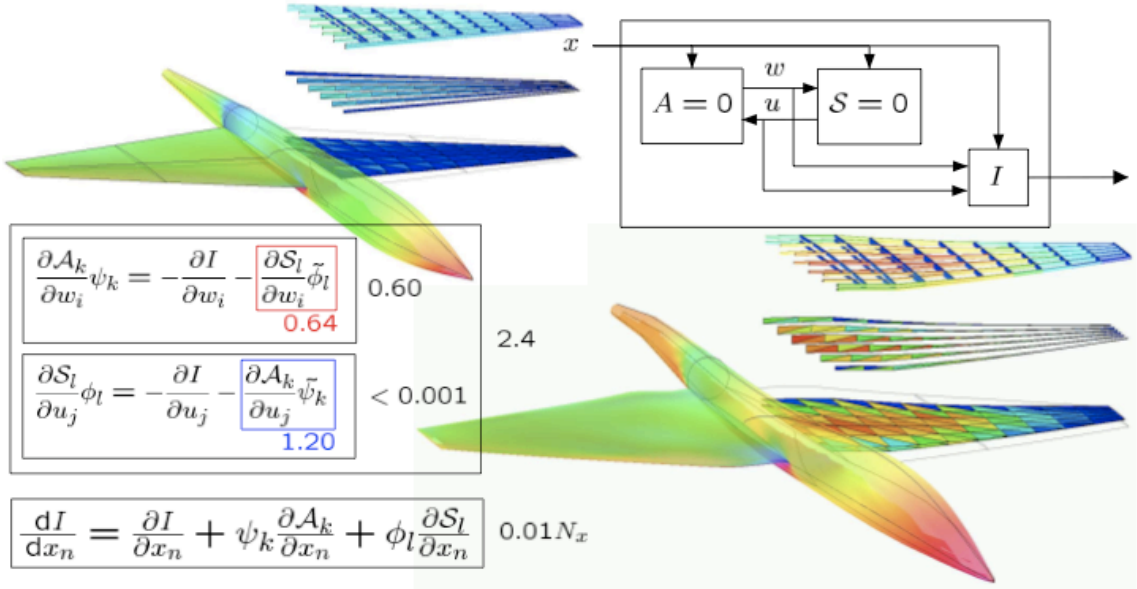
AA222: Sensitivity Analysis



AA222
Lecture 4
April 18, 2010

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:

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

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

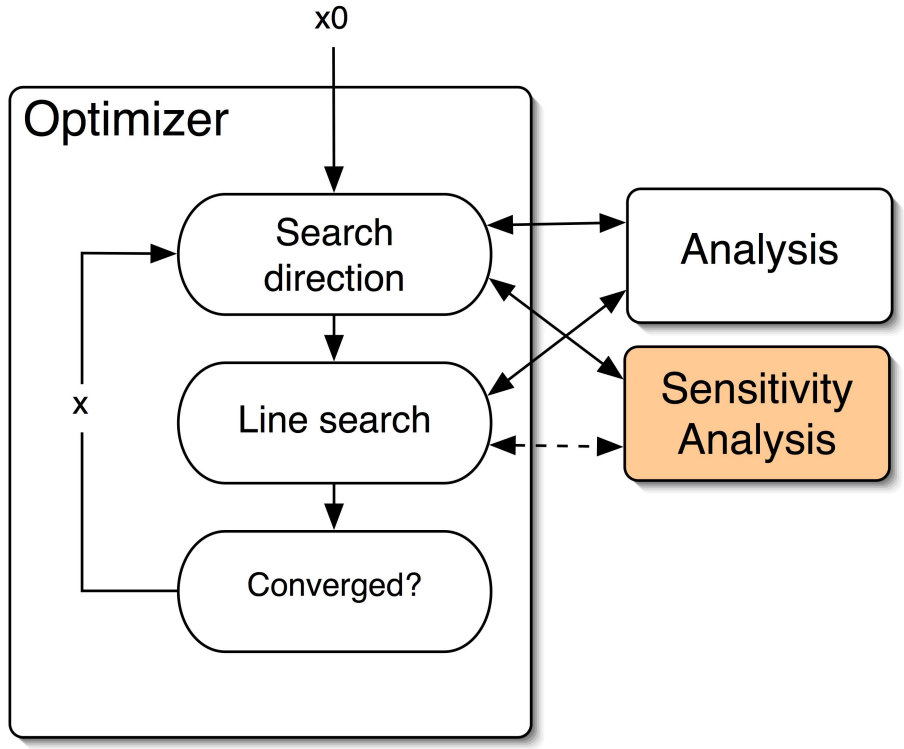
- The sensitivities of the objective function, $\nabla f(x) = \partial f / \partial x_i$ $(n \times 1)$.
- The sensitivities of all the active constraints at the current design point $\partial c_j / \partial x_i$ $(m \times n)$.

