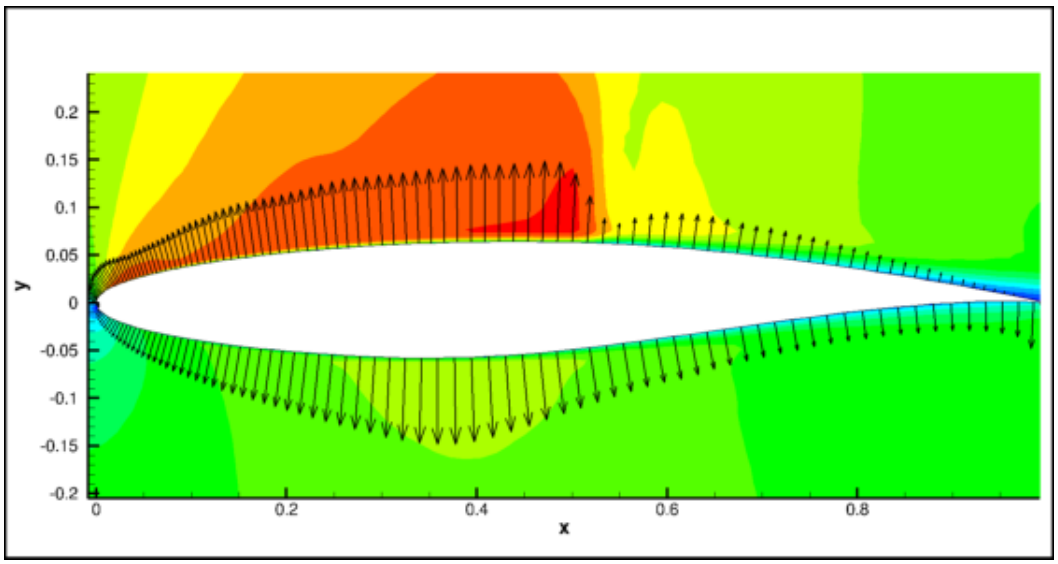
Gradient Calculation using Algorithmic Differentiation

With Application to an Open-Source CFD-Solver

Tim Albring

September 29, 2013



Contents

1 Algorithmic Differentiation

In recent numerical applications in engineering and sciences the handling of gradients has become an important task. Especially for optimization problems, where gradient-based methods perform much better than gradient-free methods, the need for an efficient evaluation of gradients has grown in the past. Although the performance of supercomputers is increasing and finite difference (FD) methods may be affordable for smaller problems with few unknowns, for most of the problems with many unknowns it is still not an option.

Algorithmic Differentiation or also called *automatic differentiation* is a way to calculate the derivative of a function by means of the transformation of the underlying program which calculates the numerical values of this function. As distinguished from symbolic differentiation an explicit expression for the derivative is never formed. An advantage over FD is that no truncation errors are present, thus the numerical value can be determined up to machine accuracy. Furthermore it is possible to get the whole or part of the Jacobian with less effort.

1.1 Forward Propagation

Suppose we have a function $f \in C^1 : U \subset \mathbb{R}^n \to \mathbb{R}^m$ and the expression

$$y = f(x), \tag{1}$$

that is, $x \in U, y \in \mathbb{R}^m$. The evaluation of the function f can be represented as a sequence of l elementary functions φ_i and intermediate values v_i like it is shown in Table 1. In computer programs those functions may be for example intrinsic functions provided by the compiler. The

v_{i-n}	$= x_i,$	$i =$	$1 \dots n$
v_i	$= \varphi_i(v_j)_{j \prec i},$	$i =$	$1 \dots l$
y_{m-i}	$= v_{l-i},$	$i =$	$m-1 \dots 0$

Table 1: General Evaluation Procedure

precedence relation $j \prec i$ means that i depends directly on j. Applying the chain rule to the v_i in Table 1 results in

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j \tag{2}$$

with the abbreviation $u_i = (v_j)_{j \prec i} \in \mathbb{R}^{n_i}$, i.e. the vector u_i is the concatenation of the v_j on which φ_i depends. For the derivative of the function f then holds

$$\dot{y}_{m-i} := \dot{v}_{l-i} \equiv \frac{\partial f(x)}{\partial x_{m-i}}, \quad i = m-1 \dots 0 \tag{3}$$

