}\right)\right]^{t_2}, \quad \text{for } 0 \leq x \leq 1 \tag{34}$$

In this bump equation, t_1 locates the maximum of the bump in $0 \leq x \leq 1$, t_2 controls the width of the bump, and a controls the bump amplitude. Each bump has three design variables. Figure 5 shows an example of 3 random perturbations made to t_1 and a on 6 bumps while keeping t_2 constant. From the original shape (dotted line), this example with 6 bump function, a total of 12 variables, were able to significantly alter the shape of an airfoil. For simplicity the 12 design variables for each of the 3 random perturbation are not given here however examples values for the Hicks-Henne design variables are given in section 3.1.3.

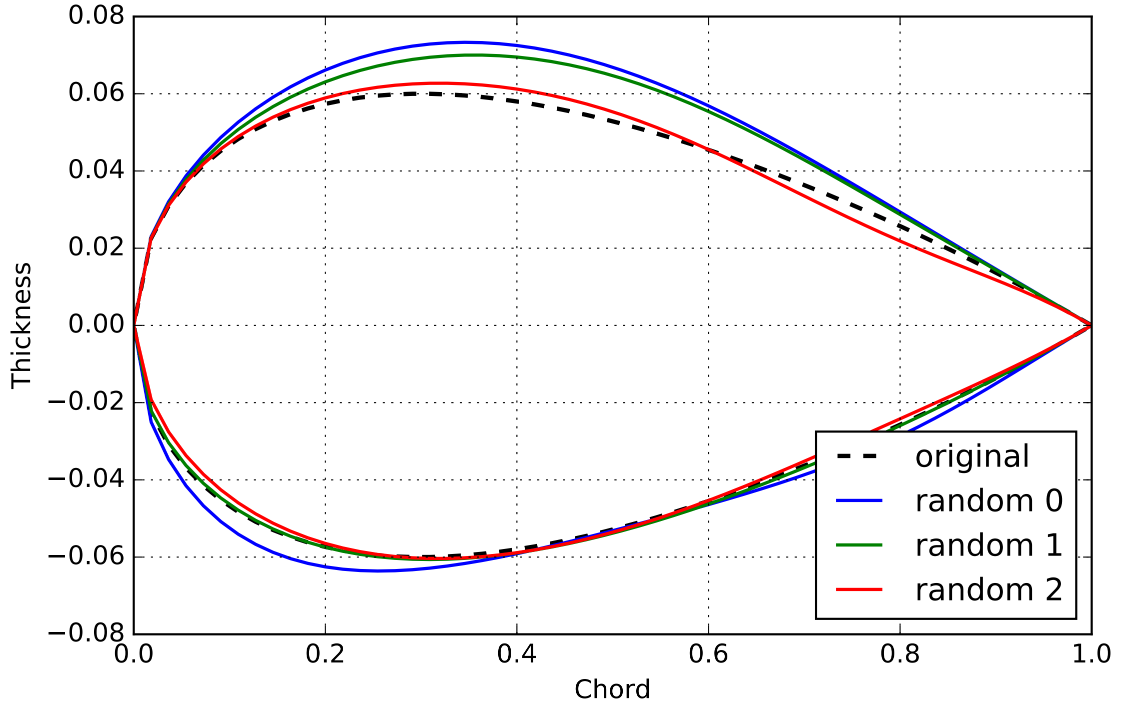


Figure 5: Hicks-Henne Bump functions with random variable perturbations

3 Results

3.1 Framework Setup

The initial milestones for this project were to set up a framework that could be used for gradient-based airfoil optimization for the Euler equations using the Adjoint to compute gradients. An overview of the framework is illustrated in figure 6. The numbers in this figure correspond to both the order in which the modules were implemented as well as the function call order for a single step of the design optimization process.

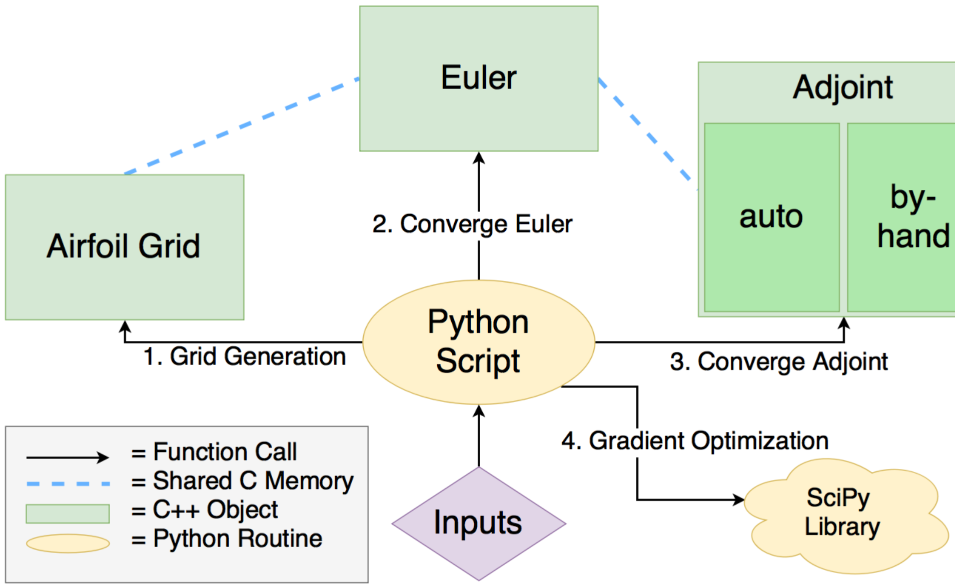


Figure 6: Overview of Adjoint-Euler Framework wrapped in Python

3.1.1 Grid Generation

Two-dimensional mesh generation is traditionally done by solving the Poisson equation:

$$\xi_{xx} + \xi_{yy} = P_s$$
$$\eta_{xx} + \eta_{yy} = Q_s$$

The notation for ξ and η here are ξ_1 and ξ_2 from the grid metric terms as presented in equation (15). Source terms P_s and Q_s are marked with the subscript to be distinguished from the flow variables or pressure while keeping consistency with the notation from Steger and Sorenson [14]. A mesh generation code had previously been developed for CFD 2 (ENAE 685) but instead using the Laplace equation ($P_s = Q_s = 0$)[15]. The source terms derived by Steger and Sorenson are meant to improve the mesh for CFD computations by enforcing uniform spacing near boundaries and orthogonality at solid-wall surfaces. The source terms are not given in this work but are described in great detail in the original paper by Steger and Sorenson [14].