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.

$$\boxed{\frac{\mathrm{d}f}{\mathrm{d}x_i} \approx \frac{f(x_i + h) - f(x)}{h} + \mathcal{O}(h)}$$

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

$$\boxed{\frac{\mathrm{d}f}{\mathrm{d}x_i} \approx \frac{\mathrm{Im}\left[f(x_i + ih)\right]}{h} + \mathcal{O}(h^2)}$$

- **Hyper-dual Numbers:** An extension of the complex-step method to compute second and higher derivatives; its use is being pioneered here at Stanford in the ADL.
- **Symbolic Differentiation:** accurate; restricted to explicit functions of low dimensionality.
- **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-differencing 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 $f(x+h)$,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \ldots, \tag{4.1}$$

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

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \tag{4.2}$$

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 $f(x-h)$,

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2!}f''(x) - \frac{h^3}{3!}f'''(x) + \ldots, \tag{4.3}$$

and subtract it from the expansion (4.1). The resulting equation can then be solved for the derivative of f to obtain the *central-difference* formula,

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2). \tag{4.4}$$

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 (4.4) to estimate the second derivative instead of the first,

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

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

$$f''(x) = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2} + \mathcal{O}(h). \tag{4.6}$$

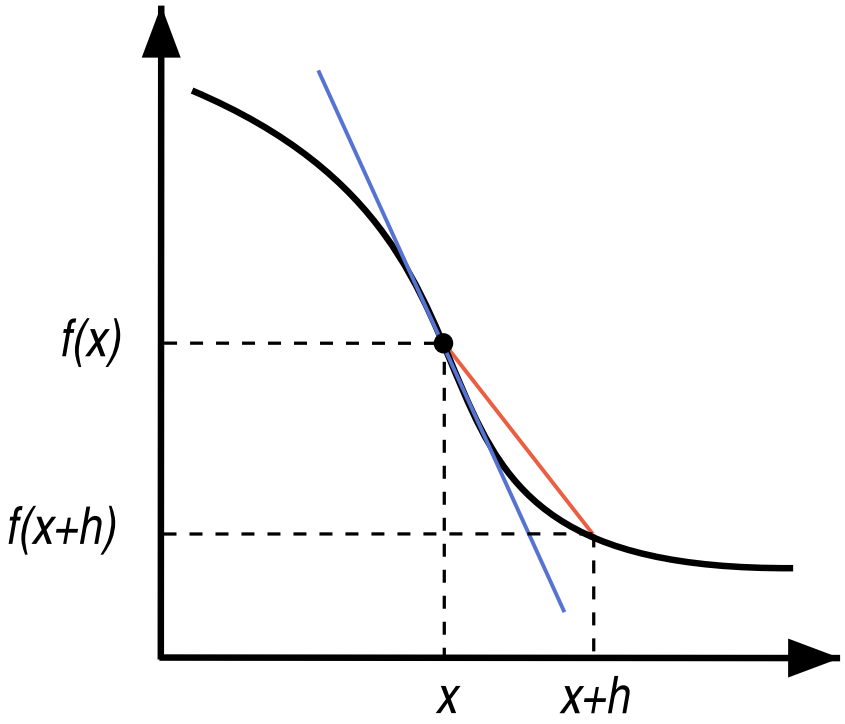
When estimating sensitivities using finite-difference formulae we are faced with the *step-size dilemma*, that is the desire to choose a small step size to minimize truncation error while avoiding the use of a step so small that errors due to subtractive cancellation become dominant.

Forward-difference approximation:

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

With 16-digit arithmetic,

$f(x+h)$	$+1.234567890123431$
$f(x)$	$+1.234567890123456$
Δf	-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 [9] and Lyness [10]. Their work produced several methods that made use of complex variables, including a reliable method for calculating the n^{th} derivative of an analytic function. However, only recently has some of this theory been rediscovered by Squire and Trapp [19] 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 [11, 15, 12].

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,

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \dots \tag{4.7}$$

Taking the imaginary parts of both sides of (4.7) and dividing the equation by h yields

$$f'(x) = \frac{\text{Im}\left[f(x+ih)\right]}{h} + h^2\frac{f'''(x)}{3!} + \ldots \tag{4.8}$$

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.,

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \tag{4.9}$$

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{4.10}$$

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 (4.9) to obtain,

$$\frac{\partial u}{\partial x} = \lim_{h\to 0}\frac{v(x+i(y+h)) - v(x+iy)}{h}. \tag{4.11}$$

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 (4.11) can then be re-written as,

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{\text{Im}\left[f\left(x + ih\right)\right]}{h}. \tag{4.12}$$

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

$$\frac{\partial f}{\partial x} \approx \frac{\text{Im}\left[f\left(x + ih\right)\right]}{h}. \tag{4.13}$$

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 (4.2, 4.4).

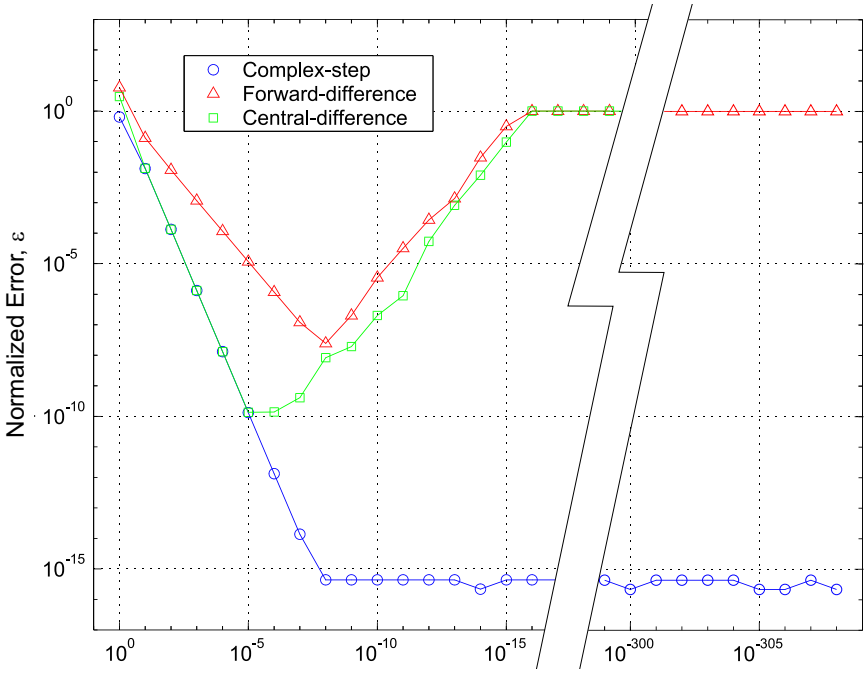
Example 4.1: The Complex-Step Method Applied to a Simple Function