That means, as long as the derivatives of the elementary functions are known a straightforward evaluation yields the derivative of f. Note that for building the derivative of the General Evaluation Procedure shown in Table 1 we need the derivatives of the values $v_{i-n}, i = 1 \dots n$ as an input, that is, values for $\dot{x}_i$. By formally applying the chain rule to the expression (1) we get

$$\dot{y} := \frac{df}{dx}(x) \cdot \dot{x} = \sum_{i=1}^{m} \frac{\partial f(x)}{\partial x_i} \dot{x}_i \tag{4}$$

Thus we can get the derivative $\frac{\partial f(x)}{\partial x_i} \in \mathbb{R}^m$ by setting $\dot{x}_i = 1$ and $\dot{x}_j = 0, j \neq i, j = 1, \ldots, n$. If equations (2) and (3) are combined with the evaluation trace one gets the Forward Propagation of Gradients given in Table 2.

$[v_{i-n}, \dot{v}_{i-n}]$	$= [x_i, \dot{x}_i]$	$i =$	$1 \ldots n$
$[v_i, \dot{v}_i]$	$= [\varphi_i(v_j)_{j \prec i}, \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j]$	$i =$	$1 \ldots l$
$[y_{m-i}, \dot{y}_{m-i}]$	$= [v_{l-i}, \dot{v}_{l-i}]$	$i =$	$m-1 \ldots 0$

Table 2: Forward Propagation of Gradients

1.2 Reverse Propagation

For the forward mode the derivatives are propagated in the same direction as the corresponding elemental function values, i.e. we ask how a infinitesimal change in the input values propagates through the evaluation trace and affects the output. But we could also ask the other way round: How sensitive are the output values to a change in the input values? The latter is the basis for the reverse propagation. There are many ways in which the reverse mode can be introduced, but in general it can be thought of as the backward application of the chain rule. A sophisticated derivation can be found in Griewank and Walther [1]. Here only the main results needed for the application are described.

For the application of the chain rule to the evaluation of the function f (Table 1) it is useful to introduce the state transformation Φ_i for each elemental function φ_i:

$$\mathbf{v}_i = \Phi_i(\mathbf{v}_{i-1}), \quad \Phi_i : \mathbb{R}^{n+l} \to \mathbb{R}^{n+l} \tag{5}$$

with

$$\mathbf{v}_i := (v_{1-n}, \ldots, v_i, 0, \ldots, 0)^T \in \mathbb{R}^{n+l}$$

In other words Φ_i sets v_i to $\varphi_i(v_j)_{j \prec i}$ and keeps all other components v_j for $i \neq j$ unchanged. We can then write the expression (1) as the composition

$$y = Q_m \Phi_l(\Phi_{l-1}(\ldots(\Phi_1(P_n^T x)))) \tag{6}$$

where $P_n \in \mathbb{R}^{n \times (n+l)}$ and $Q_m \in \mathbb{R}^{m \times (n+l)}$ are the matrices that project an $(n+l)$-vector onto its first n and last m components, respectively. The derivatives of the state transformations $\nabla \Phi_i$ can be explicitly calculated and written as

$$A_i := \nabla \Phi_i = I + e_{n+i}(\nabla \varphi_i(u_i) - e_{n+i})^T \in \mathbb{R}^{(n+l) \times (n+l)} \tag{7}$$

Now equation (6) can be differentiated using the chain rule:

$$\dot{y} = Q_m A_l A_{l-1} \ldots A_2 A_1 P_n^T \dot{x} \tag{8}$$

Thus, the jacobian of f can be written as

$$\frac{df}{dx}(x) = Q_m A_l A_{l-1} \ldots A_2 A_1 P_n^T \tag{9}$$

By transposing the product we obtain the adjoint relation

$$\bar{x} = P_n A_1^T A_2^T \dots A_{l-1}^T A_l^T Q_m^T \bar{y} = \left(\frac{df}{dx}\right)^T \bar{y} \tag{10}$$

with $\bar{x} \in \mathbb{R}^n, \bar{y} \in \mathbb{R}^m$ and the identity

$$\bar{y}^T \dot{y} = \bar{x}^T \dot{x}.$$

Let us look at equation (10) in more detail. Suppose we have a vector $\bar{\mathbf{v}}_i$ of adjoint quantities $\bar{v}_j, 1-n \leq j \leq l$ such that

$$\bar{\mathbf{v}}_i = A_i^T A_{i-1}^T \dots A_{l-1}^T A_l^T Q_m^T \bar{y} = A_i^T \bar{\mathbf{v}}_{i-1}.$$

