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

Dylan Jude

December 15, 2016

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.

The same way we can mathematically define a cost function, we can also define design variables which control the shape of the given airfoil. As an example, figure 1 shows an airfoil whose general shape can be altered using two variables α and c.

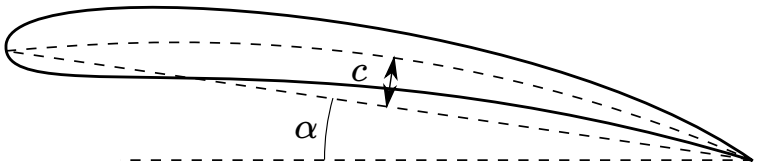

Figure 1: Example Airfoil Design Variables

1.1 Choosing a Cost Function

Assuming we already have an 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, we obtain the flow solution using CFD, and from the flow solution we can obtain a pressure at every point on the airfoil. A simple cost function typically chosen in airfoil shape optimization compares the current pressure distribution 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.

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

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

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:

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

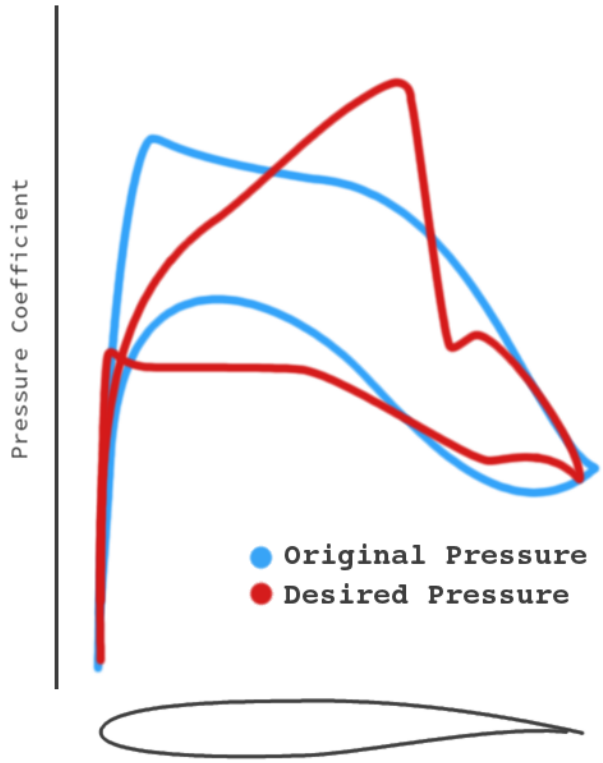

Figure 2: Comparison of a pressure distribution with the desired distribution [1]

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, 3-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. In two dimensions, 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 $u_1 \text{and} u_2$, 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 pressure-work done on the surface of the control volume 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 explicitely using a first-order difference in both space and time. In one-spacial dimension, the explicit discretization for finding time $n + 1$ is

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

where D_{Roe} is artificial dissipation from the Roe-flux difference splitting scheme [3].

The Euler equations are also discretized implicitely in time

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

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 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.