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

$$f(x) = \frac{\mathrm{e}^x}{\sqrt{\sin^3 x + \cos^3 x}} \tag{4.14}$$

The exact derivative at $x = 1.5$ was computed analytically to 16 digits and then compared to the results given by the complex-step (4.13) 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; $\varepsilon = \frac{\left|f' - f'_{ref}\right|}{\left|f'_{ref}\right|}$.



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`

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

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

4.4.4 Can the Complex-Step Method be Improved?

Improvements necessary because,

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

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,

$$iz + z = (x - h) + i\,(x + h)$$

h vanishes when using finite precision arithmetic.

Would like to keep the real and imaginary parts separate.

The complex definition of sine also problematic,

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

The complex trigonometric relation yields a better alternative,

$$\sin(x + ih) = \sin(x)\cosh(h) + i\cos(x)\sinh(h).$$

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

$$\sin(x + ih) \approx \sin(x) + ih\cos(x).$$

From the standard complex definition,

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

Need real and imaginary parts to be calculated separately.

Linearizing in h about $h = 0$,

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

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.

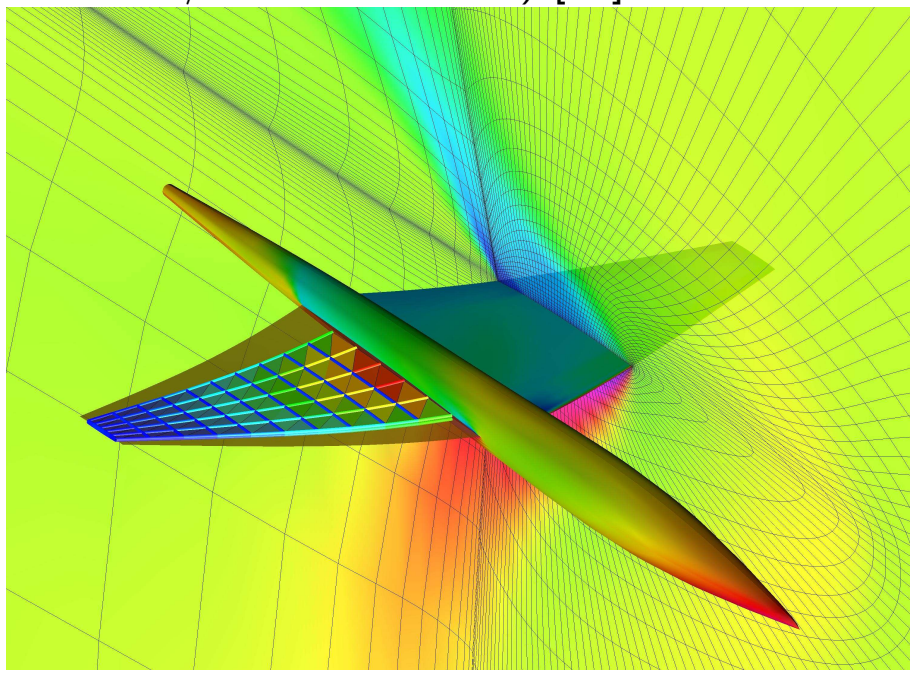
4.4.7 C/C++ Implementations

- `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 4.2: 3D Aero-Structural Design Optimization Framework (Martins, UTIAS / Alonso, Stanford / Reuther, NASA) [13]



