# Chapter 4

# Sensitivity Analysis

## 4.1 Introduction

Sensitivity analysis consists in computing derivatives of one or more quantities (outputs) with respect to one or several independent variables (inputs). Although there are various uses for sensitivity information, our main motivation is the use of this information in *gradient-based optimization*. Since the calculation of gradients is often the most costly step in the optimization cycle, using efficient methods that accurately calculate sensitivities is extremely important.

Consider a general constrained optimization problem of the form:

$$\begin{aligned} \text{minimize} \quad & f(x_i) \\ \text{w.r.t} \quad & x_i \qquad\qquad i = 1, 2, \ldots, n \\ \text{subject to} \quad & c_j(x_i) \geq 0, \quad j = 1, 2, \ldots, m \end{aligned} \tag{4.1}$$

In order to solve this problem using a gradient-based optimization algorithm we usually require:

- The sensitivities of the objective function, $\nabla f(x) = \partial f / \partial x_i$ $(n \times 1)$.
- The sensitivities of all the active constraints at the current design point $\partial c_j / \partial x_i$ $(m \times n)$.

## 4.2 Motivation

By default, most gradient-based optimizers use finite-differences for sensitivity analysis. This is both costly and subject to inaccuracies.

When the cost of calculating the sensitivities is proportional to the number of design variables, and this number is large, sensitivity analysis is the bottleneck in the optimization cycle.

Accurate sensitivities are required for convergence.

### 4.2.1 Methods for Sensitivity Analysis

**Finite Differences:** very popular; easy, but lacks robustness and accuracy; run solver $n$ times.

**Complex-Step Method:** relatively new; accurate and robust; easy to implement and maintain; run solver $n$ times.

**Symbolic Differentiation:** accurate; restricted to explicit functions of low dimensionality.

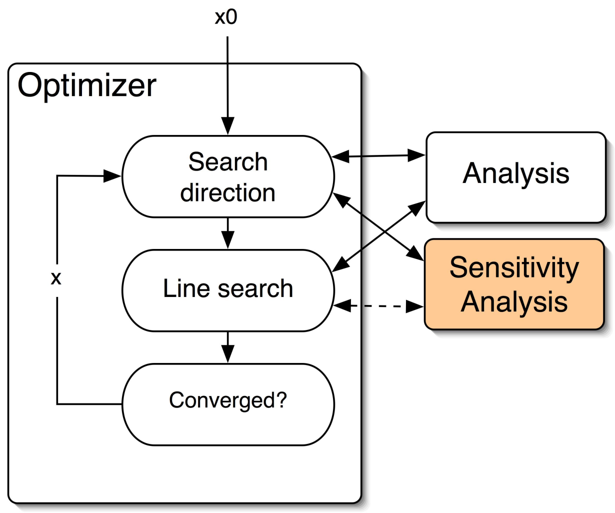


Figure 4.1: Dependence of gradient-based algorithms on sensitivity analysis

**Algorithmic/Automatic/Computational Differentiation:** accurate; ease of implementation and cost varies.

**(Semi)-Analytic Methods:** efficient and accurate; long development time; *cost can be independent of* $n$.

## 4.3 Finite Differences

Finite-difference formulae are very commonly used to estimate sensitivities. Although these approximations are neither particularly accurate or efficient, this method's biggest advantage resides in the fact that it is extremely easy to implement.

All the finite-difference formulae can be derived by truncating a Taylor series expanded about a given point $x$. Suppose we have a function $f$ of one variable $x$. A common estimate for the first derivative is the *forward-difference* which can be derived from the expansion of $f(x+h)$,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots, \tag{4.2}$$

Solving for $f'$ we get the finite-difference formula,

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \tag{4.3}$$

where $h$ is called the *finite-difference interval.* The truncation error is $\mathcal{O}(h)$, and hence this is a first-order approximation.

For a second-order estimate we can use the expansion of $f(x-h)$,

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \dots, \tag{4.4}$$

and subtract it from the expansion (4.2). The resulting equation can then be solved for the derivative of $f$ to obtain the *central-difference* formula,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \tag{4.5}$$

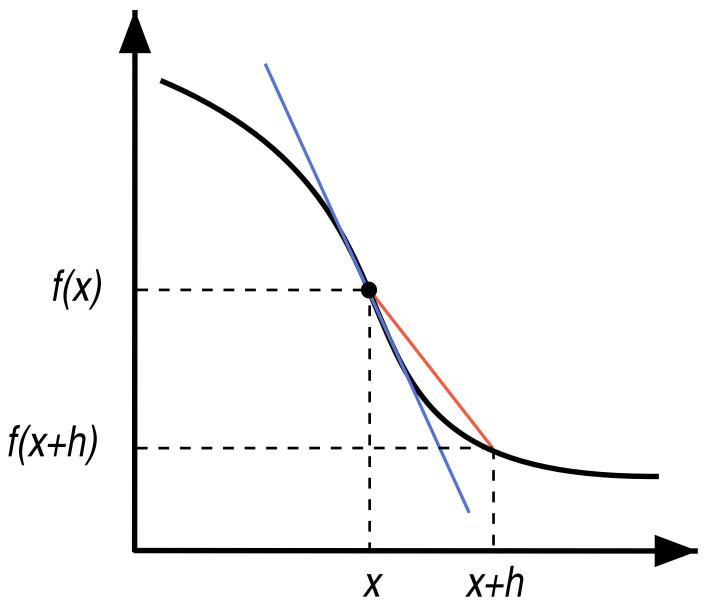


Figure 4.2: Graphical representation of the finite difference approximation

More accurate estimates can also be derived by combining different Taylor series expansions.

Formulas for estimating higher-order derivatives can be obtained by nesting finite-difference formulas. We can use, for example the central difference (4.5) to estimate the second derivative instead of the first,

$$f''(x) = \frac{f'(x+h) - f'(x-h)}{2h} + \mathcal{O}(h^2). \tag{4.6}$$

and use central difference again to estimate both $f'(x+h)$ and $f'(x-h)$ in the above equation to obtain,

$$f''(x) = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} + \mathcal{O}(h). \tag{4.7}$$

When estimating sensitivities using finite-difference formulae we are faced with the *step-size dilemma*, that is the desire to choose a small step size to minimize truncation error while avoiding the use of a step so small that errors due to subtractive cancellation become dominant.

Forward-difference approximation:

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h). \tag{4.8}$$

With 16-digit arithmetic,

| | |
|---|---|
| $f(x+h)$ | $+1.234567890123431$ |
| $f(x)$ | $+1.234567890123456$ |
| $\Delta$ f | $-0.000000000000025$ |

For functions of several variables, that is when $x$ is a vector, then we have to calculate each component of the gradient $\nabla f(x)$ by perturbing the corresponding variable $x_i$.

The cost of calculating sensitivities with finite-differences is therefore proportional to the number of design variables and $f$ must be calculated for each perturbation of $x_i$. This means that if we use forward differences, for example, the cost would be $n+1$ times the cost of calculating $f$.

## 4.4 The Complex-Step Derivative Approximation

### 4.4.1 Background

The use of complex variables to develop estimates of derivatives originated with the work of Lyness and Moler [12] and Lyness [13]. Their work produced several methods that made use of complex variables, including a reliable method for calculating the $n^{\text{th}}$ derivative of an analytic function. However, only recently has some of this theory been rediscovered by Squire and Trapp [22] and used to obtain a very simple expression for estimating the first derivative. This estimate is suitable for use in modern numerical computing and has shown to be very accurate, extremely robust and surprisingly easy to implement, while retaining a reasonable computational cost [14, 15].

### 4.4.2 Basic Theory

We will now see that a very simple formula for the first derivative of real functions can be obtained using complex calculus. The complex-step derivative approximation can also be derived using a Taylor series expansion. Rather than using a real step $h$, we now use a pure imaginary step, $ih$. If $f$ is a real function in real variables and it is also analytic, we can expand it in a Taylor series about a real point $x$ as follows,

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \dots \tag{4.9}$$

Taking the imaginary parts of both sides of (4.9) and dividing the equation by $h$ yields

$$f'(x) = \frac{\operatorname{Im}[f(x+ih)]}{h} + h^2\frac{f'''(x)}{3!} + \dots \tag{4.10}$$

Hence the approximations is a $\mathcal{O}(h^2)$ estimate of the derivative of $f$.

An alternative way of deriving and understanding the complex step is to consider a function, $f = u + iv$, of the complex variable, $z = x + iy$. If $f$ is analytic the Cauchy–Riemann equations apply, i.e.,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{4.11}$$

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{4.12}$$

These equations establish the exact relationship between the real and imaginary parts of the function. We can use the definition of a derivative in the right hand side of the first Cauchy–Riemann equation (4.11) to obtain,

$$\frac{\partial u}{\partial x} = \lim_{h\to 0}\frac{v(x+i(y+h)) - v(x+iy)}{h}. \tag{4.13}$$

where $h$ is a small real number. Since the functions that we are interested in are real functions of a real variable, we restrict ourselves to the real axis, in which case $y = 0$, $u(x) = f(x)$ and $v(x) = 0$. Equation (4.13) can then be re-written as,

$$\frac{\partial f}{\partial x} = \lim_{h\to 0}\frac{\operatorname{Im}[f(x+ih)]}{h}. \tag{4.14}$$

For a small discrete $h$, this can be approximated by,

$$\frac{\partial f}{\partial x} \approx \frac{\operatorname{Im}[f(x+ih)]}{h}. \tag{4.15}$$

We will call this the *complex-step derivative approximation*. This estimate is not subject to subtractive cancellation error, since it does not involve a difference operation. This constitutes a tremendous advantage over the finite-difference approaches expressed in (4.3, 4.5).

**Example 4.10.** The Complex-Step Method Applied to a Simple Function

To show the how the complex-step method works, consider the following analytic function:

$$f(x) = \frac{\mathrm{e}^x}{\sqrt{\sin^3 x + \cos^3 x}} \tag{4.16}$$

The exact derivative at $x = 1.5$ was computed analytically to 16 digits and then compared to the results given by the complex-step (4.15) and the forward and central finite-difference approximations.

Relative error in the sensitivity estimates given by finite-difference and the complex- step methods with the analytic result as the reference, i.e.,

$$\varepsilon = \frac{\left| f' - f'_{ref} \right|}{\left| f'_{ref} \right|}. \tag{4.17}$$

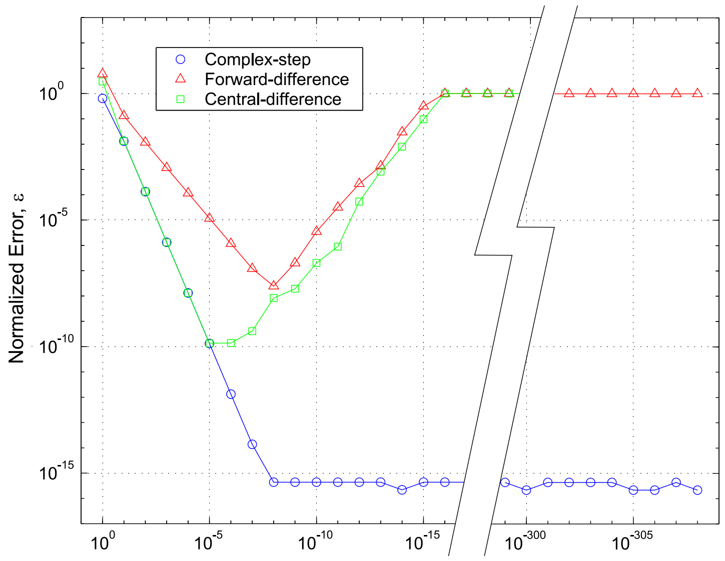


Figure 4.3: Relative error of the sensitivity vs. decreasing step size

The forward-difference estimate initially converges to the exact result at a linear rate since its truncation error is $\mathcal{O}(h)$, while the central-difference converges quadratically, as expected. However, as the step is reduced below a value of about $10^{-8}$ for the forward-difference and $10^{-5}$ for the central-difference, subtractive cancellation errors become significant and the estimates are unreliable. When the interval $h$ is so small that no difference exists in the output (for steps smaller than $10^{-16}$) the finite-difference estimates eventually yields zero and then $\varepsilon = 1$.