With definition (7) this becomes

$$\bar{\mathbf{v}}_i = \bar{\mathbf{v}}_{i-1} + (\nabla \varphi_i(u_i) - e_{n+i}) e_{n+i}^T \bar{\mathbf{v}}_{i-1}.$$

Now we can analyse what happens to the adjoint quantity $(\bar{\mathbf{v}}_i)_j = \bar{v}_j$ if we consider the i-th elemental function φ_i for $i = l, \dots, 1$:

- Since $(\nabla \varphi_i(u_i))_j = 0$ and $e_{n+i} e_{n+i}^T \bar{\mathbf{v}}_{i-1} = 0$ for $i \neq j \not\prec i$, $\bar{v}_j$ is left unchanged if φ_i does not depend on v_j.
- If $i \neq j$ and $j \prec i$ then $(\nabla \varphi_i(u_i))_j = \frac{\partial \varphi_i}{\partial v_j}$ and $e_{n+i}^T \bar{\mathbf{v}}_{i-1} = \bar{v}_i$, thus, $\bar{v}_j$ is incremented by $\bar{v}_i \frac{\partial \varphi_i}{\partial v_j}$.
- $\bar{v}_i$ is set to zero.

With this information it is possible to rewrite the adjoint relation (10) as an evaluation procedure like it was done for the forward evaluation. Since the matrix-vector products in equation (10) are calculated for $i = l, l-1, \dots, 1$ we have to go backward, or reverse, through the sequence of elementary functions in Table 1. Since the intermediate values v_i are needed they have to be computed first by evaluating the sequence of elemental functions. Summarizing this yields the Reverse Propagation of Gradients shown in table 3 [see also 1].

v_{i-n}	$= x_i,$	$i =$	$1 \dots n$
v_i	$= \varphi_i(v_j)_{j \prec i},$	$i =$	$1 \dots l$
y_{m-i}	$= v_{l-i},$	$i =$	$m-1 \dots 0$
$\bar{v}_{l-i}$	$= \bar{y}_{m-i}$	$i =$	$0 \dots m-1$
$\bar{v}_j$	$= \bar{v}_j + \bar{v}_i \frac{\partial}{\partial v_j} \varphi_i(u_i), j \prec i,$	$i =$	$l \dots 1$
$\bar{x}_i$	$= \bar{v}_{i-n},$	$i =$	$n \dots 1$

Table 3: Reverse Propagation of Gradients

As an input we need the vector $\bar{y}$. At the end we have $\bar{x}$, that is, the matrix-vector product $\bar{x} = \nabla f(x)^T \bar{y}$.

1.3 Implementation and Software

The generation of the Tangent Recursion or of the Adjoint Recursion for the forward and reverse propagation, respectively, is of pure mechanical fashion provided that the derivatives of the elemental functions are known. To incorporate this fact into an actual application there exist two basic computer science concepts, namely *Source Code Transformation* and *Operator Overloading*. For the first method the source code of the function to be differentiated is augmented by derivative assignments according to section 1.1 or section 1.2. This can be done either by hand or with a preprocessor.