- 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

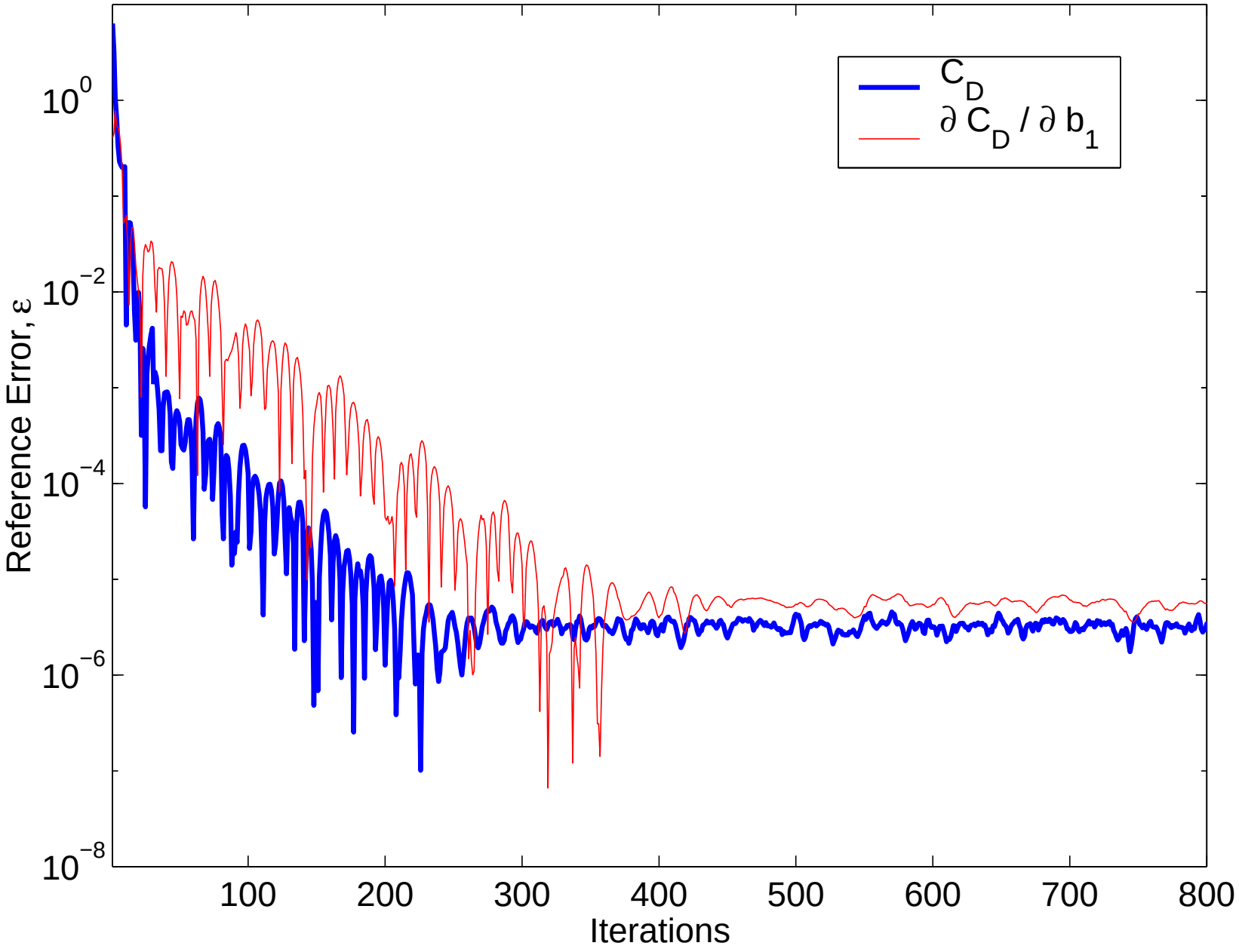


Figure 4.1: Convergence of C_D and $\partial C_D / \partial b_1$

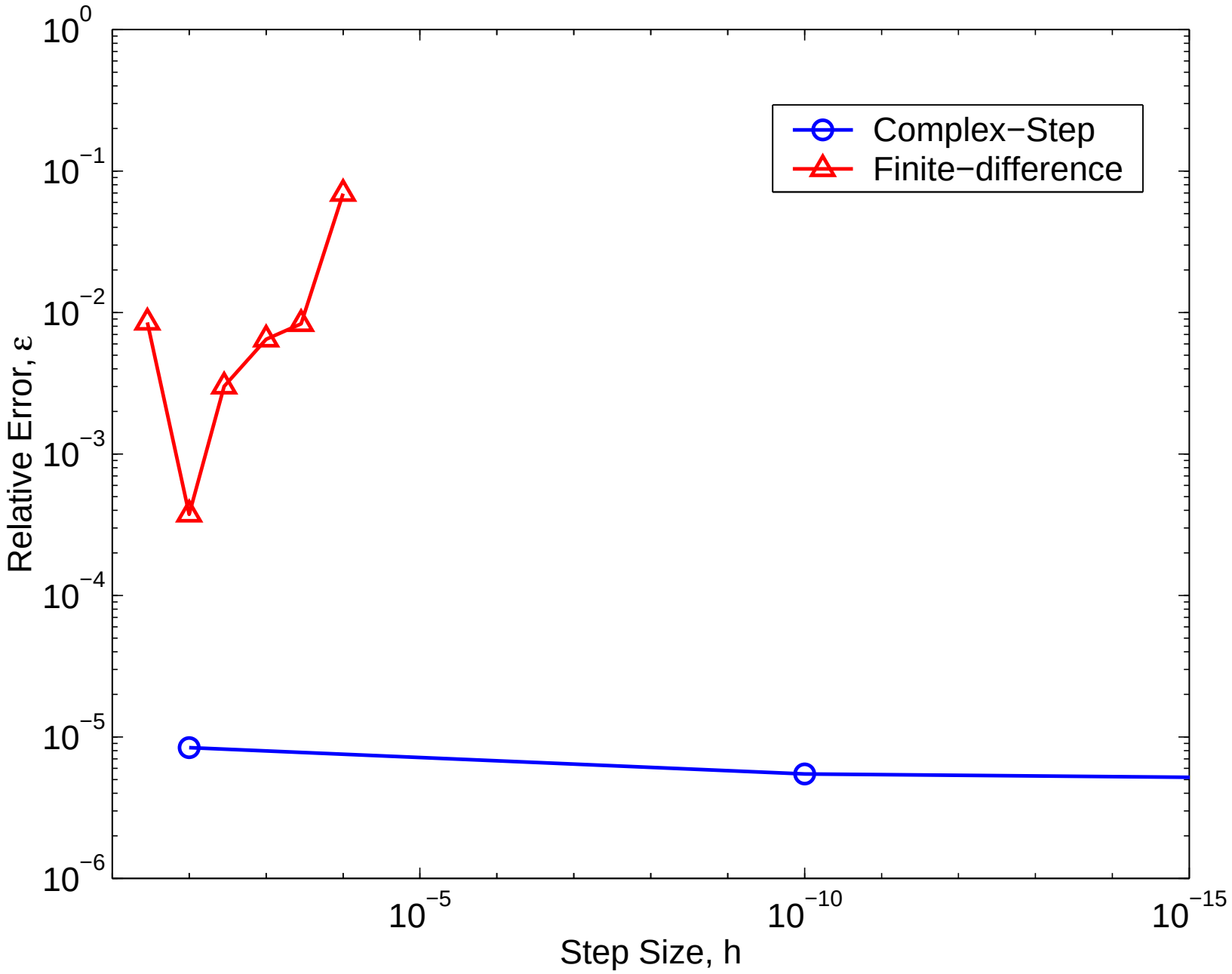


Figure 4.2: Sensitivity estimate vs. step size. Note that the finite-difference results are practically useless.