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

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

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

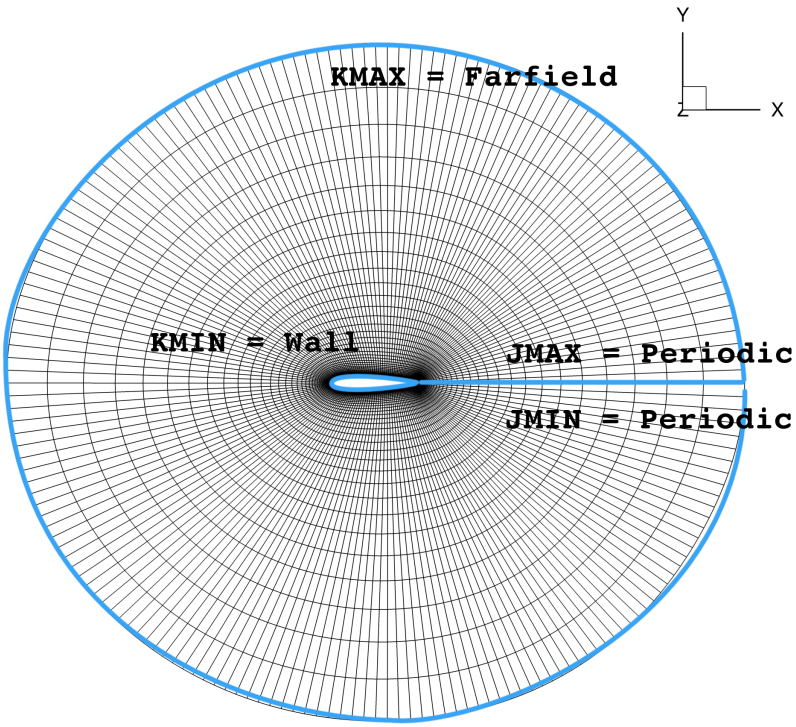

Figure 3: O-Mesh Topology with coordinates `j` and `k`

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:

$$\int_D \left[\frac{\partial \psi}{\partial \xi_i}\frac{\partial f_i}{\partial q}\right] dD = 0$$
$$\frac{\partial \psi}{\partial \xi_i}\frac{\partial f_i}{\partial q} = 0$$

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:

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

2.3 Auto-Differentiation

Solving the adjoint equation is essentially a differentiation of the flow residual R and many research groups have shown successful implementations of adjoint methods using auto-differentiation [6]. These methods tend to be much 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:

$$\frac{\partial I_c}{\partial \alpha} = \frac{\partial I_c}{\partial Q}\frac{\partial Q}{\partial X}\frac{\partial X}{\partial \alpha} \tag{23}$$

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:

$$\left(\frac{\partial I_c}{\partial \alpha}\right)^T = \left(\frac{\partial X}{\partial \alpha}\right)^T \left(\frac{\partial Q}{\partial X}\right)^T \left(\frac{\partial I_c}{\partial Q}\right)^T \tag{24}$$

Using "bar" notation to denote the partial derivative of the cost function I_c with respect to a variable, equation (24) 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:

$$\dot{I}_c = \frac{\partial I_c}{\partial \alpha} = \bar{Q}^T \dot{Q} = \bar{X}^T \dot{X} = \bar{\alpha}^T \dot{\alpha} \tag{25}$$

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.

2.4 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, we are left with 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 the direction of steepest descent. Though there are other optimization methods that would likely result in faster convergence to the minimum of the cost function, using the method of steepest descent is the simplest method and has shown to work well for simple airfoil optimization [10].

$$\alpha_i^{n+1} = \alpha_i^n - \lambda \frac{\partial I}{\partial \alpha_i} \tag{26}$$

2.5 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"[11]. These bump functions are sinesoidal perturbations applied at different locations along the airfoil. A commonly used form is

$$b(x) = a\left[sin\left(\pi x^{\frac{log(0.5)}{log(t_1)}}\right)\right]^{t_2}, \quad \text{for } 0 \leq x \leq 1 \tag{27}$$

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 4 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.

3 Progress

The project is currently on-schedule with the milestones set forth at the beginning of the semester. Table 1 shows the completed tasks as well as the remaining tasks for the second semester. The green checkmarks represent completed tasks whereas the gray checkmark for Mid-December is an almost completed but ongoing task.