The complex-step estimate converges quadratically with decreasing step size, as predicted by the truncation error estimate. The estimate is practically insensitive to small step sizes and below an $h$ of the order of $10^{-8}$ it achieves the accuracy of the function evaluation. Comparing the best accuracy of each of these approaches, we can see that by using finite-difference we only achieve a fraction of the accuracy that is obtained by using the complex-step approximation.

The complex-step size can be made extremely small. However, there is a lower limit on the step size when using finite precision arithmetic. The range of real numbers that can be handled in numerical computing is dependent on the particular compiler that is used. In this case, the smallest non-zero number that can be represented is $10^{-308}$. If a number falls below this value, underflow occurs and the number drops to zero. Note that the estimate is still accurate down to a step of the order of $10^{-307}$. Below this, underflow occurs and the estimate results in `NaN`.

Comparing the accuracy of complex and real computations, there is an increased error in basic arithmetic operations when using complex numbers, more specifically when dividing and multiplying.

### 4.4.3 New Functions and Operators

To what extent can the complex-step method be used in an arbitrary algorithm? To answer this question, we have to look at each operator and function in the algorithm.

- Relational operators
  - Used with `if` statements to direct the execution thread.
  - Complex algorithm must follow same thread.
  - Therefore, compare only the real parts.
  - Also, `max`, `min`, etc.
- Arithmetic functions and operators:
  - Most of these have a mathematical standard definition that is analytic.
  - Some of them are implemented in Fortran.
  - Exception: `abs`

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} = \begin{cases} -1 & \Leftarrow x < 0 \\ +1 & \Leftarrow x > 0 \end{cases} \tag{4.18}$$

$$\texttt{abs}(x+iy) = \begin{cases} -x-iy & \Leftarrow x < 0 \\ +x+iy & \Leftarrow x \geq 0 \end{cases}. \tag{4.19}$$

### 4.4.4 Can the Complex-Step Method be Improved?

Improvements necessary because,

$$arcsin(z) = -i \log\left[iz + \sqrt{1-z^2}\right], \tag{4.20}$$

may yield a zero derivative...
How? If $z = x + ih$, where $x = \mathcal{O}(1)$ and $h = \mathcal{O}(10^{-20})$ then in the addition,

$$iz + z = (x-h) + i\,(x+h) \tag{4.21}$$

$h$ vanishes when using finite precision arithmetic.
Would like to keep the real and imaginary parts separate.
The complex definition of sine also problematic,

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}. \tag{4.22}$$

The complex trigonometric relation yields a better alternative,

$$\sin(x+ih) = \sin(x)\cosh(h) + i\cos(x)\sinh(h). \tag{4.23}$$

Note that linearizing this equation (that is for small $h$) this simplifies to,

$$\sin(x+ih) \approx \sin(x) + ih\cos(x). \tag{4.24}$$

From the standard complex definition,

$$\arcsin(z) = -i\log\left[iz + \sqrt{1-z^2}\right]. \tag{4.25}$$

Need real and imaginary parts to be calculated separately.
Linearizing in $h$ about $h=0$,

$$\arcsin(x+ih) \equiv \arcsin(x) + i\frac{h}{\sqrt{1-x^2}}. \tag{4.26}$$

### 4.4.5 Implementation Procedure

- Cookbook procedure for any programming language:
  - Substitute all `real` type variable declarations with `complex` declarations.
  - Define all functions and operators that are not defined for complex arguments.
  - A complex-step can then be added to the desired variable and the derivative can be estimated by $f' \approx \mathrm{Im}[f(x+ih)]/h$.
- Fortran 77: write new subroutines, substitute some of the intrinsic function calls by the subroutine names, e.g. `abs` by `c_abs`. But... need to know variable types in original code.
- Fortran 90: can overload intrinsic functions and operators, including comparison operators. Compiler knows variable types and chooses correct version of the function or operator.
- C/C++: also uses function and operator overloading.

### 4.4.6 Fortran Implementation

- `complexify.f90`: a module that defines additional functions and operators for complex arguments.
- `Complexify.py`: Python script that makes necessary changes to source code, e.g., type declarations.
- Features:
  - Script is versatile:
    * Compatible with many more platforms and compilers.
    * Supports MPI based parallel implementations.
    * Resolves some of the input and output issues.
  - Some of the function definitions were improved: tangent, inverse and hyperbolic trigonometric functions.

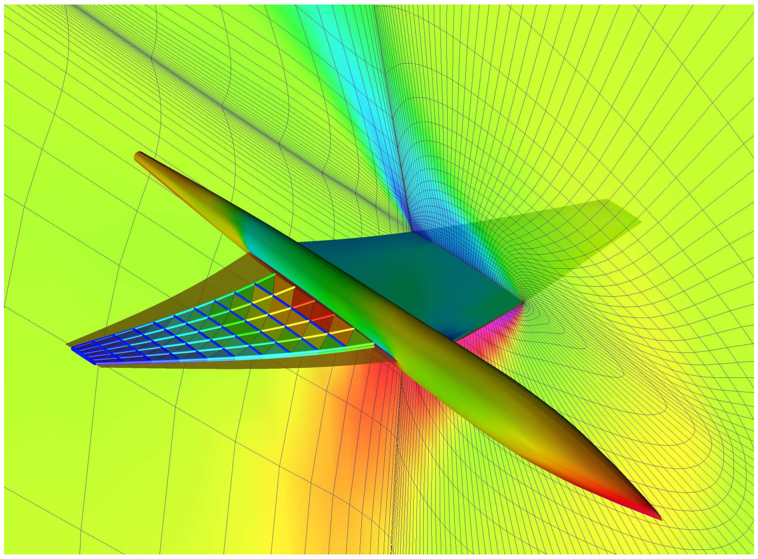


Figure 4.4: Aerostructural solution for the supersonic business jet

### 4.4.7 C/C++ Implementations

- `complexify.h`: defines additional functions and operators for the complex-step method.
- `derivify.h`: simple automatic differentiation. Defines a new type which contains the value and its derivative.

Templates, a C++ feature, can be used to create program source code that is independent of variable type declarations.

- Compared run time with real-valued code:
  - Complexified version: $\approx \times 3$
  - Algorithmic differentiation version: $\approx \times 2$

**Example 4.11.** 3D Aero-Structural Design Optimization Framework [16]

- Aerodynamics: SYN107-MB, a parallel, multiblock Navier–Stokes flow solver.
- Structures: detailed finite element model with plates and trusses.
- Coupling: high-fidelity, consistent and conservative.
- Geometry: centralized database for exchanges (jig shape, pressure distributions, displacements.)
- Coupled-adjoint sensitivity analysis

**Example 4.12.** Supersonic Viscous/Inviscid Solver [23]

Framework for preliminary design of natural laminar flow supersonic aircraft

- Transition prediction

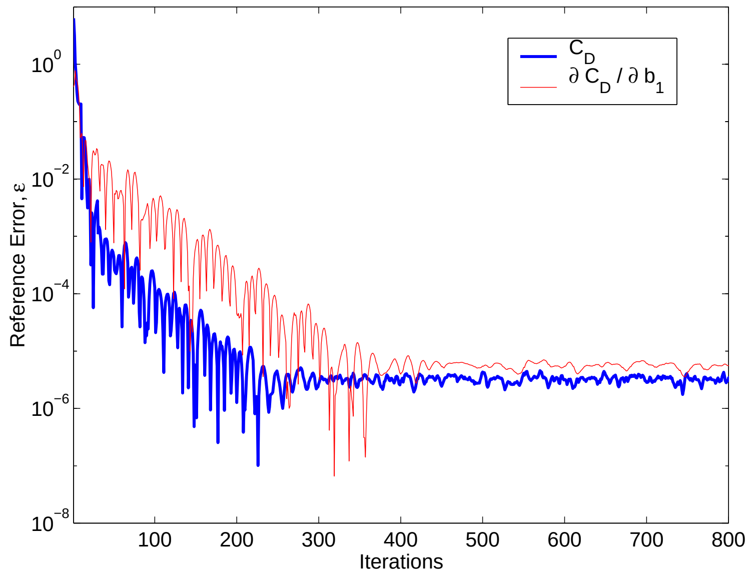


Figure 4.5: Convergence of $C_D$ and $\partial C_D/\partial b_1$

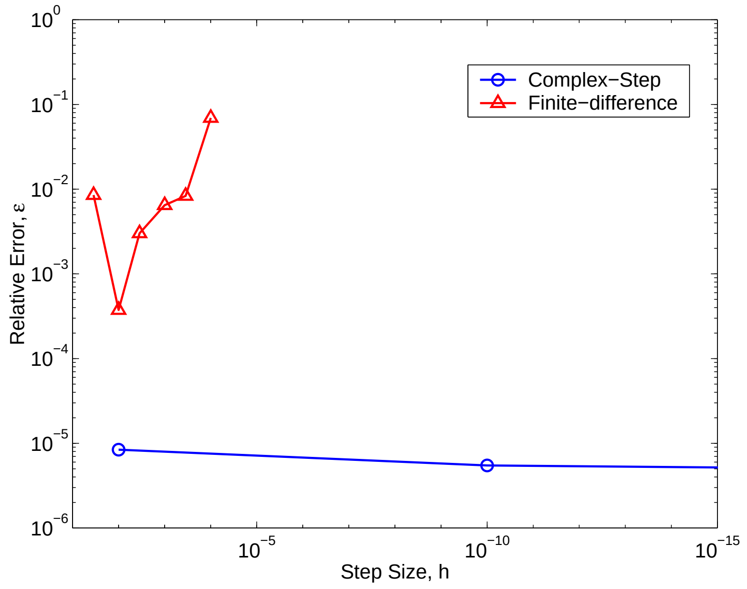


Figure 4.6: Sensitivity estimate vs. step size. Note that the finite-difference results are practically useless.

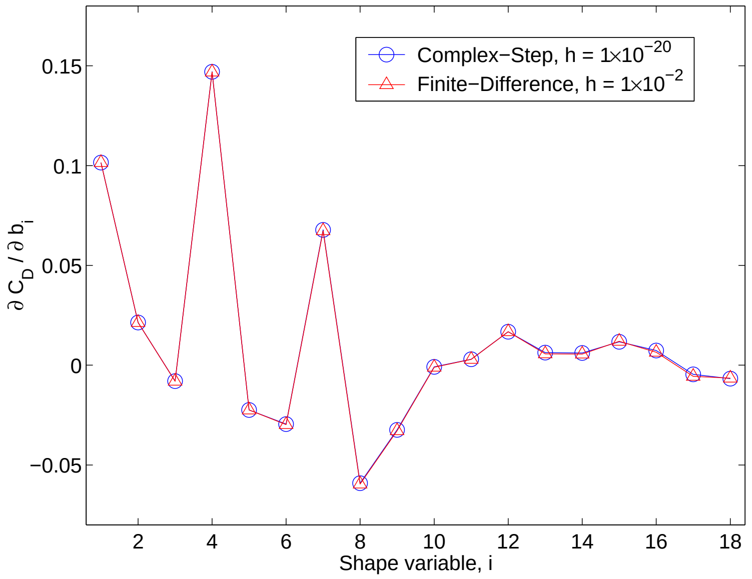
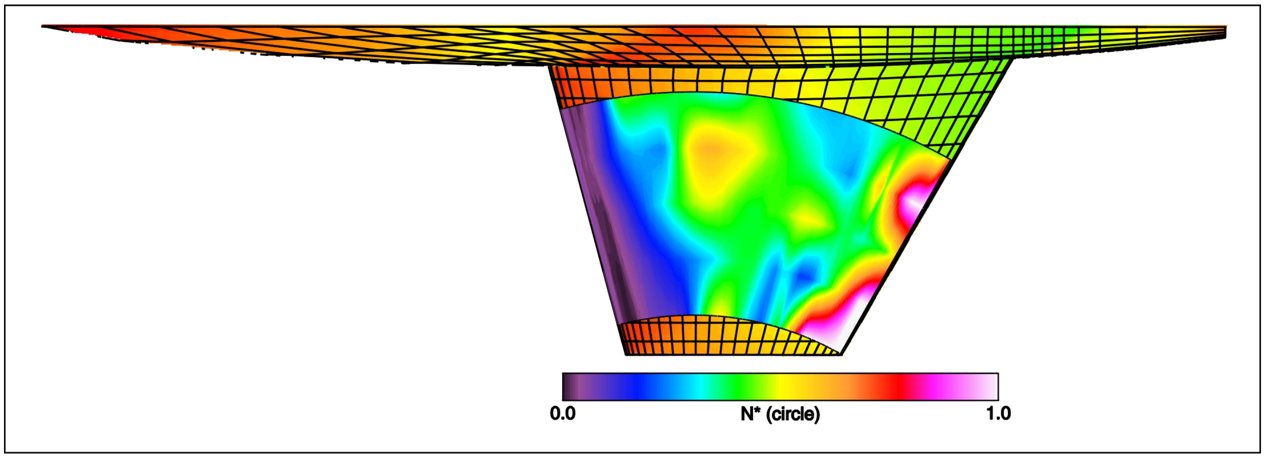


Figure 4.7: Sensitivity of $C_D$ to shape functions. The finite-different results were obtained after much effort to choose the best step.