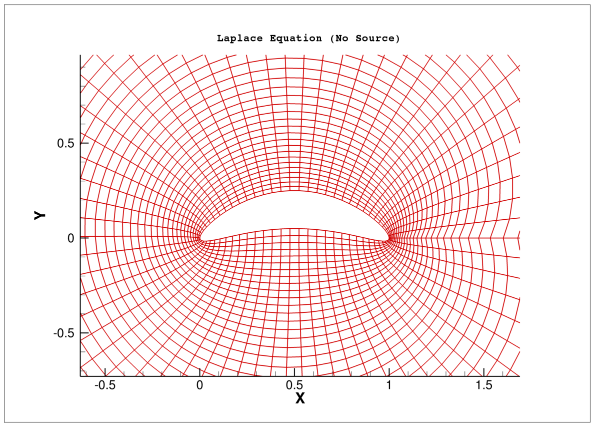


Figure 7: Grid generation without source terms.

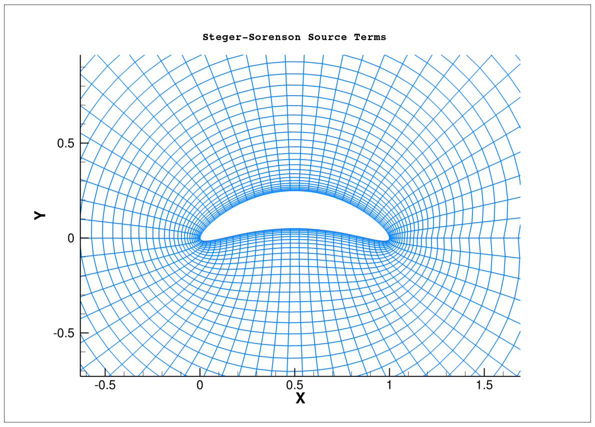


Figure 8: Grid generation with Steger-Sorenson source terms shows more uniform spacing normal to the wall.

Figures 7 and 8 show how the grid quality is improved especially near the wall when the Steger-Sorenson source terms are introduced. A simplified geometry was used here to highlight the especially poor mesh in the near-body for the Laplace equation in concave regions.

The mesh-generation C++ class was wrapped in python for convenience. The code to generate a NACA0012 with 93 points on the upper surface and 64 points in the normal direction with initial wall-spacing of 0.001 is shown in the following listing.

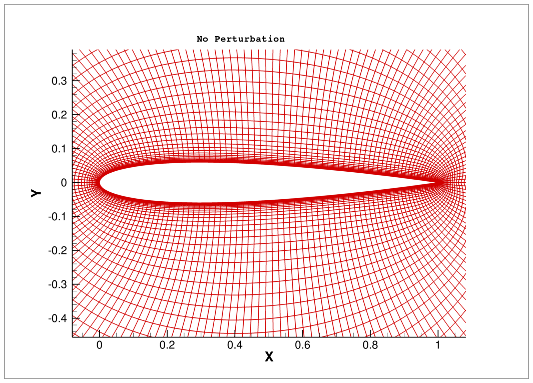
```
# ----------------------------------------
# Airfoil Surface
#
airfoil = naca.naca4('0012', 93, False, True)
# ----------------------------------------
# Mesh Generation
#
mg   = libflow.MeshGen(airfoil, 64, 0.001)
mg.poisson(500)
xy   = mg.get_mesh()
```

3.1.2 Airfoil Perturbation

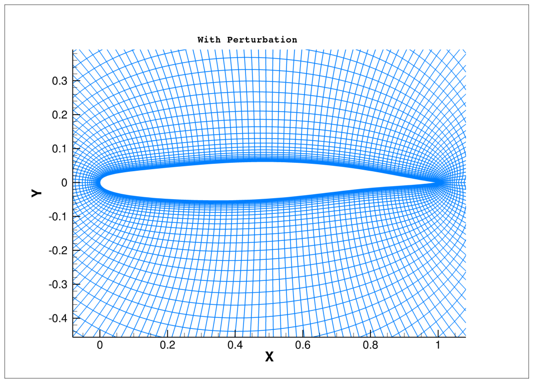
Airfoil perturbations are done using the Hicks-Henne "bump" function described in an earlier section, governed by equation (34). Each bump is parameterized by three variables; one for the location, one for the amplitude, and one for the width of the bump. Furthermore bumps must be specified for both the upper and lower surfaces of the airfoil. The following python listing shows example code to create 6 bumps: 3 on the bottom and 3 on the top of the airfoil. The width of the bump in this case is fixed and only the location and amplitude are design variables. The first and second lines of the design variable array are the locations of the bumps on the lower and upper surface respectively. The third and fourth lines are the amplitudes of the bumps on the lower and upper surface respectively. Figure 9(a) shows the original NACA0012 airfoil, and figure 9(b) shows the airfoil grid after perturbations.

```
#---------------------------------------------
# Hicks Henne Perturbation
#
design_vars = np.array([[ 0.25,  0.50 , 0.75 ],  # lower loc
                        [ 0.25,  0.50 , 0.75 ],  # upper loc
                        [ 0.01, -0.005, 0.01 ],  # lower amp
                        [-0.02,  0.01 , 0.005]]) # upper amp

airfoil     = perturb(airfoil,design_vars)
```



(a) Original NACA0012 grid



(b) Perturbed grid with 6 bumps

Figure 9: Comparison between perturbed and un-perturbed grid

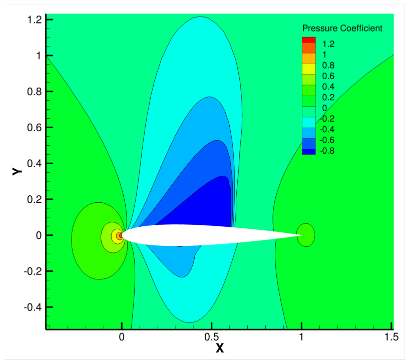
3.1.3 2D Euler Code

An in-house 3D Navier-Stokes code was trimmed down into a 2D Euler solver. The Euler solver is not intended to be a high-order accuracy code and therefore only employs first-order flux reconstruction at cell interfaces. The code can be run with explicit time marching or with implicit time marching using a Diagonalized Alternating Direction Implicit (DADI) inversion routine.