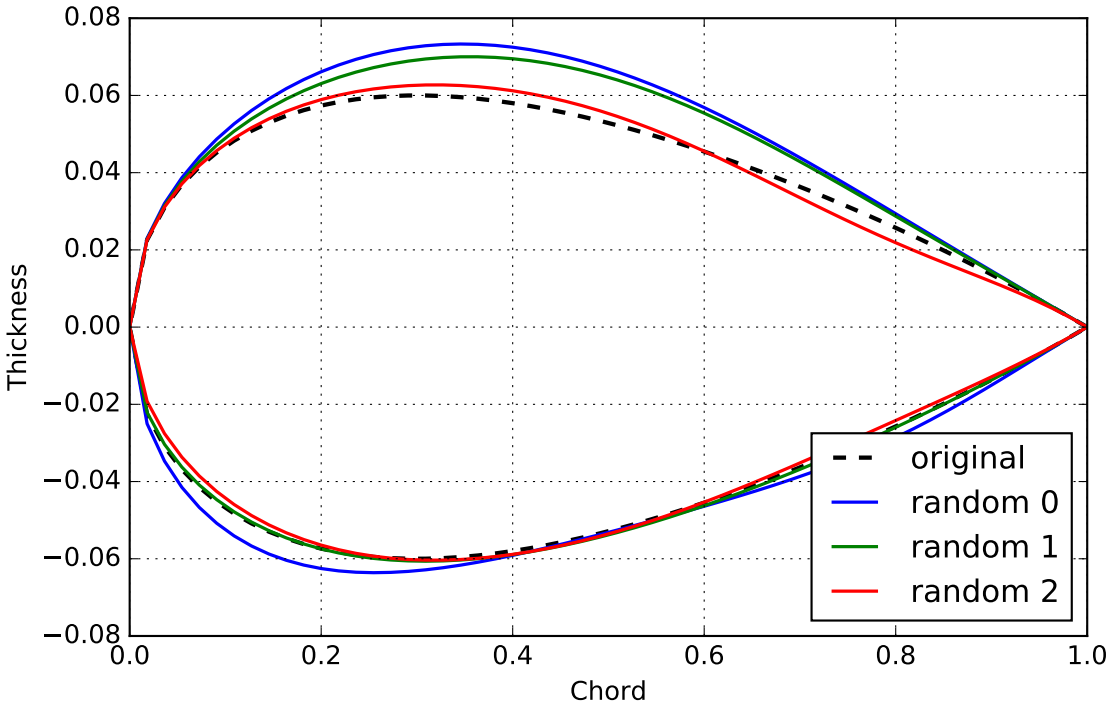

Figure 4: Hicks-Henne Bump functions with random variable perturbations

3.1 Milestone 1: Framework Preparation

The first milestone was to obtain a "functioning airfoil perturbation function in combination with mesh generation and 2D Euler Solver". This will be presented in individual subsections for the mesh generation, 2D Euler code, and perturbation routine. All codes are written in C/C++ and wrapped in Python using the Boost-Python libraries.

3.1.1 Mesh Generation

2-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 [12]. A mesh generation code had previously been developed for CFD 2 (ENAE

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 Jan	
Implicit routine for discrete adjoint solver.	Early Feb	
Validate discrete adjoint solver against auto-diff and brute-force methods.	Late Feb	
Test discrete adjoint solver with full reverse-design cases.	Mid Mar	

Table 1: Milestone Completion

685) but instead using the Laplace equation ($P_s = Q_s = 0$)[13]. 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 [12].