- Viscous and inviscid drag
- Design optimization
  - Wing planform and airfoil design
  - Wing-Body intersection design

- Python wrapper defines geometry
- CH_GRID automatic grid generator
  - Wing only or wing-body
  - Complexified with our script
- CFL3D calculates Euler solution

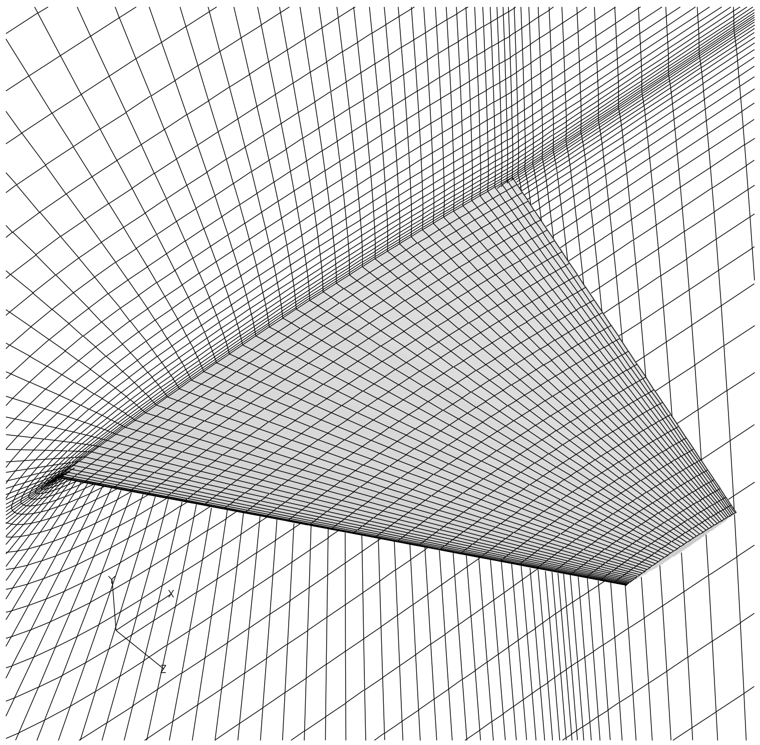


Figure 4.8: CFD grid

- Version 6 includes complex-step
- New improvements incorporated

- C++ post-processor for the...
- Quasi-3D boundary-layer solver
  - Laminar and turbulent
  - Transition prediction
  - C++ automatic differentiation
- Python wrapper collects data and computes structural constraints

## 4.5 Automatic Differentiation

Automatic differentiation (AD) — also known as computational differentiation or algorithmic differentiation — is a well known method based on the systematic application of the differentiation chain rule to computer programs. Although this approach is as accurate as an analytic method, it is potentially much easier to implement since this can be done automatically.

### 4.5.1 How it Works

The method is based on the application of the chain rule of differentiation to each operation in the program flow. The derivatives given by the chain rule can be propagated forward (*forward mode*) or backward (*reverse mode*).

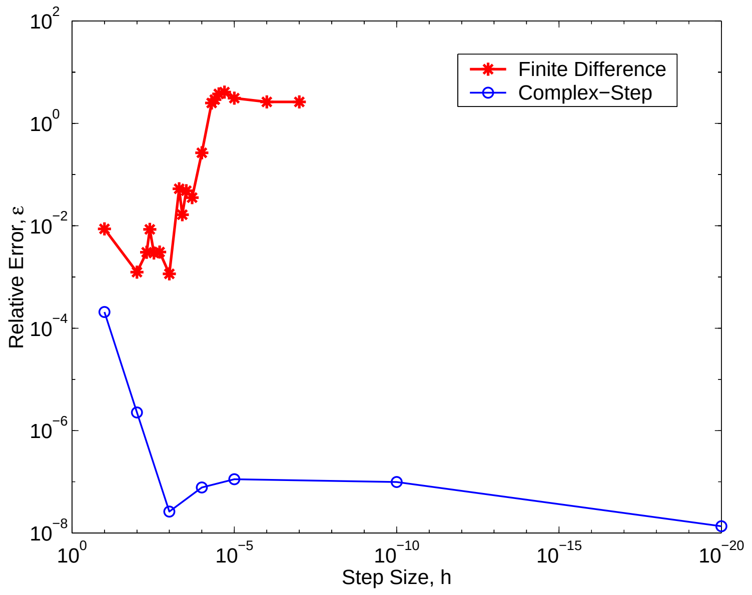


Figure 4.9: Sensitivity estimate vs. step size

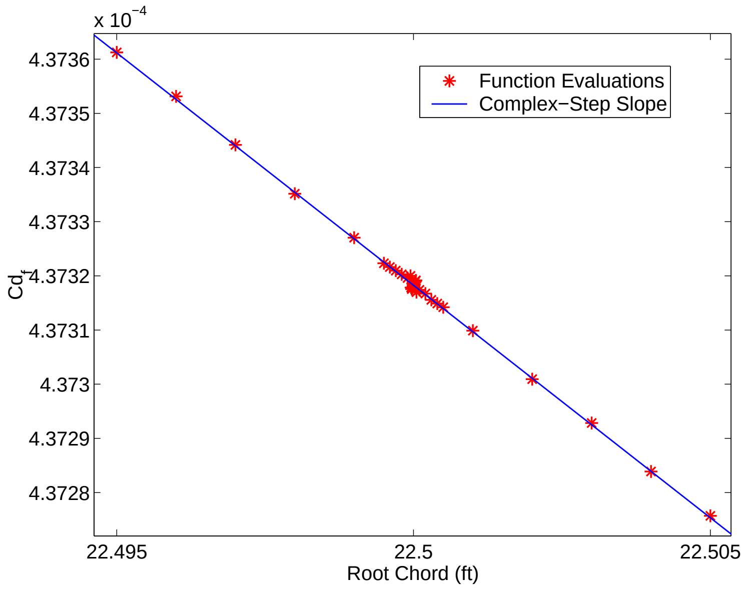


Figure 4.10: Laminar skin friction coefficient ($C_D$) vs. root chord

When using the forward mode, for each intermediate variable in the algorithm, a variation due to one input variable is carried through. This is very similar to the way the complex-step method works. To illustrate this, suppose we want to differentiate the multiplication operation, $f = x_1 x_2$, with respect to $x_1$. Table 4.1 compares how the differentiation would be performed using either automatic differentiation or the complex-step method.

| Automatic | Complex-Step |
|---|---|
| $\Delta x_1 = 1$ | $h_1 = 10^{-20}$ |
| $\Delta x_2 = 0$ | $h_2 = 0$ |
| $f = x_1 x_2$ | $f = (x_1 + ih_1)(x_2 + ih_2)$ |
| $\Delta f = x_1 \Delta x_2 + x_2 \Delta x_1$ | $f = x_1 x_2 - h_1 h_2 + i(x_1 h_2 + x_2 h_1)$ |
| $\mathrm{d}f/dx_1 = \Delta f$ | $\mathrm{d}f/dx_1 = \operatorname{Im} f/h$ |

Table 4.1: The differentiation of the multiplication operation $f = x_1 x_2$ with respect to $x_1$ using automatic differentiation and the complex-step derivative approximation.

As we can see, automatic differentiation stores the derivative value in a separate set of variables while the complex step carries the derivative information in the imaginary part of the variables. It is shown that in this case, the complex-step method performs one additional operation — the calculation of the term $h_1 h_2$ — which, for the purposes of calculating the derivative is superfluous (and equals zero in this particular case). The complex-step method will nearly always include these unnecessary computations which correspond to the higher order terms in the Taylor series expansion. For very small $h$, when using finite precision arithmetic, these terms have no effect on the real part of the result.

Although this example involves only one operation, both methods work for an algorithm involving an arbitrary sequence of operations by propagating the variation of one input forward throughout the code. This means that in order to calculate $n$ derivatives, the differentiated code must be executed $n$ times.

In general, any program can be written as a sequence of $m$ elementary functions $T_i$ for $i = 1, \dots, m$, where $T_i$ is a function only of variables $t_1, \dots, t_{i-1}$, and

$$t_i = T_i(t_1, \dots, t_{i-1}) \tag{4.27}$$

The chain rule for composite functions can be applied to each of these operations and is written as

$$\frac{\partial t_i}{\partial t_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial T_i}{\partial t_k} \frac{\partial t_k}{\partial t_j}, \quad \text{for} \quad j \le i \le n \tag{4.28}$$

where $\delta_{ij} = 1$ if $i = j$, or $\delta_{ij} = 0$ otherwise. Using the forward mode, we chose one $j$ and keep it fixed. We then work our way forward in the index $i = 1, 2, \dots, m$ until we get the desired derivative.

The reverse mode, on the other hand, works by fixing $i$, the desired quantity we want to differentiate, and working our way backward in the index $j = m, m-1, \dots, 1$ all the way to the independent variables.

**Example 4.13.** Forward mode differentiation of a simple function by hand

There is nothing like an example, so we will now use the forward mode of the chain rule to compute the derivatives of the vector function,

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} (x_1 x_2 + \sin x_1)\left(3x_2^2 + 6\right) \\ x_1 x_2 + x_2^2 \end{bmatrix} \tag{4.29}$$

evaluated at $x = [\pi/4, 2]$.

The solution can be obtained by hand differentiation,

$$\frac{\partial f}{\partial x} = \begin{bmatrix} (x_2 + \cos x_1)\left(3x_2^2 + 6\right) & x_1\left(3x_2^2 + 6\right) + 6x_2\left(x_1 x_2 + \sin x_1\right) \\ x_2 & x_1 + 2x_2 \end{bmatrix} \tag{4.30}$$

This matrix of sensitivities at the specified point is,

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 48.73 & 41.47 \\ 2.00 & 4.79 \end{bmatrix} \tag{4.31}$$

However, the point of this example is to show how the chain rule can be systematically applied in an automated fashion. To illustrate this more clearly, we write the function as a series of unary and binary computations:

$$\begin{aligned}
t_1 &= x_1 \\
t_2 &= x_2 \\
t_3 &= T_3(t_1) = \sin t_1 \\
t_4 &= T_4(t_1, t_2) = t_1 t_2 \\
t_5 &= T_5(t_2) = t_2^2 \\
t_6 &= 3 \\
t_7 &= T_7(t_3, t_4) = t_3 + t_4 \\
t_8 &= T_8(t_5, t_6) = t_5 t_6 \\
t_9 &= 6 \\
t_{10} &= T_{10}(t_8, t_9) = t_8 + t_9 \\
t_{11} &= T_{11}(t_7, t_{10}) = t_7 t_{10} \quad (= f_1) \\
t_{12} &= T_{12}(t_4, t_5) = t_4 + t_5 \quad (= f_2)
\end{aligned}$$

Thus in this case, $m = 12$.