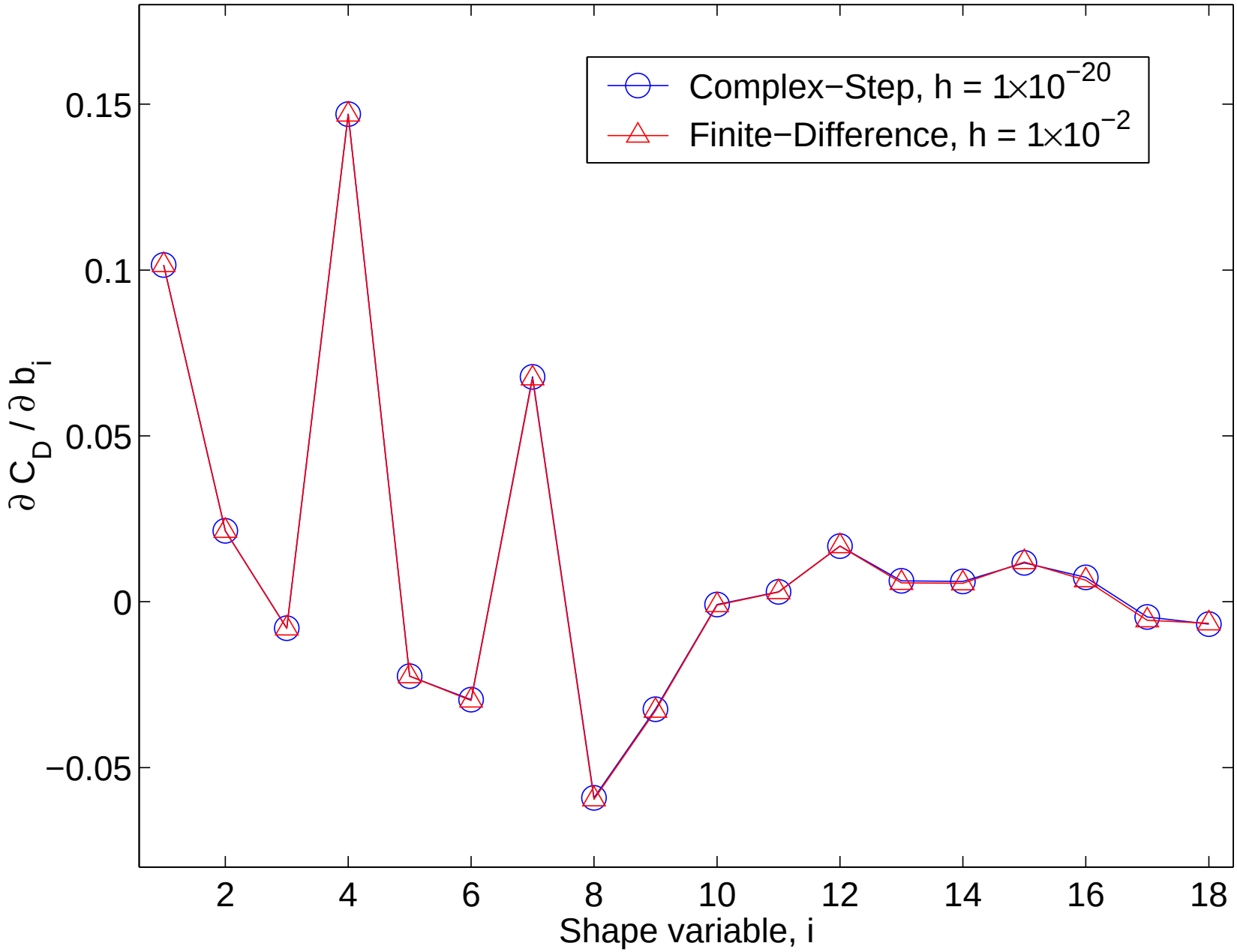
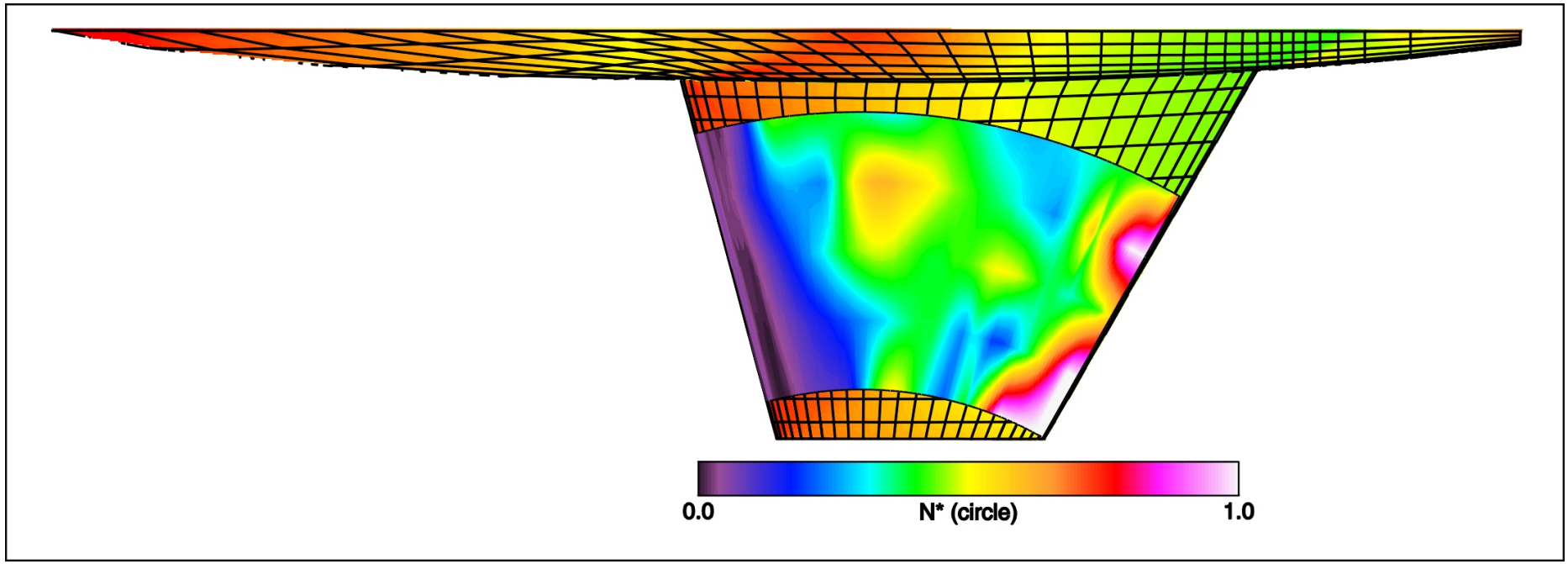