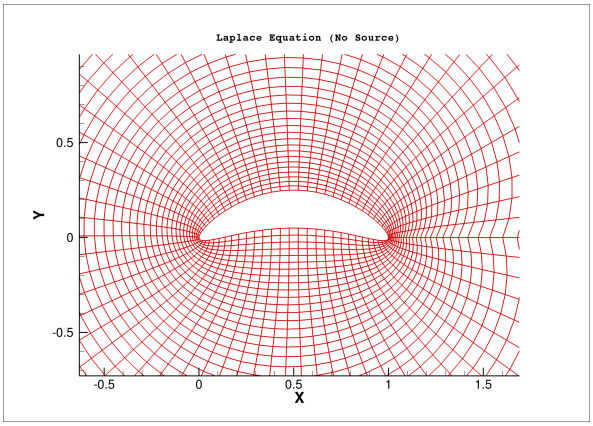

Figure 5: Grid generation without source terms

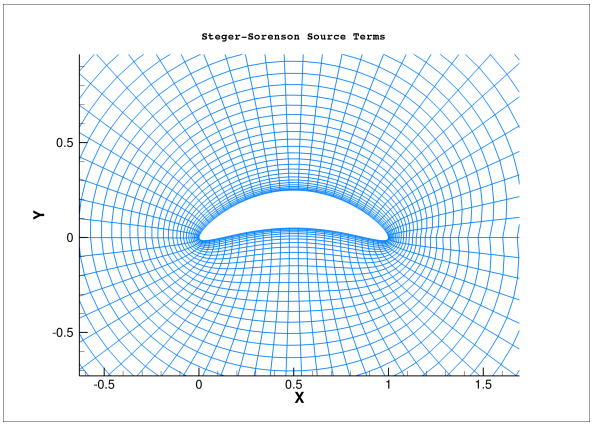

Figure 6: Grid generation with Steger-Sorenson source terms

Figures 5 and 6 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 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 7 shows the solution pressure contours from the 2D flow solver and figure 8 shows the comparison with the trusted 3D reference code. As expected, the results are exactly the same. Figure 9 shows the convergence of the 2D Euler solver, which converges to machine zero within 1500 iterations.

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.

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

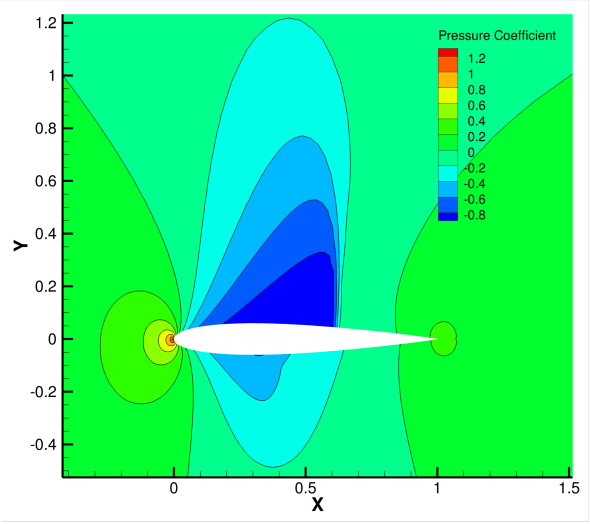

Figure 7: Example solution pressure contours

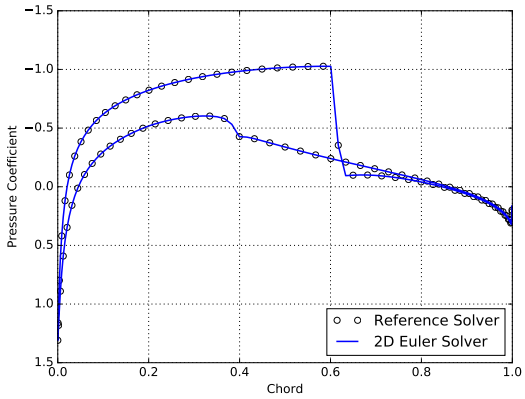

Figure 8: Comparison of pressure distribution with original in-house 3D solver

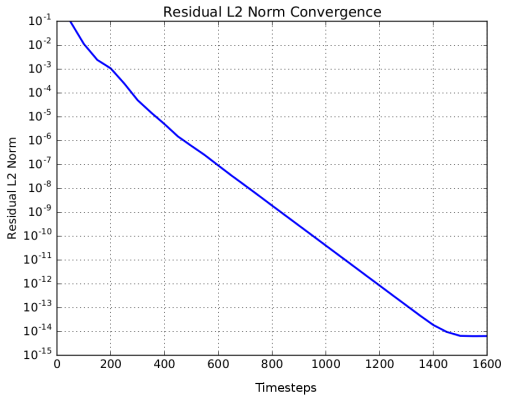

Figure 9: L2 norm convergence of the Euler solver residual

3.1.3 Airfoil Perturbation

Airfoil perturbations are done using the Hicks-Henne "bump" function described in an earlier section, governed by equation (27). 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 10 shows the