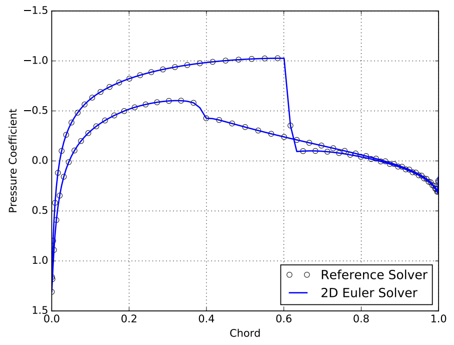
As an example test case for validation, the 2D Euler solver was compared with the original 3D in-house code for a NACA0012 airfoil at 1.25° angle of attack in Mach 0.8 flow. The 3D code was run in without viscous terms and also with only a first-order flux reconstruction. Running a 2D case in a 3D solver is accomplished by extruding the 2D grid and applying periodic boundary conditions in the Z-direction. Figure 10(a) shows the solution pressure contours from the 2D flow solver and figure 10(b) shows the comparison with the trusted 3D reference code. As expected, the results are exactly the same. Figure 10(c) shows the convergence of the 2D Euler solver. Realistically only a few orders of convergence is required; convergence to machine zero is shown for completeness.

The developed solver was wrapped in Python for convenience and modular connectivity with other parts of the adjoint framework. The python code to read inputs, take 1000 timesteps, and get the surface pressure is shown in the following listing.

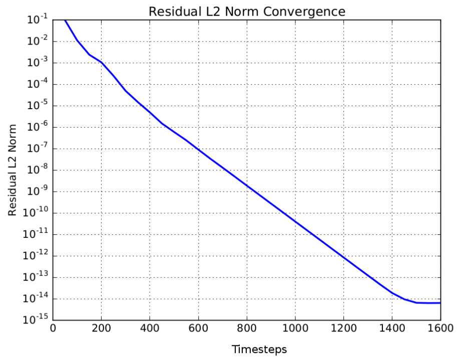
```python
# --------------------------------------------------
# Start CFD
inputs     = euler_utils.read_inputs("input.yaml")
euler      = libflow.Euler(grid, yaml.dump(inputs))
euler.take_steps(1000)
pressure   = euler.pressure()
```



(a) Example solution pressure contours



(b) Comparison of pressure distribution with original in-house 3D solver



(c) L2 norm convergence of the Euler solver residual

Figure 10: Example solution of Euler equation for transonic airfoil

3.1.4 Adjoint Solvers

Both the Auto-Differentiated and By-Hand Adjoint solvers have the same routine names and can be thought of as two separate implementations of the same class. Either version of the adjoint code can be used to compute gradients and the solution from one adjoint code can be saved, then read by the other adjoint code for debugging purposes. This works because in theory both adjoint codes solve exactly the same discrete adjoint equations. In reality the only difference between codes is that the auto-differentiated code does not make the simplification of the dissipation derivatives presented in section 2.3.2.

Auto-differentiation of the Euler code was conducted using *Tapenade*[8]. Support for the C programming appears to be new for *Tapenade* and while there are a few unsupported features and warning messages, by simplifying some of the routines in Euler solver the software worked as expected for the explicit routines. It was shown in section 2.4 that auto-differentiation in adjoint mode for a code presents itself as differentiating the code in reverse starting at the cost function and ending at the design variables. The routines within the Euler solver are roughly structured as follows:

```
while(n < nsteps){

  boundary_conditions(Q, X);

  flux(Q, R, X);

  for(i=0; i<all_pts; i++)
    Q[i] = Q[i] + R[i]*dt[i];

  n++;
}
compute_cost_function(Q, I);
```

In this pseudo-code, `Q` is the vector of flow variables, `X` is the grid, and `R` is the residual. Reverse differentiation will find the "bar" of each of these variables using the suffix "b" ($\bar{Q}$ = `Qb`) by differentiating each of the three shown subroutines. When auto-differentiating in reverse mode, *Tapenade* also adds a "_b" suffix to the function name. The resulting adjoint code looks like this:

```
Ib = 1.0
compute_cost_function_b(Q, Qb, I, Ib);
while(n < nsteps){

  flux_b(Q, Qb, R, Rb, X, Xb);

  boundary_conditions_b(Q, Qb, X, Xb);

  for(i=0; i<all_pts; i++)
    Rb[i] = Rb[i] + Qb[i]*dt[i];

  n++;
}
```

The first line of the above adjoint pseudo-code is $\bar{I} = 1.0$. This is because $\bar{I} = \partial I/\partial I = 1$. The rest of the "bar" variables start at zero and are updated from the cost function, flux routines, and boundary conditions in that order. The end goal of reverse-differentiation is to obtain $\bar{X}$ =`Xb` because, again from section 2.4

$$\frac{\partial I}{\partial \alpha} = \bar{X}^T \dot{X}$$

where $\dot{X}$ is still computed in forward-mode using finite differences since grid-generation is computationally inexpensive:

$$\dot{X} = \frac{\partial X}{\partial \alpha_i} \approx \frac{X(\alpha_i + \Delta\alpha_i) - X(\alpha_i)}{\Delta\alpha_i}$$

3.2 Gradient Computation

Analysis of gradients for 6-design variable case was conducted using brute-force Euler, auto-differentiated adjoint, and by-hand adjoint methods. The term "gradient" here is sometimes referred to "sensitivity" because the gradient of the cost function with respect to the design variable can also be thought of as the sensitivity of the cost function to that design variable. The terms "gradient" and "sensitivity" will be used interchangeably in this section.

For the 6-variable case, the location of the bumps are held fixed at 25%, 50%, and 75% along the airfoil chord and only the amplitude of the bumps is allowed to change. The airfoil chord is defined as the length from the nose of the airfoil to the tail. Both initial and final airfoils are slight variations from the NACA0012 airfoil, the Mach number of the flow is 0.5 and the angle of attach is 5.0 degrees.

Table 1 shows the tabulated gradients of the cost function with respect to all 6 design variables. The sensitivities are shown for different finite-difference values of each variable $\Delta\alpha$. Using $\Delta\alpha \geq 10^{-4}$ appears to be too large to approximate the gradients for both the brute-force and adjoint methods. Also values of $\Delta\alpha < 10^{-14}$ start showing variations from round-off error.

Gradients from the brute-force and AD adjoint methods match very well with each other. The by-hand adjoint, however, does not match exactly with the AD adjoint only because of the approximation of the dissipation flux. This approximation was presented in section 2.3.2. If the auto-differentiated routine is used for the dissipation flux in the by-hand code, both adjoint methods return the same gradients to machine precision.