Implementation of a Forward Propagation Tool

Since for this work the second approach is used it will be explained in a slightly more detailed way. First, consider the forward mode. In order to calculate the derivative of a function we define a new data type or class that contains the numerical value of v_i and $\dot{v}_i$. A very basic implementation of an `fdouble`-class is shown in the C++ listing 1. Of course in a real implementation the data fields `value` and `dot_value` would be declared as `private`. Then overloaded versions of the arithmetic operations for scalars are implemented. These overloaded operations must manipulate $\dot{v}_i$ according to the chain rule. As a last step any floating point program variable whose derivatives are needed is redeclared to be of the this new type of class. Then the gradient information of a function is generated along with the execution of this function if the derivatives of the inputs are properly initialized. In the C++ listing 2 the overloaded versions of the multiplication operator as well as of the sine function based on the `fdouble`-class are implemented exemplary. One particular advantage of this approach is that all of the AD-specific functions can be concealed in suitable header files without modifying the original code. Though there are additionally some changes that cannot be avoided in order to initialize the independent variables and their derivatives and extracting the derivative values of the dependent variables. This is done by calling functions that can set and return the `dot_value` data field of a `fdouble` variable.

```
1 class fdouble
2 {
3   double value;
4   double dot_value;
5 };
```

Listing 1: `fdouble`-Class Implementation for Forward Propagation

```
1  const fdouble operator*(const fdouble &a, const fdouble& b)
2  {
3      fdouble result;
4      result.value = a.value * b.value;
5      result.dot_value = a.dot_value*b.value + a.value*b.dot_value;
6      return result;
7  }
8
9  fdouble sin(fdouble a)
10 {
11     fdouble result;
12     result.value = sin(a.value);
13     result.dot_value = cos(a.value) * a.dot_value;
14     return result;
15 }
```

Listing 2: Overloaded Multiplication Operator and Sine Function

Implementation of a Reverse Propagation Tool

Now let us consider the reverse mode. Like for the forward mode we introduce a new data type. Though in this case it contains the numerical value and an identifier or index (see C++-listing 3 for `adouble`-Class implementation). During the execution of the evaluation procedure we build up an internal representation of the computation, which we call *tape*. The tape is essentially an array consisting of tape entries `tape_entry` (Listing 4). A tape entry holds the operation code encoded as an integer (`oc`), a function value v_i (`value`) and the corresponding value of $\bar{v}_i$ (`bar_value`). Furthermore it stores the values of the arguments of the specific operation (`arg1,arg2`). Additionally there is usually a numerical record to store the overwritten variables that is not shown here. Then, again, we define arithmetic operations of this new data type that correspond to the usual floating point operations. These operations calculate the floating point operations for v_i as usual, but as a side effect they create a tape entry and thus record themselves and their arguments on the tape. We can then define a routine `interpret_tape()` that reverses through the tape and calculates the adjoint variables $\bar{v}_i$ correctly according to the second half of Table 3. Just like in the forward implementation all floating point program variables must be redeclared to be of the new data type.

```
1  class adouble
2  {
3    double value;
4    int index;
5  };
```

Listing 3: `adouble`-Class Implementation for Reverse Propagation

```
1  class tape_entry
2  {
3    int oc;
4    double value;
5    double bar_value;
6    int arg1;
7    int arg2;
8  };
```

Listing 4: `tape_entry`-Class Implementation

```
1  const adouble operator*(const adouble &a, const adouble& b)
2  {
3    adouble c;
4    indexcount += 1;
5    c.index = indexcount;
6    tape_entry new_entry;
7    new_entry.oc = MULT;
8    new_entry.arg1 = a.index;
9    new_entry.arg2 = b.index;
10   c.value = a.value * b.value;
11   new_entry.value = c.value;
12   tape[indexcount - 1] = new_entry;
13   return c;
14 }
15
16 adouble sin(adouble a)
17 {
18   adouble b;
19   indexcount += 1;
20   b.index = indexcount;
21   tape_entry new_entry;
22   new_entry.oc = SIN;
23   new_entry.arg1 = a.index;
24   b.value = sin(a.value);
25   new_entry.value = b.value;
26   tape[indexcount] = new_entry;
27   return b;
28 }
```

Listing 5: Overloaded Multiplication Operator and Sine Function for Reverse Propagation

```
1  void interpret_tape()
2  {
3    for(int i = indexcount - 1; i >= 0; i--)
4    {
5      switch(tape[i].oc)
6      ...
7      case MULT:
8      {
9        tape[tape[i].arg1].bar_value += tape[tape[i].arg2].value*tape[i].bar_value;
10       tape[tape[i].arg2].bar_value += tape[tape[i].arg1].value*tape[i].bar_value;
11       break;
12     }
13     ...
14     case SIN:
15     {
16       tape[tape[i].arg1].bar_value += cos(tape[tape[i].arg1].value)*tape[i].bar_value;
17       break;
18     }
19     ...
20   }
21 }
```