Now let's use the forward mode to compute $\partial f_1/\partial x_1$ in our example, which is $\partial t_{11}/\partial t_1$. Thus, we set $j = 1$ and keep it fixed, and then vary $i = 1, 2, \ldots, 11$. Note that in the sum in the chain rule, we only include the $k$'s for which $\partial T_i/\partial t_k \neq 0$.

$$\begin{aligned}
\frac{\partial t_1}{\partial t_1} &= 1\\
\frac{\partial t_2}{\partial t_1} &= 0\\
\frac{\partial t_3}{\partial t_1} &= \frac{\partial T_3}{\partial t_1}\frac{\partial t_1}{\partial t_1} = \cos t_1 \times 1 = \cos t_1\\
\frac{\partial t_4}{\partial t_1} &= \frac{\partial T_4}{\partial t_1}\frac{\partial t_1}{\partial t_1} + \frac{\partial T_4}{\partial t_2}\frac{\partial t_2}{\partial t_1} = t_2 \times 1 + t_1 \times 0 = t_2\\
\frac{\partial t_5}{\partial t_1} &= \frac{\partial T_5}{\partial t_2}\frac{\partial t_2}{\partial t_1} = 2t_2 \times 0 = 0\\
\frac{\partial t_6}{\partial t_1} &= 0\\
\frac{\partial t_7}{\partial t_1} &= \frac{\partial T_7}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial T_7}{\partial t_4}\frac{\partial t_4}{\partial t_1} = 1 \times \cos t_1 + 1 \times t_2 = \cos t_1 + t_2\\
\frac{\partial t_8}{\partial t_1} &= \frac{\partial T_8}{\partial t_5}\frac{\partial t_5}{\partial t_1} + \frac{\partial T_8}{\partial t_6}\frac{\partial t_6}{\partial t_1} = t_6 \times 0 + t_5 \times 0 = 0\\
\frac{\partial t_9}{\partial t_1} &= 0\\
\frac{\partial t_{10}}{\partial t_1} &= \frac{\partial T_{10}}{\partial t_8}\frac{\partial t_8}{\partial t_1} + \frac{\partial T_{10}}{\partial t_9}\frac{\partial t_9}{\partial t_1} = 1 \times 0 + 1 \times 0 = 0\\
\frac{\partial t_{11}}{\partial t_1} &= \frac{\partial T_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_1} + \frac{\partial T_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_1} = t_{10}\left(\cos t_1 + t_2\right) + t_7 \times 0 = \left(3t_2^2 + 6\right)\left(\cos t_1 + t_2\right)
\end{aligned}$$

Note that although we did not set out to compute $\partial t_{12}/\partial t_1$, this can be found with little additional cost:

$$\frac{\partial t_{12}}{\partial t_1} = \frac{\partial T_{12}}{\partial t_4}\frac{\partial t_4}{\partial t_1} + \frac{\partial T_{12}}{\partial t_5}\frac{\partial t_5}{\partial t_1} = 1 \times t_2 + 1 \times 0 = t_2$$

**Example 4.14.** Forward mode algorithmic differentiation of a simple subroutine

```
SUBROUTINE CALCF(x, f)
  REAL :: x(2), f(2), t(12)
  t(1) = x(1)
  t(2) = x(2)
  t(3) = SIN(t(1))
  t(4) = t(1) * t(2)
  t(5) = t(2)**2
  t(6) = 3
  t(7) = t(3) + t(4)
  t(8) = t(5) * t(6)
  t(9) = 6
  t(10) = t(8) + t(9)
  t(11) = t(7) * t(10)
  t(12) = t(4) + t(5)
  f(1) = t(11)
  f(2) = t(12)
END SUBROUTINE CALCF
```

Figure 4.11: Fortran source code for simple function

```
SUBROUTINE CALCF_D(x, xd, f, fd)
  REAL :: x(2), f(2), t(12)
  REAL :: xd(2), fd(2), td(12)
  td = 0.0
  td(1) = xd(1)
  t(1) = x(1)
  td(2) = xd(2)
  t(2) = x(2)
  td(3) = td(1)*COS(t(1))
  t(3) = SIN(t(1))
  td(4) = td(1)*t(2) + t(1)*td(2)
  t(4) = t(1)*t(2)
  td(5) = 2*t(2)*td(2)
  t(5) = t(2)**2
  td(6) = 0.0
  t(6) = 3
  td(7) = td(3) + td(4)
  t(7) = t(3) + t(4)
  td(8) = td(5)*t(6) + t(5)*td(6)
  t(8) = t(5)*t(6)
  td(9) = 0.0
  t(9) = 6
  td(10) = td(8) + td(9)
  t(10) = t(8) + t(9)
  td(11) = td(7)*t(10) + t(7)*td(10)
  t(11) = t(7)*t(10)
  td(12) = td(4) + td(5)
  t(12) = t(4) + t(5)
  fd(1) = td(11)
  f(1) = t(11)
  fd(2) = td(12)
  f(2) = t(12)
END SUBROUTINE CALCF_D
```

Figure 4.12: Fortran source code for simple function differentiated in forward mode using source code transformation

In the real world, you would not code this way, instead, you would use two lines of code, as shown in Fig. 4.13.

```
SUBROUTINE CALCF2(x, f)
  REAL :: x(2), f(2)
  f(1) = (x(1)*x(2) + SIN(x(1))) * (3*x(2)**2 + 6)
  f(2) = x(1)*x(2) + x(2)**2
END SUBROUTINE CALCF2
```

Figure 4.13: Fortran source code for simple function

```
SUBROUTINE CALCF2_D(x, xd, f, fd)
  REAL :: x(2), f(2)
  REAL :: xd(2), fd(2)
  fd(1) = (xd(1)*x(2)+x(1)*xd(2)+xd(1)*COS(x(1)))*(3*x(2)**2+6) + (x(1)*&
&    x(2)+SIN(x(1)))*3*2*x(2)*xd(2)
  f(1) = (x(1)*x(2)+SIN(x(1)))*(3*x(2)**2+6)
  fd(2) = xd(1)*x(2) + x(1)*xd(2) + 2*x(2)*xd(2)
  f(2) = x(1)*x(2) + x(2)**2
END SUBROUTINE CALCF2_D
```

Figure 4.14: Fortran source code for simple function differentiated in forward mode using source code transformation

**Example 4.15.** Reverse mode differentiation of a simple function by hand

For the reverse mode, it is useful to have in mind the dependence of all the intermediate variables. This is given by the graph of the algorithm, shown in Figure 4.15 for the case of our simple function.

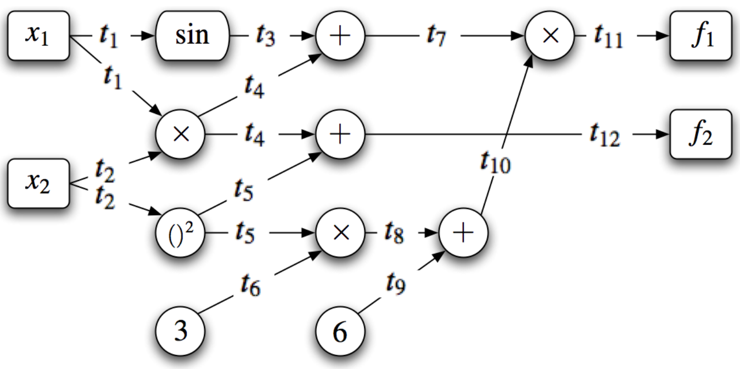


Figure 4.15: Graph showing dependency of independent, intermediate and dependent variables.

To use the chain rule in reverse, we set $i = 11$ and loop $j = 11, 10, \ldots, 1$:

$$
\begin{aligned}
\frac{\partial t_{11}}{\partial t_{11}} &= 1 \\
\frac{\partial t_{11}}{\partial t_{10}} &= \frac{\partial T_{11}}{\partial t_{10}} \frac{\partial t_{10}}{\partial t_{10}} = t_7 \times 1 = t_7 \\
\frac{\partial t_{11}}{\partial t_{9}} &= \frac{\partial T_{11}}{\partial t_{9}} \frac{\partial t_{9}}{\partial t_{9}} + \frac{\partial T_{11}}{\partial t_{10}} \frac{\partial t_{10}}{\partial t_{9}} = 0 \times 1 + t_7 \times 1 = t_7 \\
\frac{\partial t_{11}}{\partial t_{8}} &= \frac{\partial T_{11}}{\partial t_{8}} \frac{\partial t_{8}}{\partial t_{8}} + \frac{\partial T_{11}}{\partial t_{9}} \frac{\partial t_{9}}{\partial t_{8}} + \frac{\partial T_{11}}{\partial t_{10}} \frac{\partial t_{10}}{\partial t_{8}} = 0 \times 1 + 0 \times 0 + t_7 \times 1 = t_7 \\
\frac{\partial t_{11}}{\partial t_{7}} &= \frac{\partial T_{11}}{\partial t_{7}} \frac{\partial t_{7}}{\partial t_{7}} + \ldots + \frac{\partial T_{11}}{\partial t_{10}} \frac{\partial t_{10}}{\partial t_{7}} = t_{10} \times 1 + t_7 \times 0 = t_{10} \\
\frac{\partial t_{11}}{\partial t_{6}} &= t_5 t_7 \\
\frac{\partial t_{11}}{\partial t_{5}} &= t_7 t_6 \\
\frac{\partial t_{11}}{\partial t_{4}} &= t_{10} \\
\frac{\partial t_{11}}{\partial t_{3}} &= t_{10} \\
\frac{\partial t_{11}}{\partial t_{2}} &= t_{10} t_1 + t_7 t_6 2 t_2 = \left(3t_2^2 + 6\right) t_1 + 6 t_2 \left(\sin t_1 + t_1 t_2\right) \\
\frac{\partial t_{11}}{\partial t_{1}} &= t_{10} \cos t_1 + t_{10} t_2 = \left(3x_2^2 + 6\right) \left(\cos x_1 + x_2\right)
\end{aligned}
$$

Note that although we didn't set out to compute $\partial t_{11} \partial t_2$, it had to be computed anyway.

The cost of calculating the derivative of one output to many inputs is not proportional to the number of input but to the number of outputs. Since when using the reverse mode we need to store *all* the intermediate variables as well as the complete graph of the algorithm, the amount of memory that is necessary increases dramatically. In the case of three-dimensional iterative solver, the cost of using this mode can be prohibitive.

**Example 4.16.** Fortran code differentiated in reverse mode using source transformation

Now we explain forward and reverse automatic differentiation using matrix algebra. Recall the chain rule (4.32) that forms the basis for both the forward and reverse modes,

$$
\frac{\partial t_i}{\partial t_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial T_i}{\partial t_k} \frac{\partial t_k}{\partial t_j}, \quad \text{for} \quad j \le i \le n. \tag{4.32}
$$