	$\Delta\alpha$	$\partial I/\partial\alpha_1$	$\partial I/\partial\alpha_2$	$\partial I/\partial\alpha_3$	$\partial I/\partial\alpha_4$	$\partial I/\partial\alpha_5$	$\partial I/\partial\alpha_6$
Brute-Force	10^{-4}	-0.1228517	-0.0530909	-0.1132482	-1.2072553	-0.0690058	0.1432287
	10^{-6}	-0.1242372	-0.0544189	-0.1160899	-1.2186430	-0.0717039	0.1404430
	10^{-8}	-0.1242511	-0.0544322	-0.1161184	-1.2187570	-0.0717309	0.1404151
	10^{-10}	-0.1242515	-0.0544327	-0.1161186	-1.2187583	-0.0717313	0.1404148
	10^{-12}	-0.1242348	-0.0544265	-0.1161202	-1.2188544	-0.0717290	0.1403521
	10^{-14}	-0.1265480	-0.0598045	-0.1235123	-1.2189468	-0.0741594	0.1357421
AD Adjoint	10^{-4}	-0.1242607	-0.0545077	-0.1161888	-1.2188952	-0.0718753	0.1403117
	10^{-6}	-0.1242597	-0.0545077	-0.1161888	-1.2188960	-0.0718753	0.1403117
	10^{-8}	-0.1242597	-0.0545077	-0.1161888	-1.2188960	-0.0718753	0.1403117
	10^{-10}	-0.1242601	-0.0545082	-0.1161890	-1.2188961	-0.0718757	0.1403112
	10^{-12}	-0.1242750	-0.0545194	-0.1162164	-1.2189904	-0.0718894	0.1402322
	10^{-14}	-0.1259956	-0.0570447	-0.1235461	-1.2210666	-0.0736178	0.1374561
Hand Adjoint	10^{-4}	-0.1253349	-0.0538379	-0.1134897	-1.2354522	-0.0732670	0.1439928
	10^{-6}	-0.1253339	-0.0538378	-0.1134897	-1.2354525	-0.0732670	0.1439928
	10^{-8}	-0.1253339	-0.0538378	-0.1134897	-1.2354525	-0.0732670	0.1439928
	10^{-10}	-0.1253342	-0.0538389	-0.1134903	-1.2354524	-0.0732669	0.1439923
	10^{-12}	-0.1253448	-0.0538821	-0.1135104	-1.2355791	-0.0732398	0.1439078
	10^{-14}	-0.1305186	-0.0598827	-0.1242959	-1.2386824	-0.0762561	0.1402745

Table 1: Adjoint and brute-force gradient comparison

Both the brute-force and adjoint methods of finding the design variable sensitivities rely upon an approximation of the first derivative using a finite-difference formula derived from the Taylor series expansion. In the adjoint approach, the Taylor expansion is performed for the grid coordinates X:

$$\frac{\partial X}{\partial \alpha_i} = \frac{X(\alpha_i + \Delta\alpha_i) - X(\alpha_i)}{\Delta\alpha_i} - \frac{\Delta\alpha}{2}\frac{\partial^2 X}{\partial \alpha^2} + O(\Delta\alpha^2) \tag{35}$$

In the brute-force approach, the Taylor expansion is performed for the cost function I_c:

$$\frac{\partial I_c}{\partial \alpha_i} = \frac{I_c(\alpha_i + \Delta\alpha_i) - I_c(\alpha_i)}{\Delta\alpha_i} - \frac{\Delta\alpha}{2}\frac{\partial^2 I_c}{\partial \alpha^2} + O(\Delta\alpha^2)p \tag{36}$$

From table 1, at a glance it appears the adjoint methods gives better sensitivities than the brute-force approach because the sensitivities show less variation. This can be analyzed more rigorously by looking at the truncation error from the Taylor expansions above. In the brute-force method Taylor series, the dominant error is the second derivative term of I_c:

$$Error_{\text{brute-force}} \sim \frac{\Delta\alpha}{2}\frac{\partial^2 I_c}{\partial \alpha^2} \tag{37}$$

In the adjoint equation, the truncated second order derivative term from the expansion of X is combined with the backwards differentiated $\bar{X}$:

$$\left[\frac{\partial I_c}{\partial \alpha_i}\right]_{adjoint} = \bar{X}^T \dot{X}$$

$$Error_{\text{adjoint}} \sim \bar{X}^T \frac{\Delta\alpha}{2}\frac{\partial^2 X}{\partial \alpha^2} \tag{38}$$

In both cases, the second derivative can be approximated with the finite difference formula:

$$\frac{\partial^2 I_c}{\partial \alpha^2} \approx \frac{I_c(\alpha + \Delta\alpha) - 2I_c(\alpha) + I_c(\alpha - \Delta\alpha)}{\Delta\alpha^2}$$
$$\frac{\partial^2 X}{\partial \alpha^2} \approx \frac{X(\alpha + \Delta\alpha) - 2X(\alpha) + X(\alpha - \Delta\alpha)}{\Delta\alpha^2}$$

The comparison of the computed truncation error for a single variable, α_3 was conducted for $\Delta\alpha = 10^{-8}$. The results, summarized in table 2, confirm that the adjoint truncation is less than that of the brute-force approach for variable α_3. We expect the truncation error would behave similarly for other variables.

Brute-Force Truncation Error	$\frac{\Delta\alpha}{2}\frac{\partial^2 I_c}{\partial\alpha^2}$	3.653466e-05
Adjoint Truncation Error	$\bar{X}^T\frac{\Delta\alpha}{2}\frac{\partial^2 X}{\partial\alpha^2}$	1.668839e-08

Table 2: Approximate truncation error for adjoint and brute-force method

3.3 Timing Results and Parallelization

One major advantage of using a by-hand adjoint is having full control over memory usage for optimization. Whereas the auto-differentiated adjoint is difficult to parallelize with OpenMP, doing so for the by-hand adjoint is trivial and shows tremendous improvement over a serial implementation. On a 8-core Intel i7 desktop, the results for 10000 adjoint iterations are compared in table 3. Note the speedup in this table is computed with respect to the auto-differentiated adjoint, which is why it is larger that 8×.