Listing 6: Tape Interpretation Function `interpret_tape()`

2 Discrete Adjoint Method

In this part the aerodynamic shape optimization is introduced as an example for the need of gradients in applications. To this end assume that we want to minimize the drag $J(y,u) : Y \times U \to \mathbb{R}$ of an airfoil that is represented by the parameters $u \in U \subset \mathbb{R}^m$. $y \in Y \subset \mathbb{R}^n$ are the flow variables, i.e. velocity, density etc.. The problem can then be formulated as the following minimization problem:

$$\min_{y,u} J(y,u) \tag{11}$$

$$\text{subject to } G(y,u) = y, \tag{12}$$

where (12) represents the flow equations. The structure of (12) is typical for fixed-point based solvers for the nonlinear system that arises during the discretization of the Navier-Stokes equations. Efficient solution methods for this problem require the gradient of J with respect to the design parameters u to determine a new shape. A first naive approach is to use forward differentiation either by using finite differences,

$$\frac{\partial J}{\partial u_i}(y(u),u) = \frac{J(y(u),u) - J(y(u+e_i h), u+e_i h)}{h} + \mathcal{O}(h) \tag{13}$$

or by using the forward propagation from section 1.1. Nevertheless, both approaches have the major drawback that the cost of evaluating the gradient is proportional to the number of design variables m. A better approach is the application of optimal control to problem (11)-(12). Therefore we introduce the Lagrangian function L:

$$L(y,u,\lambda) := J(y,u) - \lambda^T (G(y,u) - y) = N(y,u,\lambda) + \lambda^T y \tag{14}$$

with the shifted Lagrangian $N(y,u,\lambda) := J(y,u) - \lambda^T G(y,u)$ and the Lagrangian multiplier $\lambda \in \mathbb{R}^n$. The first order necessary conditions for optimality [5] are then given by

$$\frac{\partial L}{\partial \lambda}(y,u,\lambda) = G(y,u) - y = 0 \tag{15}$$

$$\frac{\partial L}{\partial y}(y,u,\lambda) = \frac{\partial N}{\partial y}(y,u,\lambda) - \lambda^T = 0 \tag{16}$$

$$\frac{\partial L}{\partial u}(y,u,\lambda) = \frac{\partial N}{\partial u}(y,u,\lambda) = 0 \tag{17}$$

or, equivalently, a KKT-point (y^*, u^*, λ^*), that is, a point where the gradient of the Lagrangian (14) vanishes must satisfy

$$G(y^*,u^*) = y^* \qquad \Rightarrow \text{state equation} \tag{18}$$

$$N_y^T(y^*,u^*,\lambda^*) = \lambda^* \qquad \Rightarrow \text{adjoint equation} \tag{19}$$

$$N_u(y^*,u^*,\lambda^*) = 0, \qquad \Rightarrow \text{optimality condition} \tag{20}$$

where the subscript denotes the derivative with respect to this variable. If we assume that the solution of the state equation is given to us, e.g. by the flow solver, then we only need to solve the adjoint equation for λ to evaluate the gradient N_u. We use the structure of the shifted Lagrangian to derive an iterative method for the adjoint variable λ. The adjoint equation

$$N_y(y^*,u,\lambda^*) = J_y(y^*,u) + G_y^T(y^*,u)\lambda = \lambda. \tag{21}$$

can be rewritten as

$$(G_y^T(y^*,u) - I)\lambda^* = -J_y(y^*,u). \tag{22}$$

Equation (22) above is a linear system for λ^* if y^* is the (numerical) solution of the state equation and u is fixed. In principle any solver that can handle non-symmetric matrices would be suitable for solving this system. Another way of obtaining a solution relies upon the fact that (22) implies a splitting of the system matrix $G_y^T - I$ such that we obtain the iterative method

$$\lambda^{m+1} := G_y^T(y^*,u)\lambda^m + J_y(y^*,u) = N_y(y^*,u,\lambda^m), \quad m = 0,1,\ldots \tag{23}$$

This recurrence converges for arbitrary λ^0 in the image of N_y if $\rho(G_y^T) < 1$. In practice that restriction does not pose a problem since it is fulfilled if the state equation (18) has been solved with a converging Newton-type method because then

$$\rho(G_y^T) = \rho(G_y) < 1 \tag{24}$$

holds. All occuring gradients in equation (23) and equation (17) can be calculated using the Reverse Propagation of Gradients from section 1.2.