The partial derivatives of the elementary functions $T_i$ with respect to $t_i$ form the Jacobian matrix,

$$
\mathbf{DT} = \frac{\partial T_i}{\partial t_j} = \begin{bmatrix} 0 & \cdots & & \\ \frac{\partial T_2}{\partial t_1} & 0 & \cdots & \\ \frac{\partial T_3}{\partial t_1} & \frac{\partial T_3}{\partial t_2} & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{4.33}
$$

```
SUBROUTINE CALCF_B(x, xb, f, fb)
  REAL :: x(2), f(2), t(12)
  REAL :: xb(2), fb(2), tb(12)
  INTRINSIC SIN
  t(1) = x(1)
  t(2) = x(2)
  CALL PUSHREAL4(t(3))
  t(3) = SIN(t(1))
  CALL PUSHREAL4(t(4))
  t(4) = t(1)*t(2)
  CALL PUSHREAL4(t(5))
  t(5) = t(2)**2
  CALL PUSHREAL4(t(6))
  t(6) = 3
  CALL PUSHREAL4(t(7))
  t(7) = t(3) + t(4)
  CALL PUSHREAL4(t(8))
  t(8) = t(5)*t(6)
  CALL PUSHREAL4(t(9))
  t(9) = 6
  CALL PUSHREAL4(t(10))
  t(10) = t(8) + t(9)
  tb = 0.0
  tb(12) = tb(12) + fb(2)
  fb(2) = 0.0
  tb(11) = tb(11) + fb(1)
  fb(1) = 0.0
  tb(4) = tb(4) + tb(12)
  tb(5) = tb(5) + tb(12)
  tb(12) = 0.0
  tb(7) = tb(7) + t(10)*tb(11)
  tb(10) = tb(10) + t(7)*tb(11)
  tb(11) = 0.0
  CALL POPREAL4(t(10))
  tb(8) = tb(8) + tb(10)
  tb(9) = tb(9) + tb(10)
  tb(10) = 0.0
  CALL POPREAL4(t(9))
  tb(9) = 0.0
  CALL POPREAL4(t(8))
  tb(5) = tb(5) + t(6)*tb(8)
  tb(6) = tb(6) + t(5)*tb(8)
  tb(8) = 0.0
  CALL POPREAL4(t(7))
  tb(3) = tb(3) + tb(7)
  tb(4) = tb(4) + tb(7)
  tb(7) = 0.0
  CALL POPREAL4(t(6))
  tb(6) = 0.0
  CALL POPREAL4(t(5))
  tb(2) = tb(2) + 2*t(2)*tb(5)
  tb(5) = 0.0
  CALL POPREAL4(t(4))
  tb(1) = tb(1) + t(2)*tb(4)
  tb(2) = tb(2) + t(1)*tb(4)
  tb(4) = 0.0
  CALL POPREAL4(t(3))
  tb(1) = tb(1) + COS(t(1))*tb(3)
  tb(3) = 0.0
  xb = 0.0
  xb(2) = xb(2) + tb(2)
  tb(2) = 0.0
  xb(1) = xb(1) + tb(1)
END SUBROUTINE CALCF_B
```

```
SUBROUTINE CALCF2_B(x, xb, f, fb)
  IMPLICIT NONE
  REAL :: x(2), f(2)
  REAL :: xb(2), fb(2)
  INTRINSIC SIN
  REAL :: tempb
  xb = 0.0
  xb(1) = xb(1) + x(2)*fb(2)
  xb(2) = xb(2) + (2*x(2)+x(1))*fb(2)
  fb(2) = 0.0
  tempb = (3*x(2)**2+6)*fb(1)
  xb(1) = xb(1) + (COS(x(1))+x(2))*tempb
  xb(2) = xb(2) + 3*(x(1)*x(2)+SIN(x(1)))*2*x(2)*fb(1) + x(1)*tempb
  fb(1) = 0.0
END SUBROUTINE CALCF2_B
```

Figure 4.17: Fortran source code for simple function differentiated in reverse mode using source code transformation

The total derivatives of the variables $t_i$ form another matrix,

$$\mathbf{Dt} = \frac{\partial t_i}{\partial t_j} = \begin{bmatrix} 1 & 0 & \cdots & \\ \frac{\partial t_2}{\partial t_1} & 1 & 0 & \cdots \\ \frac{\partial t_3}{\partial t_1} & \frac{\partial t_3}{\partial t_2} & 1 & 0 \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \tag{4.34}$$

This can be written as a matrix equation,

$$\begin{aligned} \mathbf{Dt} &= \mathbf{I} + \mathbf{DT}\mathbf{Dt} && \Leftrightarrow \\ (\mathbf{I} - \mathbf{DT})\,\mathbf{Dt} &= \mathbf{I} && \Leftrightarrow \qquad (4.35) \\ \mathbf{Dt} &= (\mathbf{I} - \mathbf{DT})^{-1} && \Leftrightarrow \\ \mathbf{Dt}\,(\mathbf{I} - \mathbf{DT}) &= \mathbf{I} && \Leftrightarrow \\ (\mathbf{I} - \mathbf{DT})^T\,\mathbf{Dt}^T &= \mathbf{I} && \qquad (4.36) \end{aligned}$$

**Example 4.17.** Simple function differentiation in matrix form
In our example, this matrix equation is,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\sqrt{2}/2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & -\pi/4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & -4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -18 & 0 & 0 & -2.28 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2} \\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2} \\ \frac{\partial t_3}{\partial x_1} & \frac{\partial t_3}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial t_9}{\partial x_1} & \frac{\partial t_9}{\partial x_2} \\ \frac{\partial t_{10}}{\partial x_1} & \frac{\partial t_{10}}{\partial x_2} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.37}$$

The solution of this system is

$$\begin{bmatrix} \frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2} \\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2} \\ \frac{\partial t_3}{\partial x_1} & \frac{\partial t_3}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial t_{10}}{\partial x_1} & \frac{\partial t_{10}}{\partial x_2} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.71 & 0 \\ 2 & 0.79 \\ 0 & 4 \\ 0 & 0 \\ 2.71 & 0.79 \\ 0 & 12 \\ 0 & 0 \\ 0 & 12 \\ 48.73 & 41.47 \\ 2 & 4.79 \end{bmatrix} \tag{4.38}$$

### 4.5.2 Tools for Algorithmic Differentiation

There are two main methods for implementing automatic differentiation: by source code transformation or by using derived datatypes and operator overloading.

To implement automatic differentiation by source transformation, the whole source code must be processed with a parser and all the derivative calculations are introduced as additional lines of code. The resulting source code is greatly enlarged and it becomes practically unreadable. This fact might constitute an implementation disadvantage as it becomes impractical to debug this new extended code. One has to work with the original source, and every time it is changed (or if different derivatives are desired) one must rerun the parser before compiling a new version. The advantage is that this method tends to yield faster code.

In order to use derived types, we need languages that support this feature, such as Fortran 90 or C++. To implement automatic differentiation using this feature, a new type of structure is created that contains both the value and its derivative. All the existing operators are then re-defined (overloaded) for the new type. The new operator has exactly the same behavior as before for the value part of the new type, but uses the definition of the derivative of the operator to calculate the derivative portion. This results in a very elegant implementation since very few changes are required in the original code.

There are automatic differentiation tools available for a variety of programming languages including Fortran, C/C++ and Matlab. They have been extensively developed and provide the user with great functionality, including the calculation of higher-order derivatives and reverse mode options.

#### Fortran

ADIFOR [6], TAF [8], TAMC [9], DAFOR, GRESS and Tapenade [11, 19] are some of the tools available for Fortran that use source transformation. The necessary changes to the source code are made automatically. The derived datatype approach is used in the following tools: AD01, ADOL-F, IMAS and OPTIMA90. Although it is in theory possible to have a script make the necessary changes in the source code automatically, none of these tools have this facility and the changes must be done manually.

#### C/C++:

Established tools for automatic differentiation also exist for C/C++. These include include ADIC, an implementation mirroring ADIFOR, and ADOL-C, a free package that uses operator overloading and can operate in the forward or reverse modes and compute higher order derivatives.

### 4.5.3 The Connection to Algorithmic Differentiation

The new function definitions in these examples can be generalized:

$$f(x+ih) \equiv f(x) + ih\frac{\partial f(x)}{\partial x}. \tag{4.39}$$

The real part is the real function and the imaginary part is the derivative multiplied by $h$. Defining functions this way, or for small enough step using finite precision arithmetic, *the complex-step method is the same as automatic differentiation.*

Looking at a simple operation, e.g. $f = x_1 x_2$,

| Algorithmic | Complex-Step |
|---|---|
| $\Delta x_1 = 1$ | $h_1 = 10^{-20}$ |
| $\Delta x_2 = 0$ | $h_2 = 0$ |
| $f = x_1 x_2$ | $f = (x_1 + ih_1)(x_2 + ih_2)$ |
| $\Delta f = x_1 \Delta x_2 + x_2 \Delta x_1$ | $f = x_1 x_2 - h_1 h_2 + i(x_1 h_2 + x_2 h_1)$ |
| $\mathrm{d}f/\,\mathrm{d}x_1 = \Delta f$ | $\mathrm{d}f/\,\mathrm{d}x_1 = \operatorname{Im} f/h$ |

Complex-step method computes one extra term. Other functions are similar:

- Superfluous calculations are made.
- For $h \le x \times 10^{-20}$ they vanish but still affect speed.

**Example 4.18.** Complex-step method example Consider a more involved function, e.g., $f = (xy + \sin x + 4)(3y^2 + 6)$,

$$\begin{aligned}
t_1 &= x + ih, \quad t_2 = y \\
t_3 &= xy + iyh \\
t_4 &= \sin x \cosh h + i\cos x \sinh h \\
t_5 &= xy + \sin x \cosh h + i(yh + \cos x \sinh h) \\
t_6 &= xy + \sin x \cosh h + 4 + i(yh + \cos x \sinh h) \\
t_7 &= y^2, \quad t_8 = 3y^2, \quad t_9 = 3y^2 + 6 \\
t_{10} &= (xy + \sin x \cosh h + 4)\left(3y^2 + 6\right) + i(yh + \cos x \sinh h)\left(3y^2 + 6\right) \\
\frac{\mathrm{d}f}{\mathrm{d}x} &\approx \frac{\operatorname{Im}\left[f(x+ih, y)\right]}{h} = \left(y + \cos x \frac{\sinh h}{h}\right)\left(3y^2 + 6\right)
\end{aligned}$$

Large body of research on automatic differentiation can now be applied to the complex-step method:

- Singularities: non-analytic points
- `if` statements: piecewise function definitions
- Convergence for iterative solvers
- Other issues addressed by the automatic differentiation community

### 4.5.4 Algorithmic Differentiation vs. Complex Step

- Algorithmic Differentiation
  - Source transformation (ADIFOR, ADIC):
    resulting code is unmaintainable.
  - Derived datatype and operator overloading (ADOL-F, ADOL-C):
    far fewer changes are necessary in source code, requires object-oriented language.
- Complex Step:
  - Even fewer changes are required.
  - Resulting code is maintainable.
  - Can be easily implemented in any programming language that supports complex arithmetic.

## 4.6 Analytic Sensitivity Analysis

Analytic methods are the most accurate and efficient methods available for sensitivity analysis. They are, however, more involved than the other methods we have seen so far since they require the knowledge of the governing equations and the algorithm that is used to solve those equations. In this section we will learn how to compute analytic sensitivities with direct and adjoint methods. We will start with single discipline systems and then generalize for the case of multiple systems such as we would encounter in MDO.

### 4.6.1 Notation

| | |
|---|---|
| $f$ | function of interest/output (could be a vector) |
| $\mathcal{R}_k$ | residuals of governing equation, $k = 1, \ldots, N_{\mathcal{R}}$ |
| $x_n$ | design/independent/input variables, $n = 1, \ldots, N_x$ |
| $y_i$ | state variables, $i = 1, \ldots, N_{\mathcal{R}}$ |
| $\Psi_k$ | adjoint vector, $k = 1, \ldots, N_{\mathcal{R}}$ |

### 4.6.2 Basic Equations

The main objective is to calculate the sensitivity of a multidisciplinary function of interest with respect to a number of design variables. The function of interest can be either the objective function or any of the constraints specified in the optimization problem. In general, such functions depend not only on the design variables, but also on the physical state of the multidisciplinary system. Thus we can write the function as

$$f = f(x_n, y_i), \tag{4.40}$$

where $x_n$ represents the vector of design variables and $y_i$ is the state variable vector.

For a given vector $x_n$, the solution of the governing equations of the multidisciplinary system yields a vector $y_i$, thus establishing the dependence of the state of the system on the design variables. We denote these governing equations by

$$\mathcal{R}_k\left(x_n, y_i\left(x_n\right)\right) = 0. \tag{4.41}$$

The first instance of $x_n$ in the above equation indicates the fact that the residual of the governing equations may depend *explicitly* on $x_n$. In the case of a structural solver, for example, changing the size of an element has a direct effect on the stiffness matrix. By solving the governing equations we determine the state, $y_i$, which depends *implicitly* on the design variables through the solution of the system. These equations may be non-linear, in which case the usual procedure is to drive residuals, $\mathcal{R}_k$, to zero using an iterative method.

Since the number of equations must equal the number of state variables, the ranges of the indices $i$ and $k$ are the same, i.e., $i, k = 1, \ldots, N_{\mathcal{R}}$. In the case of a structural solver, for example, $N_{\mathcal{R}}$ is the number of degrees of freedom, while for a CFD solver, $N_{\mathcal{R}}$ is the number of mesh points multiplied by the number of state variables at each point. In the more general case of a multidisciplinary system, $\mathcal{R}_k$ represents *all* the governing equations of the different disciplines, including their coupling.