The ability to parallelize the adjoint solver is very useful especially since the adjoint solver is lacking implicit routines that allow for larger time steps. The "implicit routines" refer to the Diagonalized Alternating Direction Implicit (DADI) routines briefly mentioned in section 3.1.3 for the Euler solver. Initially in the project proposal the implicit routines were to be differentiated however from limitations of *Tapenade* and a lack of literature on the subject, the implicit routines were not implemented in the adjoint solver.

	Auto-Diff	By-hand	By-hand + OpenMP
Time (s)	126.3	120.5	15.1
Speedup	1.0	1.05	8.36

Table 3: Timing Results for 10000 Adjoint iterations (8-core CPU)

3.4 Single-Variable Design

A preliminary design case was chosen with a single variable representing the location of one hicks-henne bump along the surface of an airfoil. Typically doing a parametric sweep of the design space is very expensive since each point requires a solution of the Euler equations. For a single-variable case, however, this is not too computationally expensive and the results are shown in figure 11. This design problem was chosen specifically to highlight the possibility of local minima in the design space.

For the starting point shown in red in figure 11, the gradient was computed using the brute-force, AD adjoint, and by-hand adjoint methods. The results are summarized in table 4. As mentioned in the previous section, the gradient from the by-hand adjoint differs slightly

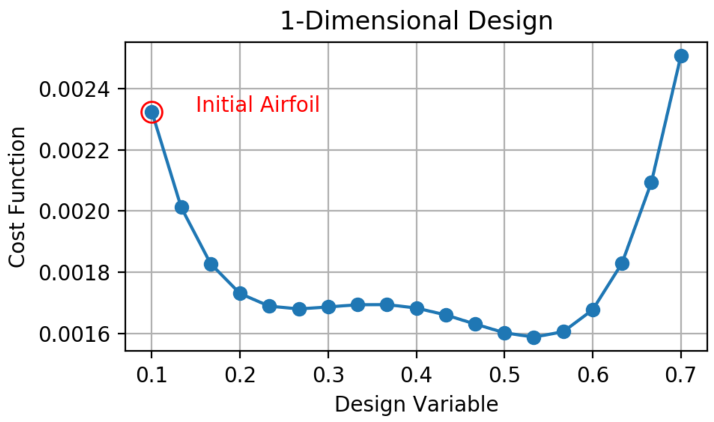


Figure 11: Example design space for 1 design-variable

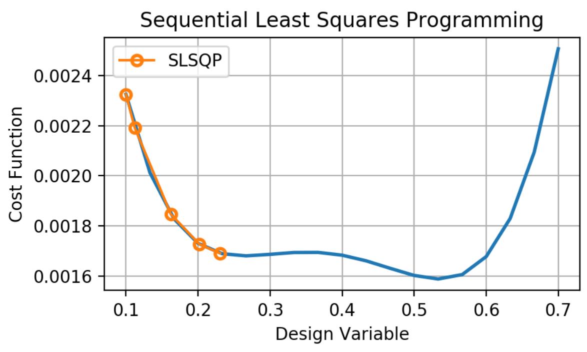
from the other method from the dissipation approximation.

Choosing Sequential Least SQuares Programming (SLSQP) as the optimization algorithm, the progression of the solution is shown in figure 12(a) and the Conjugate Gradient (CG) results in figure 12(b). These results are plotted along side the parametric sweep of the design space to show that both gradient-based methods converge to a local minimum and not a global minimum. This is a common issue encountered with gradient-based methods. The most common way to check for local minima is by altering the starting guess for the optimization. Figure 12(c) shows that with the starting guess with the bump closer to 70% along the airfoil chord, the optimization method converges to the global minimum.

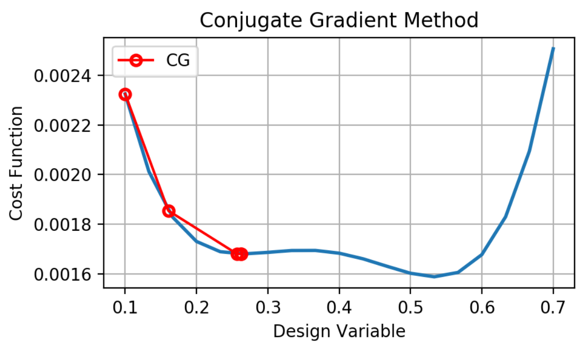
Figure 12(d) shows the convergence result plotted against design iteration number. The conjugate gradient method clearly shows faster convergence to the local minimum of the cost function. This result is slightly misleading, however, since the conjugate gradient method requires more that 1 function call (for bot the Euler and Adjoint routines) per design iteration. The run times and function-calls are tabulated in table 5 showing SLSQP as requiring significantly fewer function calls for this case. One possible explanation is that a line-search method within the CG algorithm is attempting to run multiple Euler and adjoint solutions to better estimate the step direction. An important factor to consider is that the 1-design variable case was chosen with a particularly poor desired pressure distribution in order to show the possibility of local minima. As shown in the following section, more feasible design cases will behave more expectedly.

	Brute-Force	Auto-Diff Adjoint	By-Hand Adjoint
Gradients	-0.0054687	-0.0054693	-0.0055054

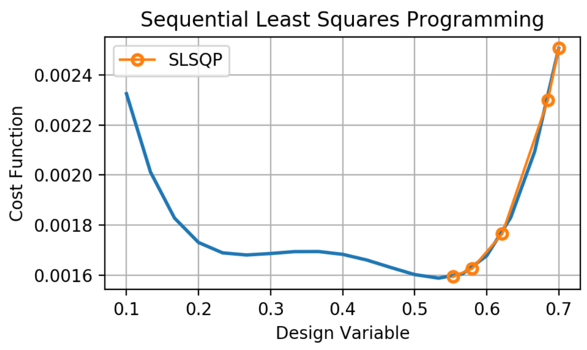
Table 4: Gradients at the initial guess using different methods



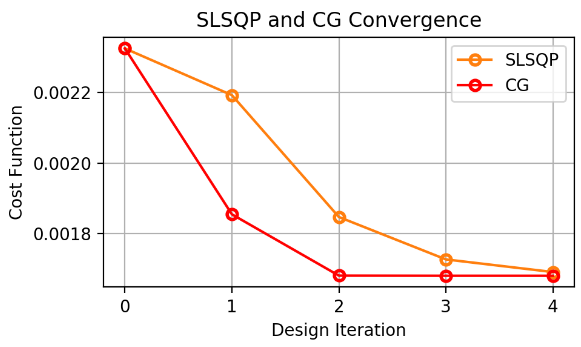
(a) SLSQP Method



(b) CG Method



(c) Alternate Starting Guess with SLSQP



(d) Convergence of both Methods

Figure 12: SLSQP and CG method overlayed with design space

	Design Iterations	Adjoint Calls	Euler Calls	Total Time (s)
SLSQP	4	5	5	209.7
CG	4	14	14	469.7

Table 5: Summary of timing results for SLSQP and CG methods for single design variable case

3.5 Six-Variable Design

Unlike with the single design variable case, a 6 design variable case is not easily visualized in two or even three-dimensions. A parametric sweep of the design space would also be extremely costly since a range of N points per variable results in N^6 executions of the Euler solver. This design problem therefore more clearly highlights the advantage of adjoint methods for shape optimization.