original NACA0012 airfoil, and figure 11 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)
```

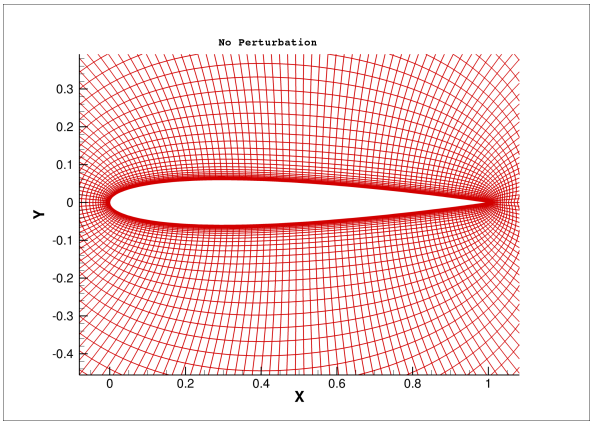

Figure 10: Original NACA0012 grid

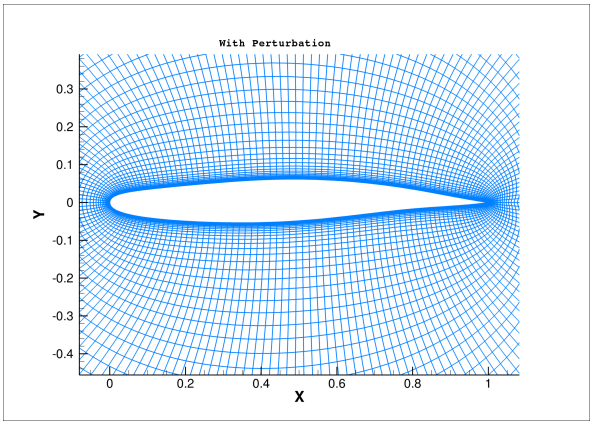

Figure 11: Perturbed grid with 6 bumps

3.2 Milestone 2: Brute-Force Sensitivities

The developed framework for the airfoil perturbation, grid generation, and Euler solver can be used to find sensitivities of a cost function to a set of design variables using a brute-force, finite difference approach. Recall equation (4) for the calculating sensitivity of a cost function using a brute-force approach:

$$\frac{\partial I_c}{\partial \alpha_i} \approx \frac{I_c(\alpha_i + \Delta\alpha_i) - I_c(\alpha_i)}{\Delta\alpha_i}$$

Similar to the Hicks-Henne perturbations described in the previous section, define 6 bumps over a NACA0012 airfoil with fixed widths and locations at 25%, 50%, and 75% along the airfoil on both the upper and lower surfaces. This leaves 6 design variables for the amplitudes of the 6 bumps. After obtaining an initial Euler solution for a starting airfoil, the sensitivities with respect to each variable can be obtained by perturbing each of the 6 design variables and using the finite-difference formula above. The finite-difference sensitivities were conducted for

all 6 variables at a range of different variable perturbations $\Delta\alpha$. The results are summarized in the top half table 2.

For a cost function that does not vary linearly as a function of the variables, the finite-difference sensitivity would be most accurate as the $lim(\Delta\alpha) \to 0$. This explains why in table 2 the sensitivities are not constant for $\Delta\alpha > 10^{-6}$. On a machine where double floating point precision allows for 16 decimal digits after the decimal point, numbers cannot be compare with differences on the order of 10^{-16} due to round off error. This is why the sensitivities also do not remain constant for values of $\Delta\alpha < 10^{-12}$. In the range of $10^{-12} \leq \Delta\alpha \leq 10^{-6}$, the sensitivities remain approximately constant, though without an analytical solution it is difficult to know which is most accurate.

3.3 Milestone 3: Auto-Differentiation

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. It was shown in section 2.2 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:

```
1  while(n < nsteps){
2
3    boundary_conditions(Q, X);
4
5    flux(Q, R, X);
6
7    for(i=0; i<all_pts; i++)
8      Q[i] = Q[i] + R[i]*dt[i];
9
10   n++;
11 }
12 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:

```
1  Ib = 1.0
2  compute_cost_function_b(Q, Qb, I, Ib);
3  while(n < nsteps){
4
```

```
5     flux_b(Q, Qb, R, Rb, X, Xb);
6
7     boundary_conditions_b(Q, Qb, X, Xb);
8
9     for(i=0; i<all_pts; i++)
10      Rb[i] = Rb[i] + Qb[i]*dt[i];
11
12    n++;
13  }
```

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 2.2

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

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

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

With this auto-differentiated code, the same 6-design-variable problem from the last section but this time with sensitivities computed from the adjoint formulation. The results are shown in the bottom half of table 2. As shown in figure 12, the adjoint residual is dropping to zero while the cost sensitivity converges to a constant value. Only the explicit formulation of the Euler equations have been auto-differentiated so the timestep is restricted to being very small, which is why so many iterations were required to converge.

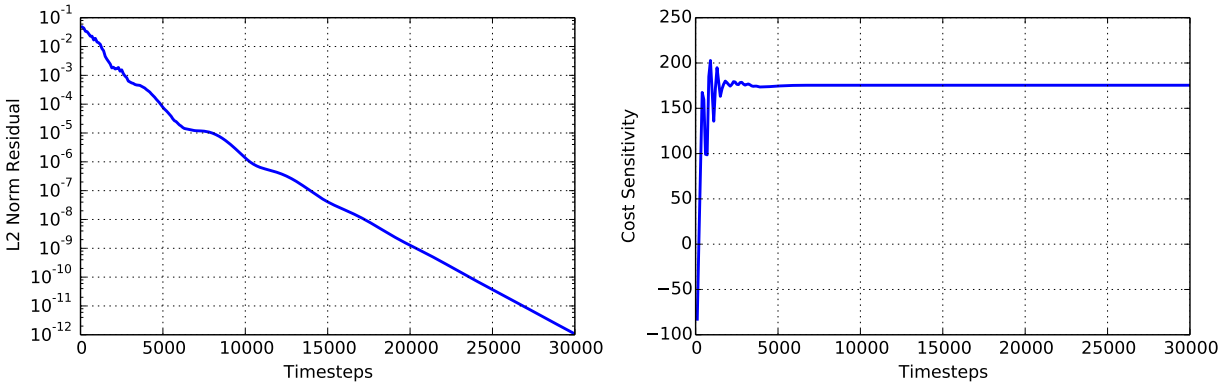

Figure 12: Adjoint residual and cost sensitivity convergence

	$\Delta\alpha$	$\partial I/\partial\alpha_1$	$\partial I/\partial\alpha_2$	$\partial I/\partial\alpha_3$	$\partial I/\partial\alpha_4$	$\partial I/\partial\alpha_5$	$\partial I/\partial\alpha_6$
Brute-Force	10^{-2}	4.858167	8.984251	14.729829	-15.220139	26.463971	45.337180
	10^{-4}	-12.122452	-7.537323	-19.113102	-132.017122	-8.441582	11.684732
	10^{-6}	-12.293774	-7.709047	-19.469728	-133.221209	-8.774178	11.363687
	10^{-8}	-12.295486	-7.710764	-19.473296	-133.233252	-8.777502	11.360479
	10^{-10}	-12.295447	-7.710790	-19.473206	-133.233318	-8.777504	11.360487
	10^{-12}	-12.296330	-7.704226	-19.469870	-133.227207	-8.774869	11.363132
	10^{-14}	-11.624035	-7.466249	-18.884893	-132.854838	-8.193445	11.818324
	10^{-16}	73.829831	67.723604	-15.543122	-147.659662	21.649348	46.629367
Adjoint	10^{-2}	-12.311956	-7.711442	-19.473012	-133.187438	-8.777613	11.360054
	10^{-4}	-12.295667	-7.710788	-19.473329	-133.232903	-8.777536	11.360442
	10^{-6}	-12.295506	-7.710782	-19.473332	-133.233369	-8.777535	11.360445
	10^{-8}	-12.295504	-7.710782	-19.473332	-133.233374	-8.777535	11.360445
	10^{-10}	-12.295500	-7.710783	-19.473334	-133.233371	-8.777539	11.360443
	10^{-12}	-12.295245	-7.710735	-19.473840	-133.232466	-8.777156	11.361694
	10^{-14}	-12.349382	-7.749904	-19.402156	-133.237464	-8.795003	11.371354
	10^{-16}	-13.273199	-7.619278	-23.770912	-139.868916	-5.445232	10.199848