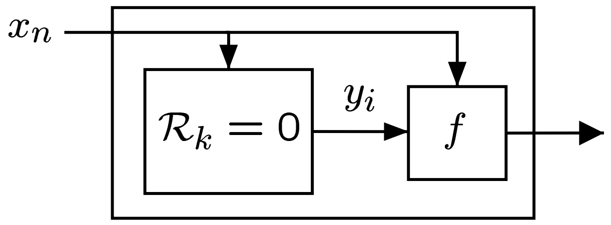


Figure 4.18: Schematic representation of the governing equations ($\mathcal{R}_k = 0$), design variables or inputs ($x_j$), state variables ($y_i$) and the function of interest or output ($f$).

A graphical representation of the system of governing equations is shown in Figure 4.18, with the design variables $x_n$ as the inputs and $f$ as the output. The two arrows leading to $f$ illustrate the fact that the objective function typically depends on the state variables and may also be an explicit function of the design variables.

As a first step toward obtaining the derivatives that we ultimately want to compute, we use the chain rule to write the total sensitivity of $f$ as

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} + \frac{\partial f}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n}, \tag{4.42}$$

for $i = 1, \ldots, N_{\mathcal{R}}$, $n = 1, \ldots, N_x$. Index notation is used to denote the vector dot products. It is important to distinguish the total and partial derivatives in this equation. The partial derivatives can be directly evaluated by varying the denominator and re-evaluating the function in the numerator. The total derivatives, however, require the solution of the multidisciplinary problem. Thus, all the terms in the total sensitivity equation (4.42) are easily computed except for $\mathrm{d}y_i / \mathrm{d}x_n$.

Since the governing equations must always be satisfied, the total derivative of the residuals (4.41) with respect to any design variable must also be zero. Expanding the total derivative of the governing equations with respect to the design variables we can write,

$$\frac{\mathrm{d}\mathcal{R}_k}{\mathrm{d}x_n} = \frac{\partial \mathcal{R}_k}{\partial x_n} + \frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = 0, \tag{4.43}$$

for all $i, k = 1, \ldots, N_{\mathcal{R}}$ and $n = 1, \ldots, N_x$. This expression provides the means for computing the total sensitivity of the state variables with respect to the design variables. By rewriting equation (4.43) as

$$\frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = -\frac{\partial \mathcal{R}_k}{\partial x_n}. \tag{4.44}$$

We can solve for $\mathrm{d}y_i/\,\mathrm{d}x_n$ and substitute this result into the total derivative equation (4.42), to obtain

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} - \underbrace{\frac{\partial f}{\partial y_i} \overbrace{\left[\frac{\partial \mathcal{R}_k}{\partial y_i}\right]^{-1}}_{\Psi_k} \frac{\partial \mathcal{R}_k}{\partial x_n}}^{-\,\mathrm{d}y_i/\,\mathrm{d}x_n}. \tag{4.45}$$

The inverse of the Jacobian $\partial \mathcal{R}_k/\partial y_i$ is not necessarily explicitly calculated. In the case of large iterative problems neither this matrix nor its factorization are usually stored due to their prohibitive size.

### 4.6.3 Direct Sensitivity Equations

The approach where we first calculate $\mathrm{d}y_i/\,\mathrm{d}x_n$ using equation (4.44) and then substitute the result in the expression for the total sensitivity (4.45) is called the *direct* method. Note that solving for $\mathrm{d}y_i/\,\mathrm{d}x_n$ requires the solution of the matrix equation (4.44) *for each design variable $x_n$*. A change in the design variable affects only the right-hand side of the equation, so for problems where the matrix $\partial \mathcal{R}_k/\partial y_i$ can be explicitly factorized and stored, solving for multiple right-hand-side vectors by back substitution would be relatively inexpensive. However, for large iterative problems — such as the ones encountered in CFD — the matrix $\partial \mathcal{R}_k/\partial y_i$ is never factorized explicitly and the system of equations requires an iterative solution which is usually as costly as solving the governing equations. When we multiply this cost by the number of design variables, the total cost for calculating the sensitivity vector may become unacceptable.

### 4.6.4 Adjoint Sensitivity Equations

Returning to the total sensitivity equation (4.45), we observe that there is an alternative option for computing the total sensitivity $\mathrm{d}f/\,\mathrm{d}x_n$. The auxiliary vector $\Psi_k$ can be obtained by solving the *adjoint equations*

$$\frac{\partial \mathcal{R}_k}{\partial y_i}\Psi_k = -\frac{\partial f}{\partial y_i}. \tag{4.46}$$

The vector $\Psi_k$ is usually called the *adjoint vector* and is substituted into equation (4.45) to find the total sensitivity. In contrast with the direct method, the adjoint vector does not depend on the design variables, $x_n$, but instead depends on the function of interest, $f$.

### 4.6.5 Direct vs. Adjoint

We can now see that the choice of the solution procedure (direct vs. adjoint) to obtain the total sensitivity (4.45) has a substantial impact on the cost of sensitivity analysis. Although all the partial derivative terms are the same for both the direct and adjoint methods, the order of the operations is not. Notice that once $\mathrm{d}y_i/\,\mathrm{d}x_n$ is computed, it is valid for any function $f$, but must be recomputed for each design variable (direct method). On the other hand, $\Psi_k$ is valid for all design variables, but must be recomputed for each function (adjoint method).

The cost involved in calculating sensitivities using the adjoint method is therefore practically independent of the number of design variables. After having solved the governing equations, the adjoint equations are solved only once for each $f$. Moreover, the cost of solution of the adjoint equations is similar to that of the solution of the governing equations since they are of similar complexity and the partial derivative terms are easily computed.

| Step | Direct | Adjoint |
|---|---|---|
| Factorization | same | same |
| Back-solve | $N_x$ times | $N_f$ times |
| Multiplication | same | same |

Therefore, if the number of design variables is greater than the number of functions for which we seek sensitivity information, the adjoint method is computationally more efficient. Otherwise, if the number of functions to be differentiated is greater than the number of design variables, the direct method would be a better choice.

A comparison of the cost of computing sensitivities with the direct versus adjoint methods is shown in Table ??. With either method, we must factorize the same matrix, $\partial \mathcal{R}_k/\partial y_i$. The difference in the cost comes form the back-solve step for solving equations (4.44) and (4.59) respectively. The direct method requires that we perform this step for each design variable (i.e. for each $j$) while the adjoint method requires this to be done for each function of interest (i.e. for each $i$). The multiplication step is simply the calculation of the final sensitivity expressed in equations (4.44) and (4.59) respectively. The cost involved in this step when computing the same set of sensitivities is the same for both methods.

In this discussion, we have assumed that the governing equations have been discretized. The same kind of procedure can be applied to continuous governing equations. The principle is the same, but the notation would have to be more general. The equations, in the end, have to be discretized in order to be solved numerically. Figure 4.19 shows the two ways of arriving at the discrete sensitivity equations. We can either differentiate the continuous governing equations first and then discretize them, or discretize the governing equations and differentiate them in the second step. The resulting sensitivity equations should be equivalent, but are not necessarily the same.

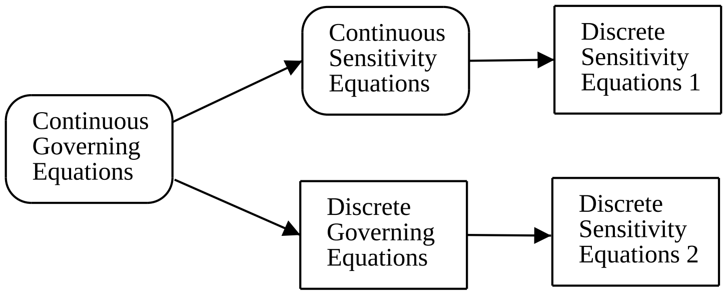

Continuous Governing Equations
Continuous Sensitivity Equations
Discrete Sensitivity Equations 1
Discrete Governing Equations
Discrete Sensitivity Equations 2

Figure 4.19: The two ways of obtaining the discretized sensitivity equations

Differentiating the continuous governing equations first is usually more involved. In addition, applying boundary conditions to the differentiated equations can be non-intuitive as some of these boundary conditions are non-physical.

### 4.6.6 Example: Structural Sensitivity Analysis

The discretized governing equations for a finite-element structural model are,

$$\mathcal{R}_k = K_{ki}u_i - F_k = 0, \tag{4.47}$$

where $K_{ki}$ is the stiffness matrix, $u_i$ is the vector of displacement (the state) and $F_k$ is the vector of applied force (not to be confused with the function of interest from the previous section!).

We are interested in finding the sensitivities of the stress, which is related to the displacements by the equation,

$$\sigma_m = S_{mi} u_i. \tag{4.48}$$

We will consider the design variables to be the cross-sectional areas of the elements, $A_j$. We will now look at the terms that we need to use the generalized total sensitivity equation (4.45).

For the matrix of sensitivities of the governing equations with respect to the state variables we find that it is simply the stiffness matrix, i.e.,

$$\frac{\partial \mathcal{R}_k}{\partial y_i} = \frac{\partial (K_{ki} u_i - F_k)}{\partial u_i} = K_{ki}. \tag{4.49}$$

Let's consider the sensitivity of the residuals with respect to the design variables (cross-sectional areas in our case). Neither the displacements of the applied forces vary explicitly with the element sizes. The only term that depends on $A_j$ directly is the stiffness matrix, so we get,

$$\frac{\partial \mathcal{R}_k}{\partial x_j} = \frac{\partial (K_{ki} u_i - F_k)}{\partial A_j} = \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.50}$$

The partial derivative of the stress with respect to the displacements is simply given by the matrix in equation (4.48), i.e.,

$$\frac{\partial f_m}{\partial y_i} = \frac{\partial \sigma_m}{\partial u_i} = S_{mi} \tag{4.51}$$

Finally, the explicit variation of stress with respect to the cross-sectional areas is zero, since the stresses depends only on the displacement field,

$$\frac{\partial f_m}{\partial x_j} = \frac{\partial \sigma_m}{\partial A_j} = 0. \tag{4.52}$$

Substituting these into the generalized total sensitivity equation (4.45) we get:

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = -\frac{\partial \sigma_m}{\partial u_i} K_{ki}^{-1} \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.53}$$

Referring to the theory presented previously, if we were to use the direct method, we would solve,

$$K_{ki} \frac{\mathrm{d}u_i}{\mathrm{d}A_j} = -\frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.54}$$

and then substitute the result in,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = \frac{\partial \sigma_m}{\partial u_i} \frac{\mathrm{d}u_i}{\mathrm{d}A_j} \tag{4.55}$$

to calculate the desired sensitivities.

The adjoint method could also be used, in which case we would solve equation (4.59) for the structures case,

$$K_{ki}^T \psi_k = \frac{\partial \sigma_m}{\partial u_i}. \tag{4.56}$$

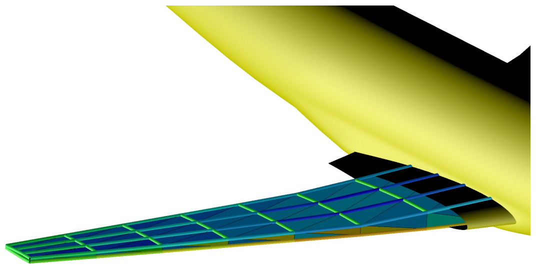
Then we would substitute the adjoint vector into the equation,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = \frac{\partial \sigma_m}{\partial A_j} + \psi_k^T \left( -\frac{\partial K_{ki}}{\partial A_j} u_i \right). \tag{4.57}$$

to calculate the desired sensitivities.

**Example 4.19.** Computational Accuracy and Cost

| Method | Sample Sensitivity | Time | Memory |
|---|---|---|---|
| Complex | –39.049760045804646 | 1.00 | 1.00 |
| ADIFOR | –39.049760045809059 | 2.33 | 8.09 |
| Analytic | –39.049760045805281 | 0.58 | 2.42 |
| FD | –39.049724352820375 | 0.88 | 0.72 |



- All except finite-difference achieve the solver's precision.
- Analytic: best, but not easy to implement.
- ADIFOR: costly.
- Complex-step: good compromise.
- Caveat: ratios depend on problem.