3 Results

For the results in this section the discrete adjoint method was implemented in the Stanford University Unstructured (SU^2) environment [6] using algorithmic differentiation. The SU^2 environment features a finite volume-based flow solver with state of the art preconditioners like for example full multi-grid schemes. It works with arbitrary unstructured grids and comprises solvers for the Euler, Navier-Stokes and RANS equations. In the following the drag coefficient is used as the target functional J.

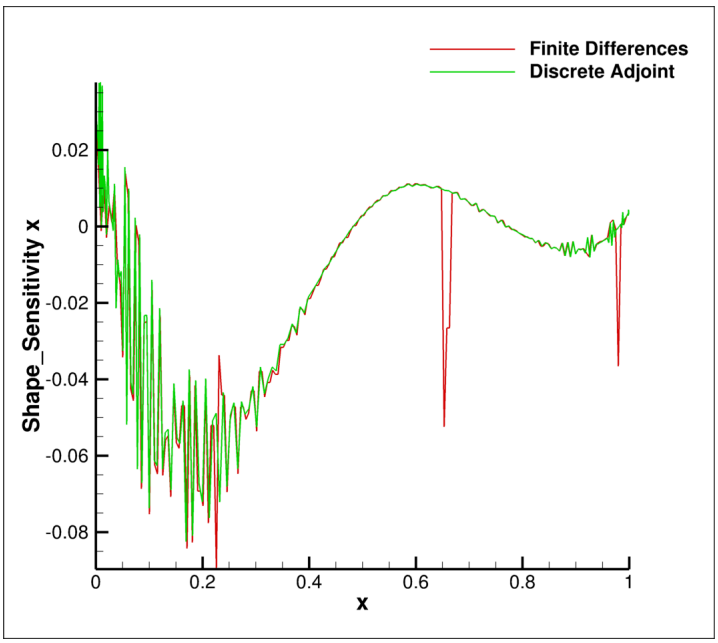


Figure 1: Validation of the gradient against finite differences.

3.1 Cylinder

The first test case is a stationary two-dimensional flow around a cylinder. This case was chosen to verify the proper differentiation of the turbulence model. We consider the Reynolds numbers (Re) 40 and 100 with and without the Spalart-Allmaras (SA) turbulence model activated. For the multigrid preconditioner a W-cycle with 3 levels is used. Time and space integration is accomplished by using the implicit Euler scheme and the Jameson-Schmidt-Turkel scheme, respectively. The design variables are the nodes on the cylinder surface.

Figure 1 shows the component in the x-direction of the gradient plotted against the finite difference solution on the upper half of the cylinder. The turbulence model is activated and the Reynolds number is set to 40. Although the gradient exhibits strong high frequency variations the two solutions conform suprisingly well. It is well known that at $Re = 40$ the flow is laminar almost everywhere such that the turbulence model has little influence on the flow. This is correctly depicted by the gradient in figure 2. At $Re = 40$ there is almost no difference between the gradients with enabled and disabled turbulence model, whereas for $Re = 100$ a difference is visible especially in the range between $x = 0.4$ to $x = 0.8$ where the flow starts to transition from laminar to turbulent. Thus, we can assume that the differentiation is correct.

3.2 RAE 2822

The second test case is a more practical example. We consider the RAE 2822 airfoil at transsonic flight conditions. The time and space discretization is the same as for the cylinder test case. The Reynolds number is fixed at $Re = 6.5 \cdot 10^6$ and the free-stream Mach number is $Ma = 0.729$. Furthermore the angle of attack is $\alpha = 2.31$. Again, the nodes comprising the airfoil are used as design variables.