The six-design variable case fixes 3 bumps on both the upper and lower surface of the airfoil at the 25%, 50%, and 75% chord (along the airfoil from nose to tail). Only the amplitudes of the 6 bumps are altered, resulting in 6 design variables. The starting airfoil is a NACA0012, a commonly used symmetric airfoil. The NACA2312 is used to obtain the desired pressure distribution. This is a realistic case since the NACA2312 has camber of 2% located at 30% along the chord and has significantly better lifting properties. The initial and desired airfoils are shown as dotted lines in figure 16. Similarly the initial and desired pressure are shown in figure 14.

Table 6 shows the gradient of the cost function at the initial guess where the airfoil is a NACA0012. Again the gradients match very well between the AD adjoint and the brute-force methods but differ slightly for the by-hand adjoint from the dissipation approximation. The gradients are still more that accurate enough to for use in gradient-based optimization libraries to converge toward a minimum of the cost function.

	$\partial I/\partial \alpha_1$	$\partial I/\partial \alpha_2$	$\partial I/\partial \alpha_3$	$\partial I/\partial \alpha_4$	$\partial I/\partial \alpha_5$	$\partial I/\partial \alpha_6$
Brute-Force	-0.8495	-0.4387	-0.1935	-3.588	-0.8853	-0.1000
AD Adjoint	-0.8495	-0.4389	-0.1937	-3.588	-0.8855	-0.1003
Adjoint	-0.8549	-0.4365	-0.1851	-3.635	-0.8902	-0.0946

Table 6: Adjoint and brute-force sensitivity comparison

Figure 13 shows the convergence of 5 iterations of the SLSQP and CG algorithms. Unlike with the one-variable case, the six-variable convergence looks almost indistinguishable between both algorithms. Also unlike the one-variable case, the 5 design iterations required 6 function calls each of the Euler and adjoint solvers for both SLSQP and CG methods. This behavior is unexpected and will require further investigation to explain. For this project, because the results are so similar, only SLSQP results for the airfoil and pressure distribution are shown.

Using a desired pressure distribution in the cost function:

$$I_c(\alpha) = \sum_{i=0}^{N} \frac{1}{2}(P_i - P_{d,i})^2$$

is referred to as a reverse-design problem because the solution to the minimization is known, namely where $P = P_d$ and $I_c = 0$. The convergence of the cost function shown in figure 13 indicates the solution is approaching the correct answer as the cost function approaches 0. In theory, an infinite number of design variables for an infinite number of bumps along the surface of one airfoil can be used to exactly match the shape of another airfoil. The six bumps used in this case are unable to exactly match the solution airfoil geometry. The convergence therefore approaches zero but flattens out at a finite value representing the error of the resulting geometry.

The development of the airfoil geometry and pressure distributions over 5 SLSQP iterations are shown in figures 16 and 14. After 5 iterations, only the tail region of the airfoil shows large deviation from the desired distribution. A zoomed-in region of the plots is shown in figure 15. The closest bump location to the tail is at 75% of the chord however the deviation occurs closer to 93% along the airfoil. The 6 bumps used for this case are likely not enough to fully match the desired airfoil geometry however this can easily be remedied by adding more bumps along the airfoil.

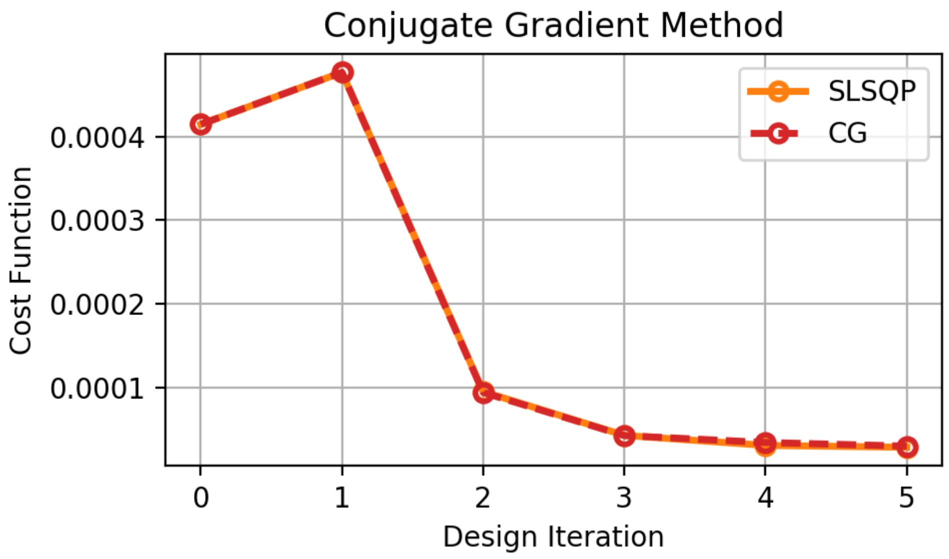
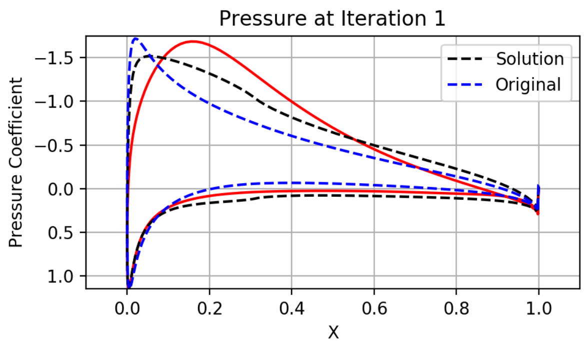
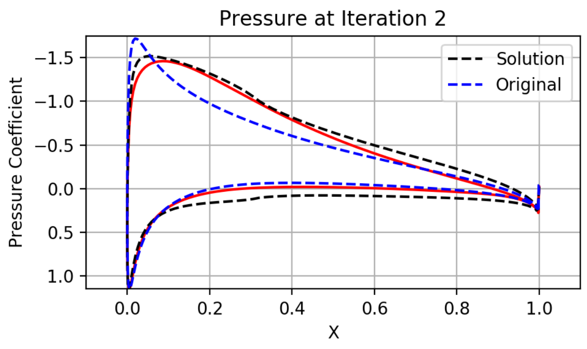