Table 2: Adjoint and brute-force sensitivity comparison

3.4 Analysis of Results

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:

$$\frac{\partial X}{\partial \alpha_i} = \frac{X(\alpha_i + \Delta\alpha_i) - X(\alpha_i)}{\Delta\alpha_i} - \frac{\Delta\alpha}{2}\frac{\partial^2 X}{\partial \alpha^2} + O(\Delta\alpha^2) \tag{28}$$

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

$$\frac{\partial I_c}{\partial \alpha_i} = \frac{I_c(\alpha_i + \Delta\alpha_i) - I_c(\alpha_i)}{\Delta\alpha_i} - \frac{\Delta\alpha}{2}\frac{\partial^2 I_c}{\partial \alpha^2} + O(\Delta\alpha^2) \tag{29}$$

From table 2, at a glance it appears the adjoint method gives better sensitivities than the brute-force approach becuase 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:

$$Error_{\text{brute-force}} \sim \frac{\Delta\alpha}{2}\frac{\partial^2 I_c}{\partial \alpha^2} \tag{30}$$

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

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

$$Error_{\text{adjoint}} \sim \bar{X}^T \frac{\Delta\alpha}{2}\frac{\partial^2 X}{\partial \alpha^2} \tag{31}$$

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 3, 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 3: Approximate truncation error for adjoint and brute-force method

4 Future Work

The previous section presented progress made over the last four months, which has remained on track with the schedule set forth at the beginning of the sememster. The remaining milestones for the project are outlined in table 4. This table includes the milestone for validation of the auto-differentiation because, although this step is for the most part complete, there are a few minor details which could be improved upon. One example is the auto-differentiation of the implicit routines of the Euler solver. This would accelerate convergence of the Adjoint solver, which currently takes a few tens of thousands of iterations to converge.

Validate auto-diff and brute-force method for simple reverse-design perturbations.	Mid Dec	✓
Hand-coded explicit discrete adjoint solver.	Mid Jan	
Implicit routine for discrete adjoint solver.	Early Feb	
Validate discrete adjoint solver against auto-diff and brute-force methods.	Late Feb	
Test discrete adjoint solver with full reverse-design cases.	Mid Mar	

Table 4: Future Milestones

Starting in January, the next goal will be to code a by-hand implementation of the adjoint Euler equations. Since the auto-differentiation is complete, it can conveniently be used as a comparison to validate the by-hand version. Next semester will also be when the adjoint code is applied to an actual design problem, instead of merely analyzing cost function sensitivities. This will depend upon implementing a path-of-steepest descent for gradient-based optimization.

5 Deliverables

1. Airfoil perturbation and grid-generation code.

2. Auto-differentiated Euler CFD code.

3. Results for auto-diff and finite-difference tests on simple reverse-design perturbation problem.

4. Discrete adjoint solver code

5. Results for adjoint code validation with finite-difference and auto-diff tests

6. Results for a full reverse-design cycle test

7. Report on achievements and results

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] S. Nadarajah and A. Jameson, "A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization," *Aerospace Sciences Meetings*, 2000.

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

[12] 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.

[13] D. Jude, "ENAE 685: Assignment 4." Private Communication, 2015.