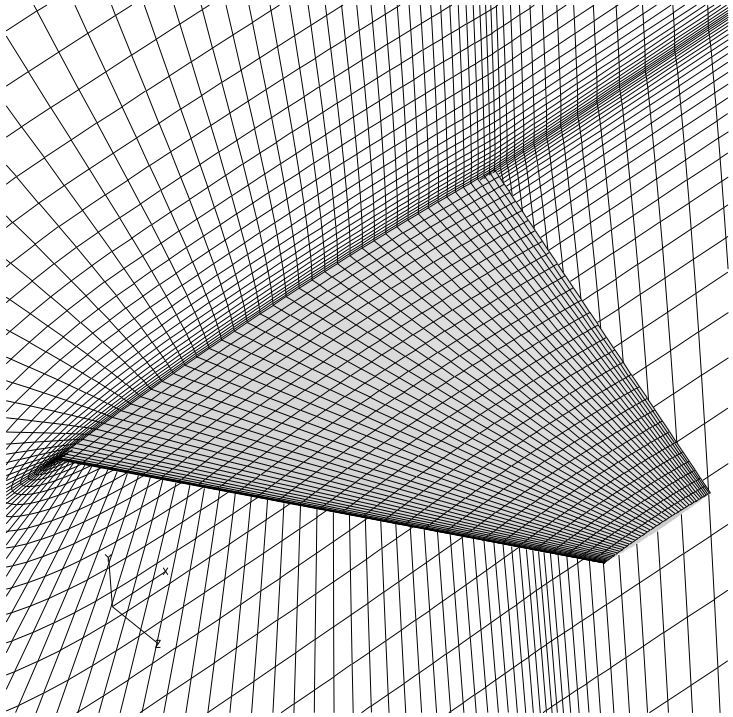