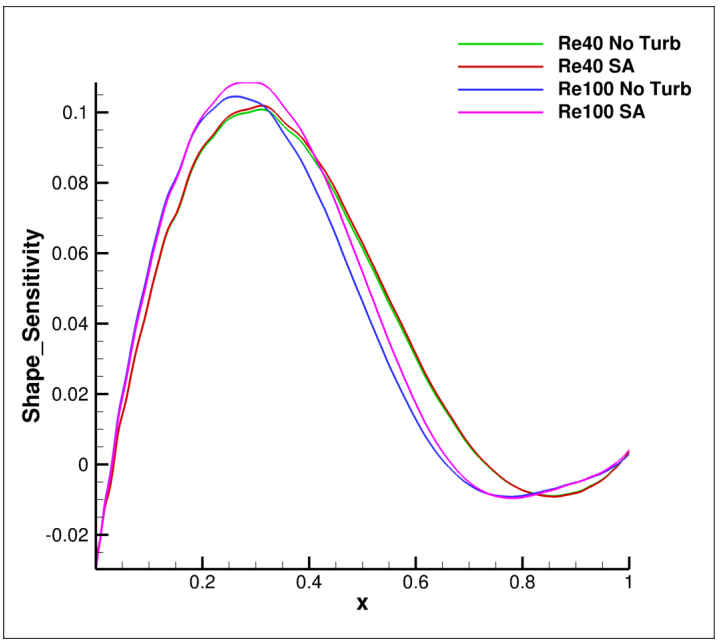


Figure 2: Comparison of the (smoothed) gradient in normal direction at $Re = 40$ and $Re = 100$

Figure 3 shows the density component of the Lagrange multiplier λ. Note, that the Lagrange multipliers have no direct physical interpretation and are shown just for verification purposes. Additionally a finite difference validation of the x-component of the gradient has been conducted. The result is shown in figure 4. Again, we have a very good agreement. Finally, in figure 5 the gradient in normal direction is plotted. It is clearly visible that around the shock position on the upper surface the gradient also exhibits a discontinuous behaviour. In front of the shock position the gradient is almost symmetric on the upper and lower surface. This essentially means that the shock has a large influence on the drag of the airfoil. Now that the gradient is available one could use a steepest descent line-search to update the design parameters and therefore determine a new shape that hopefully removes the shock and therefore has a lower drag coefficient.

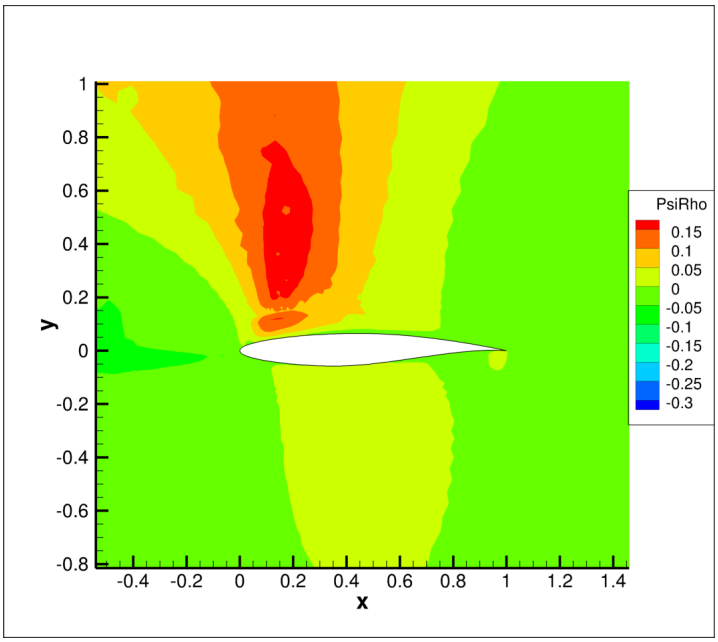


Figure 3: Calculated Adjoint density for the RAE2822 test case.