**Example 4.20.** The Automatic Differentiation Adjoint (ADjoint) Method [? ]

The automatic differentiation adjoint (ADjoint) method is a hybrid between automatic differentiation and the adjoint method. In a nutshell, we take the adjoint equations formulated above, compute the partial derivatives in those equations with automatic differentiation, and then solve the linear system and perform the necessary matrix-vector products.

We chose to use Tapenade as it is the only non-commercial tool with support for Fortran 90. Tapenade is the successor of Odyssée [7] and was developed at the INRIA. It uses source transformation and can perform differentiation in either forward or reverse mode. Furthermore, the Tapenade team is actively developing their software and has been very responsive to a number of suggestions towards completing their support of the Fortran 90 standard.

In order to verify the results given by the ADjoint approach we decided to use the complex-step derivative approximation [13, 15] as our benchmark.

In semi-discrete form the Euler equations are,

$$\frac{\mathrm{d}w_{ijk}}{\mathrm{d}t} + \mathcal{R}_{ijk}(w) = 0, \tag{4.58}$$

where $\mathcal{R}$ is the residual described earlier with all of its components (fluxes, boundary conditions, artificial dissipation, etc.).

The adjoint equations (**??**) can be re-written for this flow solver as,

$$\left[\frac{\partial \mathcal{R}}{\partial w}\right]^T \psi = -\frac{\partial I}{\partial w}. \tag{4.59}$$

where $\psi$ is the *adjoint vector*. The total sensitivity (4.42) in this case is,

$$\frac{\mathrm{d}I}{\mathrm{d}x} = \frac{\partial I}{\partial x} + \psi^T \frac{\partial \mathcal{R}}{\partial x}. \tag{4.60}$$

We propose to compute the partial derivative matrices $\partial\mathcal{R}/\partial w$, $\partial I/\partial w$, $\partial I/\partial x$ and $\partial\mathcal{R}/\partial x$ using automatic differentiation instead of using manual differentiation or using finite differences. Where appropriate we will use the reverse mode of automatic differentiation.

To better understand the choices made in the mode of differentiation as well as the effect of these choices in the general case we define the following numbers:

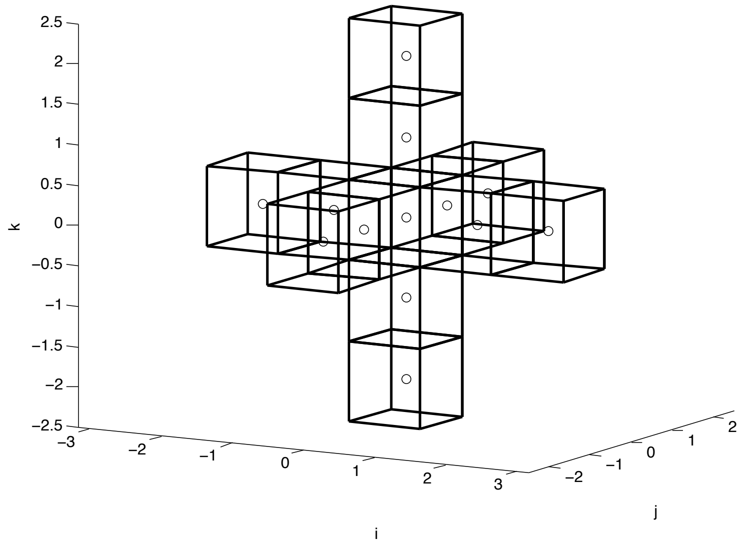


Figure 4.20: Stencil for the residual computation

```
SUBROUTINE RESIDUAL(w, r, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), w(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  INTRINSIC SQRT
  DO i=1,ni
    DO j=1,nj
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
      r(i, j) = w(i, j)*w(i-1, j) + SQRT(w(i+1, j)) + &
&                w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL
```

Figure 4.21: Simplified subroutine for residual calculation

$N_c$**:** The number of cells in the domain. For three-dimensional domains where the Navier–Stokes equations are solved, this can be $\mathcal{O}(10^6)$.

$N_s$**:** The number of cells in the stencil whose variables affect the residual of a given cell. In our case, we consider inviscid and dissipation fluxes, so the stencil is as shown in Figure 4.20 and $N_s = 13$.

$N_w$**:** The number of flow variables (and also residuals) for each cell. In our case $N_w = 5$.

In principle, because $\partial R/\partial w$ is a square matrix, neither mode should have an advantage over the other in terms of computational time. However, due to the way residuals are computed, the reverse mode is much more efficient. To explain the reason for this, consider a very simplified version of a calculation resembling the residual calculation in CFD shown in Figure 4.21. This subroutine loops through a two-dimensional domain and computes `r` (the "residual") in the interior of that domain. The residual at any cell depends only on the `w`s (the "flow variables") at that cell and at the cells immediately adjacent to it, thus the stencil of dependence forms a cross with five cells.

```
SUBROUTINE RESIDUAL_D(w, wd, r, rd, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rd(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wd(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: arg1, arg1d, result1, result1d
  INTRINSIC SQRT

  rd(1:ni, 1:nj) = 0.0
  DO i=1,ni
    DO j=1,nj
      arg1d = wd(i, j-1)*w(i, j+1) + w(i, j-1)*wd(i, j+1)
      arg1 = w(i, j-1)*w(i, j+1)
      IF (arg1d .EQ. 0.0 .OR. arg1 .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = arg1d/(2.0*SQRT(arg1))
      END IF
      result1 = SQRT(arg1)
      wd(i, j) = wd(i, j) + result1d
      w(i, j) = w(i, j) + result1
      IF (wd(i+1, j) .EQ. 0.0 .OR. w(i+1, j) .EQ. 0.0) THEN
        result1d = 0.0
      ELSE
        result1d = wd(i+1, j)/(2.0*SQRT(w(i+1, j)))
      END IF
      result1 = SQRT(w(i+1, j))
      rd(i, j) = wd(i, j)*w(i-1, j) + w(i, j)*wd(i-1, j) + &
&                 result1d + wd(i, j-1)*w(i, j+1) + &
&                 w(i, j-1)*wd(i, j+1)
      r(i, j) = w(i, j)*w(i-1, j) + result1 + &
&                w(i, j-1)*w(i, j+1)
    END DO
  END DO
END SUBROUTINE RESIDUAL_D
```

Figure 4.22: Subroutine differentiated using the forward mode

The residual computation in our three-dimensional CFD solver is obviously much more complicated: It involves multiple subroutines, a larger stencil, the computation of the different fluxes and applies many different types of boundary conditions. However, this simple example is sufficient to demonstrate the computational inefficiencies of a purely automatic approach.

Forward mode differentiation was used to produce the subroutine shown in Figure 4.22. Two new variables are introduced: `wd`, which is the seed vector and `rd`, which is the gradient of all `r`s in the direction specified by the seed vector. For example, if we want the derivative with respect to `w(1,1)`, we would set `wd(1,1)=1` and all other `wd`s to zero. One can only choose one direction at a time, although Tapenade can be run in "vectorial" mode to get the whole vector of sensitivities. In this case, an additional loop inside the nested loop and additional storage are required. For our purposes, the differentiated code would have to be called $N_c \times N_w$ times.

The subroutine produced by reverse mode differentiation is shown in Figure 4.23. This case shows the additional storage requirements: Since the `w`s are overwritten, the old values must be stored for later use in the reversed loop.

The overwriting of the flow variables in the first nested loops of the original subroutine is characteristic of iterative solvers. Whenever overwriting is present, the reverse mode needs to store the time history of the intermediate variables. Tapenade provides functions (`PUSHREAL` and `POPREAL`) to do this. In this case, we can see that the `w`s are stored before they are modified in the

```
SUBROUTINE RESIDUAL_B(w, wb, r, rb, ni, nj)
  IMPLICIT NONE
  INTEGER :: ni, nj
  REAL :: r(ni, nj), rb(ni, nj)
  REAL :: w(0:ni+1, 0:nj+1), wb(0:ni+1, 0:nj+1)
  INTEGER :: i, j
  REAL :: tempb
  INTRINSIC SQRT
  DO i=1,ni
    DO j=1,nj
      CALL PUSHREAL4(w(i, j))
      w(i, j) = w(i, j) + SQRT(w(i, j-1)*w(i, j+1))
    END DO
  END DO
  wb(0:ni+1, 0:nj+1) = 0.0
  DO i=ni,1,-1
    DO j=nj,1,-1
      wb(i, j) = wb(i, j) + w(i-1, j)*rb(i, j)
      wb(i-1, j) = wb(i-1, j) + w(i, j)*rb(i, j)
      wb(i+1, j) = wb(i+1, j) + rb(i, j)/(2.0*SQRT(w(i+1, j)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*rb(i, j)
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*rb(i, j)
      rb(i, j) = 0.0
      CALL POPREAL4(w(i, j))
      tempb = wb(i, j)/(2.0*SQRT(w(i, j-1)*w(i, j+1)))
      wb(i, j-1) = wb(i, j-1) + w(i, j+1)*tempb
      wb(i, j+1) = wb(i, j+1) + w(i, j-1)*tempb
    END DO
  END DO
END SUBROUTINE RESIDUAL_B
```

Figure 4.23: Subroutine differentiated using the reverse mode

forward sweep, and then retrieved in the reverse sweep.

On this basis, reverse mode was used to produce adjoint code for this set of routines. The call graph for the original and the differentiated routines are shown in Figures 4.24 and 4.25, respectively.

The test case is the Lockheed-Air Force-NASA-NLR (LANN) wing [21], which is a supercritical transonic transport wing. A symmetry boundary condition is used at the root and a linear pressure extrapolation boundary condition is used on the wing surface. The freestream Mach number is 0.621.

The meshes for this test case is shown in Figure **??**.

In Table 4.2 we show the sensitivities of both drag and lift coefficients with respect to freestream Mach number for different cases. The wing mesh and the fine mesh cases are the two test cases described earlier (Figures **??** and **??**). The "coarse" case is a very small mesh ($12 \times 4 \times 4$) with the same geometry as the fine bump case.

We can see that the adjoint sensitivities for these cases are extremely accurate, yielding between twelve and fourteen digits agreement when compared to the complex-step results. This is consistent with the convergence tolerance that was specified in PETSc for the adjoint solution.

To analyze the performance of the ADjoint solver, several timings were performed. They are shown in Table 4.3 for the three cases mentioned above. The coarse grid has $5,120$ flow variables, the fine grid has $203,840$ flow variables and the wing grid has $108,800$ flow variables.