Figure 4.3: Sensitivity of C_D to shape functions. The finite-different results were obtained after much effort to choose the best step.

Example 4.3: Supersonic Viscous/Inviscid Solver (Sturdza and Kroo, Stanford University) [20]
Framework for preliminary design of natural laminar flow supersonic aircraft



- Transition prediction
- 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
  - 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

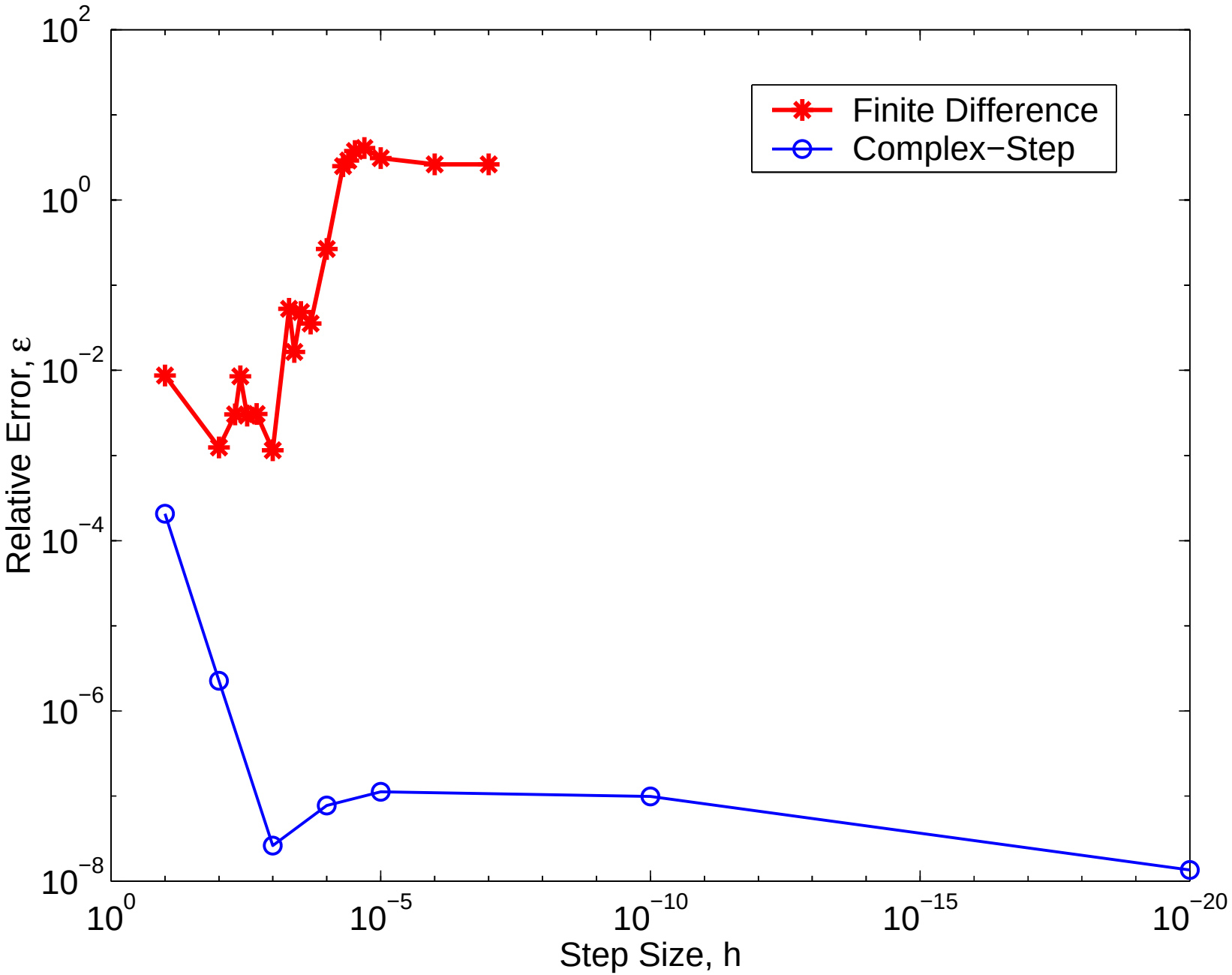


Figure 4.4: Sensitivity estimate vs. step size

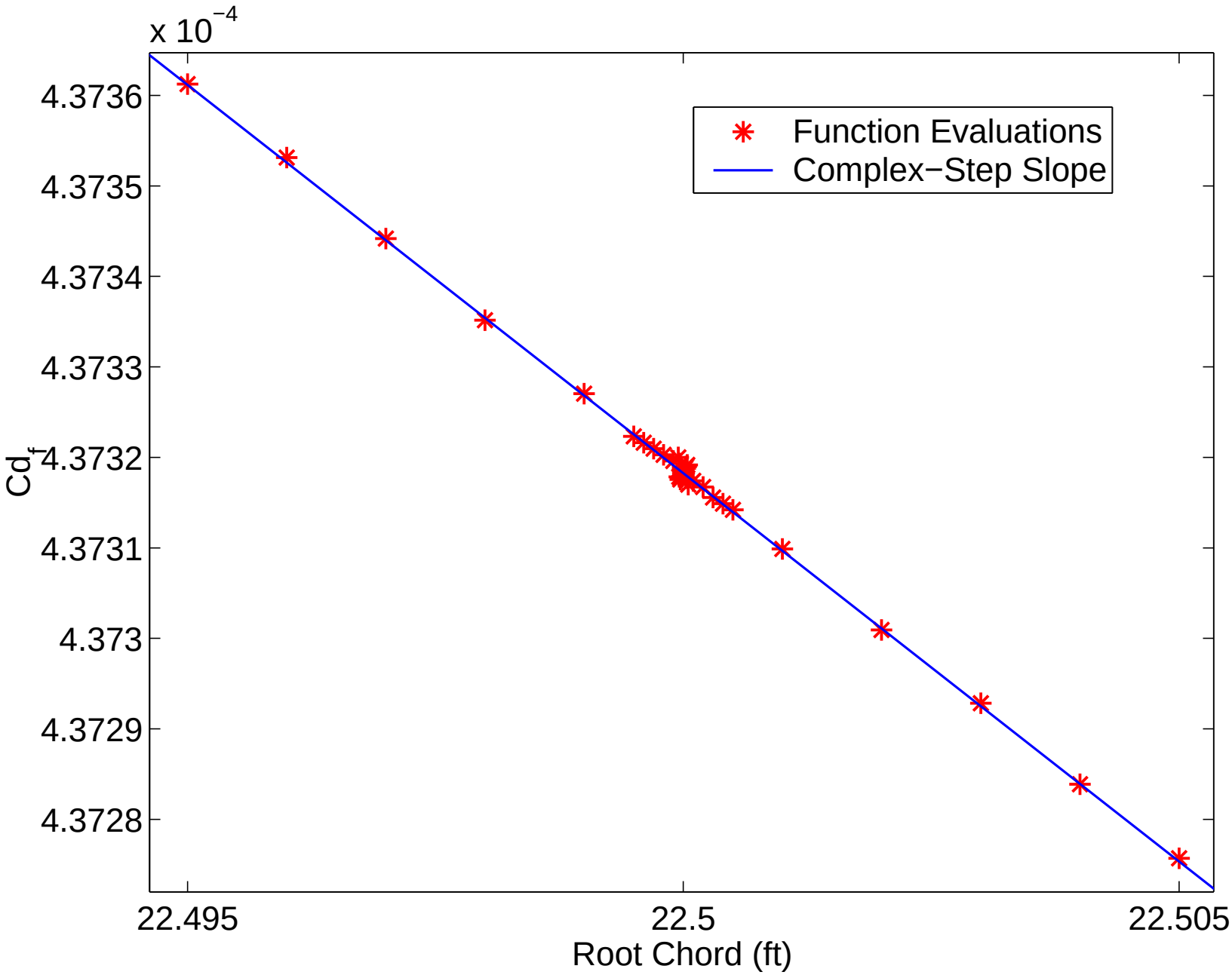


Figure 4.5: Laminar skin friction coefficient (C_D) vs. root chord

4.5 Hyper-dual Numbers

The complex-step method works very well for first derivatives. While it can also be applied to compute second and higher derivatives, subtractive cancellation will be present. In other words, the complex-step method will suffer from the same issues as finite-difference methods when computing f'', f''', etc.