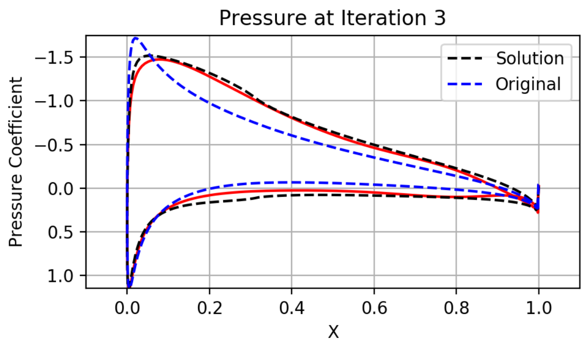
Figure 13: Cost function convergence



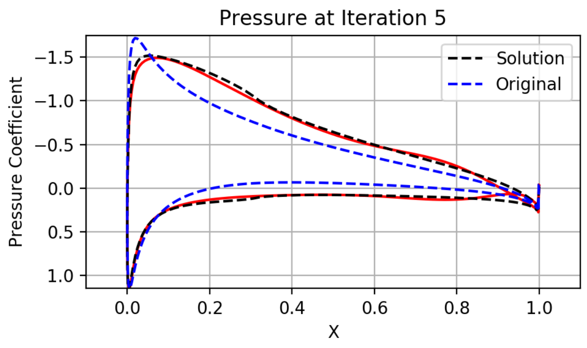
(a) Iteration 1



(b) Iteration 2

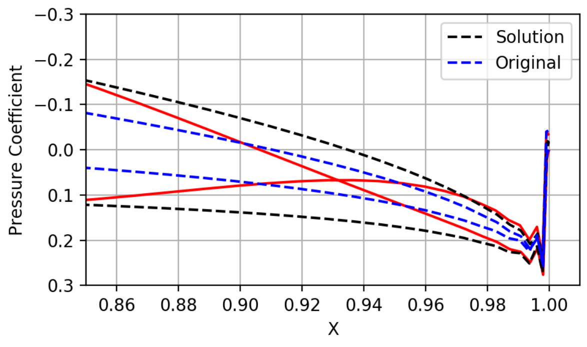


(c) Iteration 3

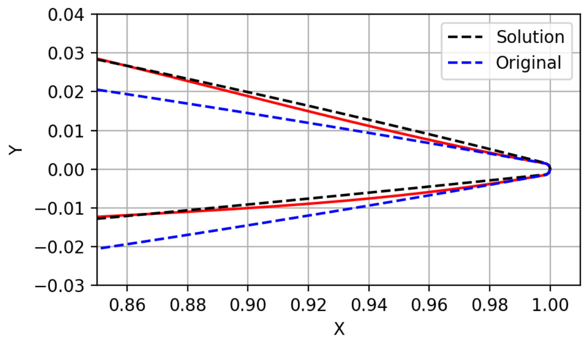


(d) Iteration 5

Figure 14: Pressure Development: SLSQP, and Cost Function Convergence

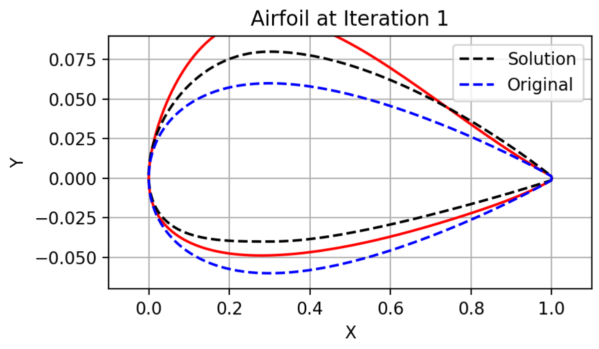


(a) Tail Pressure

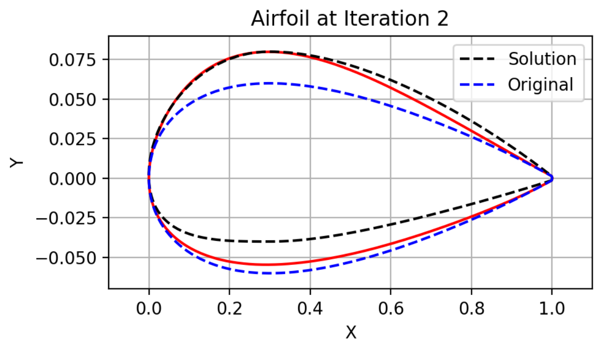


(b) Tail Airfoil

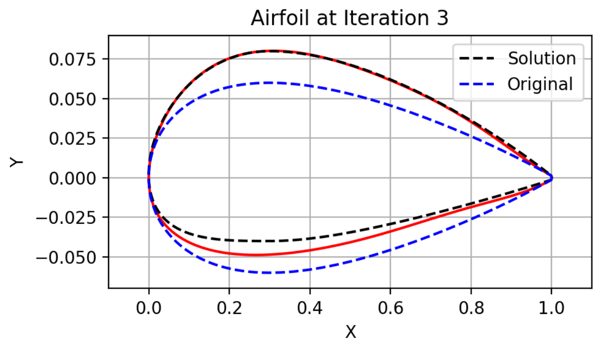
Figure 15: Tail Pressure and Airfoil geometry after 5 iterations



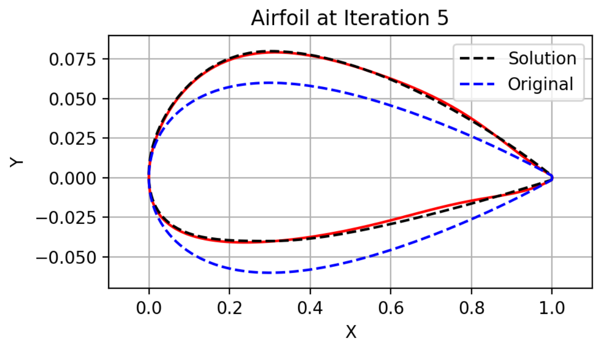
(a) Iteration 1



(b) Iteration 2



(c) Iteration 3



(d) Iteration 5

Figure 16: Airfoil Development: SLSQP

4 Conclusions

The discrete adjoint-Euler equations were derived and implemented both by-hand and through auto-differentiation within a Python-C++ framework. The presented analysis has shown:

- Fast, two-dimensional elliptic grid generation in combination with a two-dimensional Euler solver accessible through Python provides a flexible and extensible framework.
- Hicks-Henne bump functions are useful to perturb airfoil geometries.
- *Tapenade* as auto-differentiation software can be used to quickly develop an adjoint-Euler code. *Tapenade* was written for Fortran and can be challenging to work with in C/C++.
- Derivation and implementation of the discrete adjoint terms is not mathematically difficult but involves a large number of terms and bug-prone code. Certain simplifications can be made, for example in the dissipation terms, to reduce complexity.