The total cost of the adjoint solver, including the computation of all the partial derivatives and the solution of the adjoint system, is less than one fourth the cost of the flow solution for the fine bump case and less that one eighth the cost of the flow solution for the wing case. This is even

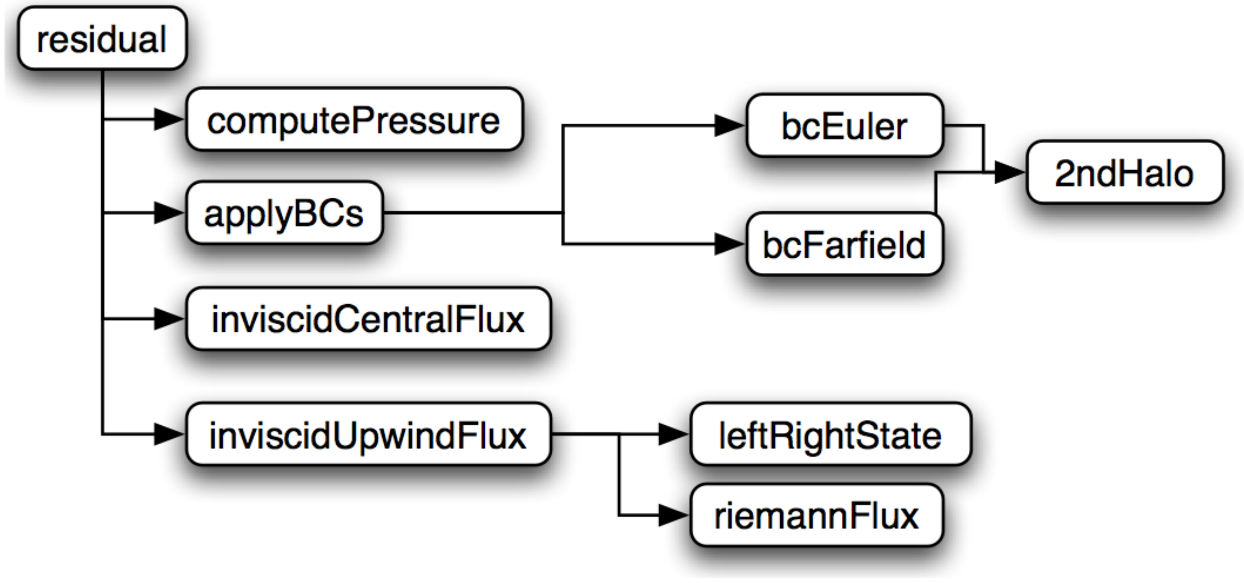


Figure 4.24: Original residual calculation

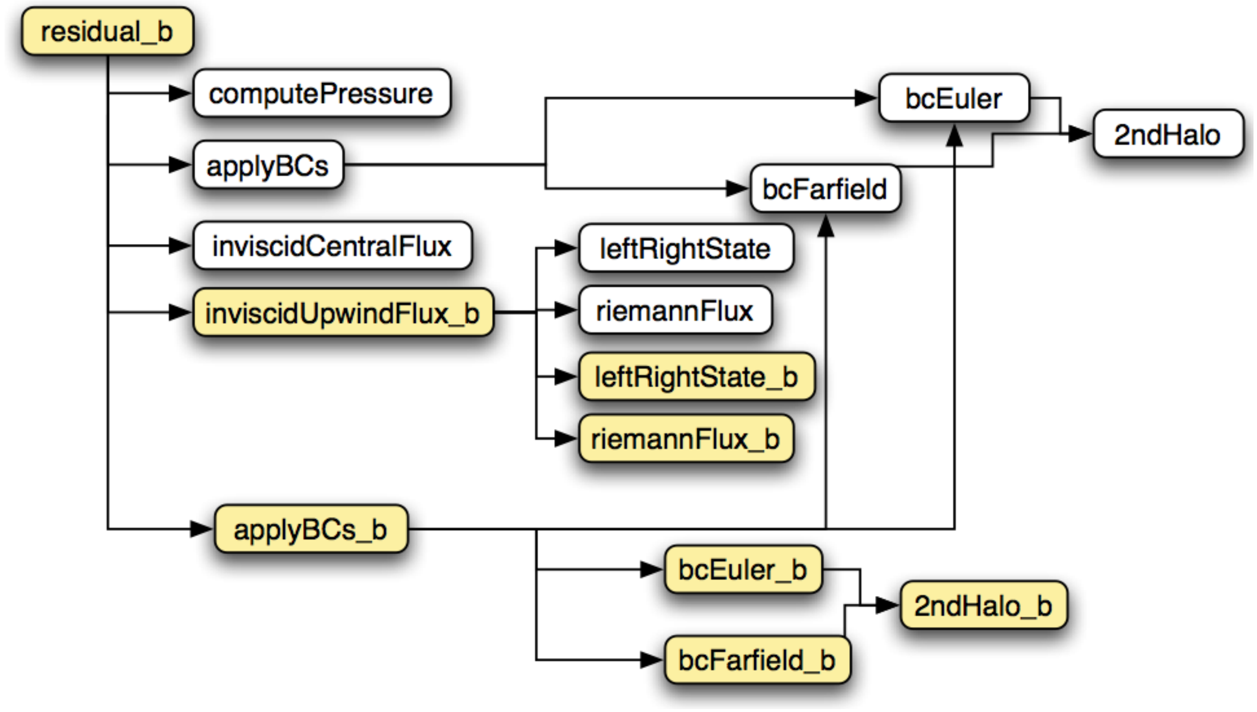


Figure 4.25: Differentiated residual calculation

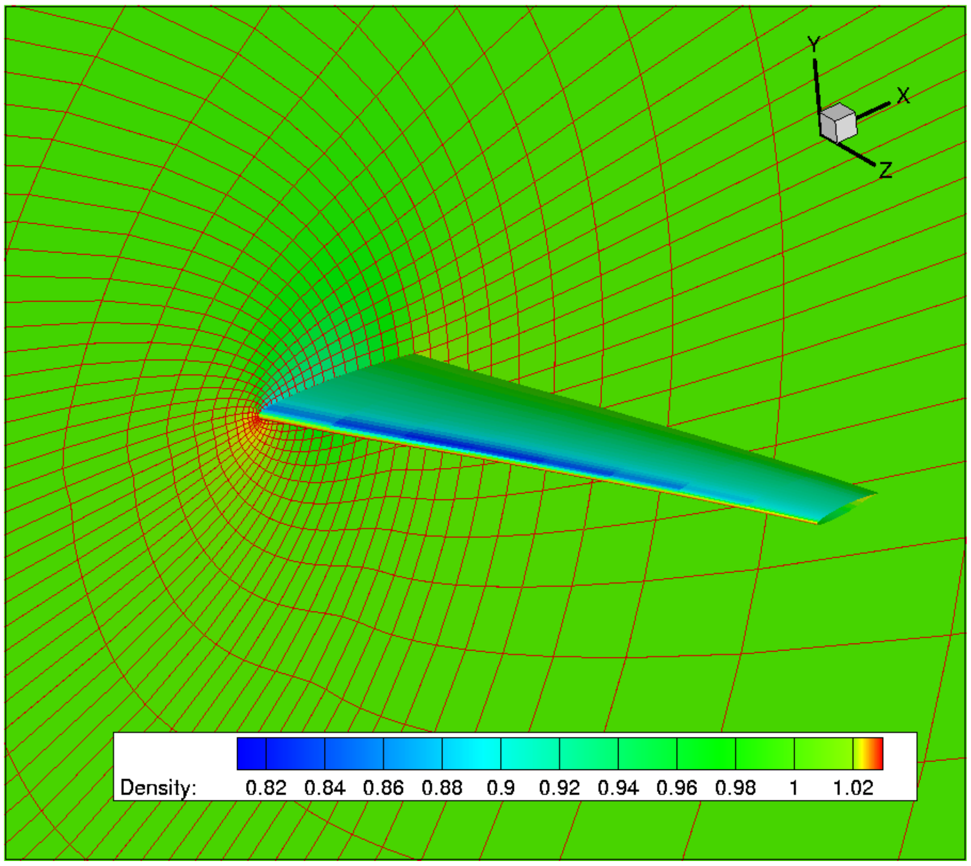


Figure 4.26: Wing mesh and density distribution

Table 4.2: Sensitivities of drag and lift coefficients with respect to $M_\infty$

| Mesh | Coefficient | Inflow direction | ADjoint | Complex step |
|---|---|---|---:|---:|
| Coarse | $C_D$ | (1,0,0) | -0.2898966327317*64* | -0.2898966327317*59* |
| | $C_L$ | | -0.267704455366*714* | -0.267704455366*683* |
| Fine | $C_D$ | (1,0,0) | -0.027950118302470*5* | -0.027950118302470*9* |
| | $C_L$ | | 0.5812860473470*7* | 0.5812860473470*8* |
| Coarse | $C_D$ | (1,0.05,0) | -0.2789076458337*86* | -0.2789076458337*92* |
| | $C_L$ | | -0.262086315233*911* | -0.262086315233*875* |
| Fine | $C_D$ | (1,0.05,0) | -0.06155986310604*38* | -0.06155986310604*44* |
| | $C_L$ | | -0.3647967546527*87* | -0.3647967546527*97* |
| Wing | $C_D$ | ( 1, 0.0102,0) | 0.00942875710535*217* | 0.00942875710535*312* |
| | $C_L$ | | 0.267882125954*74* | 0.267882125954*68* |

better than what is usually observed in the case of adjoint solvers developed by conventional means, showing that the ADjoint approach is indeed very efficient.

Table 4.3: ADjoint computational cost breakdown (times in seconds)

| | Fine | Wing |
|---|---:|---:|
| **Flow solution** | 219.215 | 182.653 |
| **ADjoint** | 51.959 | 20.843 |
| Breakdown: | | |
| Setup PETSc variables | 0.011 | 0.004 |
| Compute flux Jacobian | 11.695 | 5.870 |
| Compute RHS | 8.487 | 2.232 |
| Solve the adjoint equations | 28.756 | 11.213 |
| Compute the total sensitivity | 3.010 | 1.523 |

## Bibliography

[1] Tools for automatic differentiation. URL http://www.sc.rwth-aachen.de/Research/AD/subject.html.

[2] Thomas Beck. Automatic differentiation of iterative processes. *Journal of Computational and Applied Mathematics*, 50:109–118, 1994.

[3] Thomas Beck and Herbert Fischer. The if-problem in automatic differentiation. *Journal of Computational and Applied Mathematics*, 50:119–131, 1994.

[4] Claus Bendtsen and Ole Stauning. FADBAD, a flexible C++ package for automatic differentiation — using the forward and backward methods. Technical Report IMM-REP-1996-17, Technical University of Denmark, DK-2800 Lyngby, Denmark, 1996. URL citeseer.nj.nec.com/bendtsen96fadbad.html.

[5] C. H. Bischof, L. Roh, and A. J. Mauer-Oats. ADIC: an extensible automatic differentiation tool for ANSI-C. *Software — Practice and Experience*, 27(12):1427–1456, 1997. URL citeseer.nj.nec.com/article/bischof97adic.html.

[6] Alan Carle and Mike Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.

[7] C. Faure and Y. Papegay. *Odyssée Version 1.6: The Language Reference Manual*. INRIA, 1997. Rapport Technique 211.

[8] Ralf Giering and Thomas Kaminski. Applying TAF to generate efficient derivative code of Fortran 77-95 programs. In *Proceedings of GAMM 2002, Augsburg, Germany*, 2002.

[9] Mark S. Gockenbach. Understanding Code Generated by TAMC. IAAA Paper TR00-29, Department of Computational and Applied Mathematics, Rice University, Texas, USA, 2000. URL http://www.math.mtu.edu/~msgocken.

[10] Andreas Griewank. *Evaluating Derivatives*. SIAM, Philadelphia, 2000.

[11] L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004. URL http://www.inria.fr/rrrt/rt-0300.html.

[12] J. N. Lyness. Numerical algorithms based on the theory of complex variable. In *Proceedings — ACM National Meeting*, pages 125–133, Washington DC, 1967. Thompson Book Co.

[13] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967. ISSN 0036-1429 (print), 1095-7170 (electronic).

[14] Joaquim R. R. A. Martins. A guide to the complex-step derivative approximation, 2003. URL http://mdolab.utias.utoronto.ca/resources/complex-step/.

[15] Joaquim R. R. A. Martins, Peter Sturdza, and Juan J. Alonso. The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3):245–262, 2003. doi: 10.1145/838250.838251.

[16] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004. doi: 10.2514/1.11478.

[17] Joaquim R. R. A. Martins, Juan J. Alonso, and James J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1): 33–62, March 2005. doi: 10.1023/B:OPTE.0000048536.47956.62.

[18] Siva Nadarajah and Antony Jameson. A comparison of the continuous and discrete adjoint approach to automatic aerodynamic optimization. In *Proceedings of the 38th AIAA Aerospace Sciences Meeting and Exhibit*, Reno, NV, 2000. AIAA 2000-0667.

[19] V. Pascual and L. Hascoët. Extension of TAPENADE towards Fortran 95. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.

[20] John D. Pryce and John K. Reid. AD01, a Fortran 90 code for automatic differentiation. Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, U.K., 1998.

[21] S. Y. Ruo, J. B. Malone, J. J. Horsten, and R. Houwink. The LANN program — an experimental and theoretical study of steady and unsteady transonic airloads on a supercritical wing. In *Proceedings of the 16th Fluid and PlasmaDynamics Conference*, Danvers, MA, July 1983. AIAA 1983-1686.

[22] William Squire and George Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):110–112, 1998. ISSN 0036-1445 (print), 1095-7200 (electronic). URL http://epubs.siam.org/sam-bin/dbq/article/31241.

[23] Peter Sturdza. *An Aerodynamic Design Method for Supersonic Natural Laminar Flow Aircraft*. PhD thesis, Stanford University, Stanford, CA, December 2003.