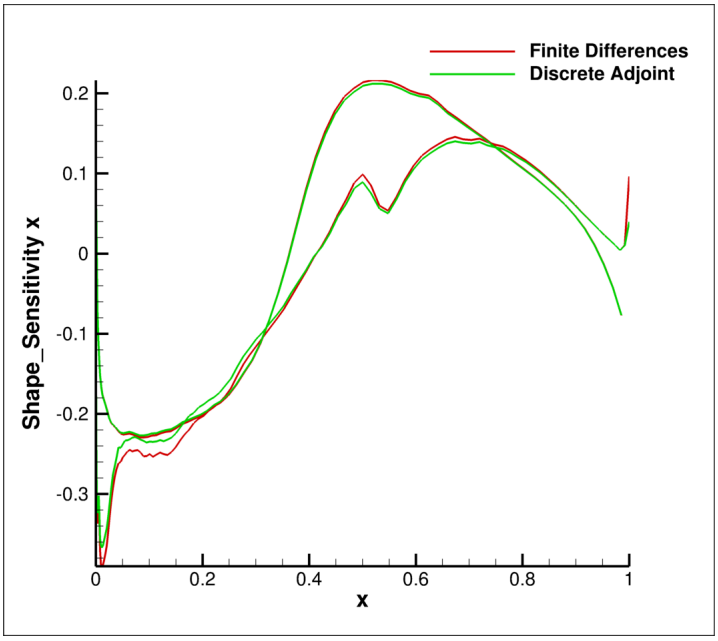


Figure 4: Finite difference validation for the RAE2822 test case.

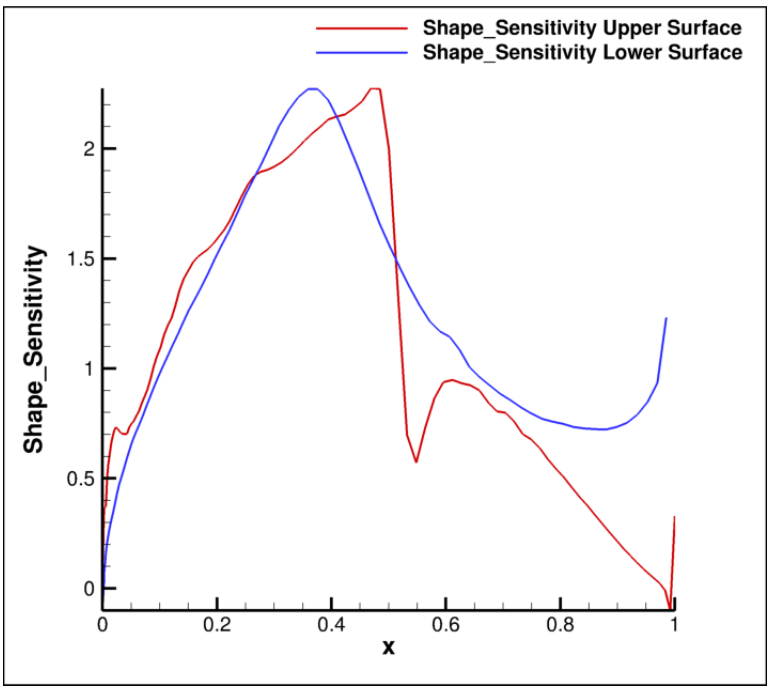


Figure 5: Gradient in normal direction for the RAE2822 test case.

References

[1] Griewank, Andreas and A. Walther (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. No. 105 in *Other Titles in Applied Mathematics*. SIAM, Philadelphia, PA, 2nd ed.

[2] Hamdi, Adel and A. Griewank (2011). *Reduced quasi-Newton method for simultaneous design and optimization*. Comput. Optim. Appl., 49(3):521–548.

[3] Hirsch, Charles (2007). *Numerical Computation of Internal and External Flows: The Fundamentals of Computational Fluid Dynamics*. No. Bd. 1. Elsevier Science.

[4] Lotz, Johannes, K. Leppkes and U. Naumann (2012). *dco/c++ - Derivative Code by Overloading in C++*. Technical Report AIB-2011-06, RWTH Aachen.

[5] Nocedal, Jorge and S. J. Wright (2006). *Numerical Optimization*. Springer, New York, 2nd ed.

[6] Palacios, F., M. R. Colonno, A. C. Aranake, A. Campos, S. R. Copeland, T. D. Economon, A. K. Lonkar, T. W. Lukaczyk, T. W. R. Taylor and J. J. Alonso (2013). *Stanford University Unstructured (SU2): An open-source integrated computational environment for multi-physics simulation and design*. 51st AIAA Aerospace Sciences Meeting and Exhibit.