- Having full control over memory and operations in the by-hand code allows parallelization through OpenMP.
- Computed gradients from both the auto-differentiated and by-hand adjoint solvers match with brute-force gradients computed directly from a finite difference of Euler solutions.
- From within Python, the Euler and adjoint routines provide the cost function and gradient respectively for use with the SciPy optimization library. This allows easy access to optimization methods such as sequential least squares programming (SLSQP) or conjugate gradient.
- The conjugate gradient method demonstrates better convergence than the SLSQP method for a one-design variable airfoil shape optimization case. SLSQP takes less wall-clock time because it required less function calls.
- For a six-design variable case, SLSQP and the conjugate gradient method converge equally well and require the same number of function calls. The cost function is successfully reduced and both the airfoil shape and pressure distribution closely match the known solution.

4.1 Milestones

A summary of the accomplished milestones is shown in table 7. Modifications from the original dates or achievements are crossed out and replaced with actual dates or altered milestones. The right-most column indicates the milestone was completed on the indicated date.

Milestone	Date
Functioning airfoil perturbation function in combination with mesh generation and 2D Euler Solver.	Late Oct
Functioning brute-force method for sensitivity of Pressure cost function to airfoil perturbation variables.	Early Nov
Auto-differentiation of Euler CFD solver.	Late Nov
Validate auto-diff and brute-force method for simple reverse-design perturbations.	Mid Dec
Hand-coded explicit discrete adjoint solver.	~~Mid Feb~~ March 6
~~Implicit routine~~ OpenMP Acceleration for discrete adjoint solver.	~~March~~ Early April
Validate discrete adjoint solver against auto-diff and brute-force methods.	~~March~~ Mid April
Test discrete adjoint solver with full reverse-design cases.	~~Mid April~~ Early May

Table 7: Milestone Accomplishments

Note: the initial project proposal stated subsonic and transonic cases would be tested however only subsonic cases were actually run. This is because the dissipation function was simplified to the scalar dissipation routine recommended from literature [1]. Without proper up-winding or flux splitting, the scalar dissipation will very poorly resolve shocks that appear in transonic cases.

4.2 Deliverables

The proposal for this project set forth the following deliverables:

- Bitbucket GIT Repository of
  - Euler, Grid-Generation, and Airfoil perturbation code in Python Framework
  - Auto-differentiated Adjoint Code
  - Hand-differentiated Adjoint Code
- Equations of hand-derived Adjoint relations for flux and dissipation terms.
- Sample run files for Euler-only, Adjoint-only, and full design case.

- Results from a full airfoil reverse design case (pressure matching NACA2312)

All of these can be found either in this report and as part of the Bitbucket repository at `https://bitbucket.org/djude/amsc663-664/src`. The repository includes a history of all commits made for the project with timestamps as well as the entire code framework of C, C++, and python files. The only items missing from the repository are headers required by *Tapenade* to auto-differentiate the code.

References

[1] S. Nadarajah, The Discrete Adjoint Approach to Aerodynamic Shape Optimization. PhD thesis, Stanford University Department of Aeronautics and Astronautics, 2003.

[2] J. Blazek, Computational Fluid Dynamics: Principles and Applications (Second Edition). Oxford: Elsevier Science, second edition ed., 2005.

[3] P. Roe, "Approximate riemann solvers, parameter vectors, and difference schemes," Journal of Computational Physics, vol. 43, no. 2, pp. 357 – 372, 1981.

[4] T. H. Pulliam and J. L. Steger, "Implicit finite-difference simulations of three-dimensional compressible flow," AIAA Journal, vol. 18, pp. 159–167, Feb 1980.

[5] M. Giles and N. Pierce, "An Introduntion to the Adjoint Approach to Design," Flow, Turbulance and Combustion, vol. 65, no. 3, pp. 393–415, 2000.

[6] M. B. Giles, D. P. Ghate, and M. C. Duta, "Using Automatic Differentiation for Adjoint CFD Code Development," Post SAROD Workshop, 2005.

[7] J.-D. Müller and P. Cusdin, "On the performance of discrete adjoint CFD codes using automatic differentiation," International Journal for Numerical Methods in Fluids, vol. 47, no. 8-9, pp. 939–945, 2005.

[8] L. Hascoët and V. Pascual, "TAPENADE 2.1 user's guide," 2004.

[9] M. B. Giles, D. P. Ghate, and M. C. Duta, "Using Automatic Differentiation for Adjoint CFD Code Development," Post SAROD Workshop, 2005.

[10] P. E. Gill, W. Murray, M. A. Saunders, and E. Wong, "User's guide for SNOPT 7.6: Software for large-scale nonlinear programming," Center for Computational Mathematics Report CCoM 17-1, Department of Mathematics, University of California, San Diego, La Jolla, CA, 2017.

[11] D. Kraft, A software package for sequential quadratic programming. Technical Report DFVLR-FB 88-28, Oberpfaffenhofen: Institut fur Dynamik der Flugsysteme, July 1988.

[12] J. Nocedal and S. J. Wright, Numerical optimization. Springer series in operations research and financial engineering., New York: Springer, 2006.

[13] R. HICKS and P. HENNE, Wing design by numerical optimization. Aircraft Design and Technology Meeting, American Institute of Aeronautics and Astronautics, Aug 1977.

[14] J. Steger and R. Sorenson, “Automatic mesh-point clustering near a boundary in grid generation with elliptic partial differential equations,” Journal of Computational Physics, vol. 33, no. 3, pp. 405 – 410, 1979.

[15] D. Jude, “ENAE 685: Assignment 4.” Private Communication, 2015.