Fortunately, there is a way to compute higher-order derivatives that does not suffer from subtractive cancellation. This approach is related to dual numbers, which are a form of generalized complex number. A dual number is defined by

$$z = x + \varepsilon y,$$

where x, y are real numbers and ε is a pure dual number satisfying $\varepsilon^2 = 0$ when $\varepsilon \neq 0$. The operations of addition and multiplication for dual numbers are defined below.

$$(p + q\varepsilon) + (r + s\varepsilon) = (p + r) + \varepsilon(q + s),$$
$$(p + q\varepsilon)(r + s\varepsilon) = pr + \varepsilon(ps + qr)$$

Question: How can dual numbers be used to compute the first derivative?

Answer: Expand $f(x)$ in a Taylor series about x with a dual-number perturbation.

$$\begin{aligned} f(x + \varepsilon h) &= f(x) + \varepsilon h f'(x) + \varepsilon^2 \frac{h^2}{2!} f''(x) + \varepsilon^3 \frac{h^3}{3!} f'''(x) + \cdots \\ &= f(x) + \varepsilon f'(x) \\ \Rightarrow f'(x) &= \frac{1}{h} \text{part}_\varepsilon \left[f(x + \varepsilon h) \right] \end{aligned}$$

Remarks:

- There is no subtractive cancellation
- There is NO truncation error! (can choose $h = 1$).
- Implementation: define a dual-number class and use operator overloading

Thus, dual numbers offer some attractive properties; however, there is no significant advantage to using dual numbers versus complex numbers when computing first derivatives. In particular, while dual numbers require fewer arithmetic operations than complex numbers, many languages have complex-number implementations that may offset this advantage. Moreover, implementing dual-number arithmetic in some languages (e.g. FORTRAN 77) presents challenges.

The advantage of dual numbers is that they can be extended to hyper-dual numbers, which can be used to compute higher-order derivatives without truncation.

Question: Think of how you might extend dual numbers to permit the computation of the second derivative. Hint: use a perturbation of the form $x + \varepsilon_1 h + \varepsilon_2 h$ where ε_1 and ε_2 are some form of "unusual" number.

Question: Think of how you might extend dual numbers to permit the computation of the second derivative. Hint: use a perturbation of the form $x + \varepsilon_1 h + \varepsilon_2 h$ where ε_1 and ε_2 are some form of "unusual" number.

Answer: Suppose ε_1 and ε_2 are dual numbers that are "independent" in the following sense:

$$\varepsilon_1^2 = 0, \quad \varepsilon_2^2 = 0, \quad \varepsilon_1 \varepsilon_2 \neq 0.$$

Then we can evaluate f at the "perturbed" state $x + \varepsilon_1 h + \varepsilon_2 h$ and obtain

$$\begin{aligned} f(x + \varepsilon_1 h + \varepsilon_2 h) &= f(x) + (\varepsilon_1 + \varepsilon_2) h f'(x) + (\varepsilon_1^2 + 2\varepsilon_1\varepsilon_2 + \varepsilon_2^2) \frac{h^2}{2!} f''(x) + \cdots \\ &= f(x) + (\varepsilon_1 + \varepsilon_2) h f'(x) + \varepsilon_1 \varepsilon_2 h^2 f''(x) \\ \Rightarrow f''(x) &= \frac{1}{h^2} \mathsf{part}_{\varepsilon_1 \varepsilon_2} \left[f(x + \varepsilon_1 h + \varepsilon_2 h) \right] \end{aligned}$$

Remarks:

- As with the first-derivative, there is no subtractive cancellation
- There is again no truncation error.
- Implementation: define a hyper-dual-number class and use operator overloading

The concept of hyper-dual numbers can easily be extended to multivariate problems to compute mixed partial derivatives, and it can also be extended to compute third or higher-order derivatives. See reference [6] for the details.

4.6 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.

4.6.1 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*).

There is nothing like an example, so we will now use both the forward and reverse modes to compute the derivatives of the vector function,

$$\begin{bmatrix} f_1 \\ f_2 \end{bmatrix} = \begin{bmatrix} (x_1 x_2 + \sin x_1)\left(3x_2^2 + 6\right) \\ x_1 x_2 + x_2^2 \end{bmatrix} \tag{4.15}$$

The exact matrix of sensitivities is

$$\frac{\partial f}{\partial x} = \begin{bmatrix} (x_2 + \cos x_1)\left(3x_2^2 + 6\right) & x_1\left(3x_2^2 + 6\right) + 6x_2\left(x_1 x_2 + \sin x_1\right) \\ x_2 & x_1 + 2x_2 \end{bmatrix} \tag{4.16}$$

We want this matrix evaluated at $x = [\pi/4, 2]$, i.e.

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 48.73 & 41.47 \\ 2.00 & 4.79 \end{bmatrix} \tag{4.17}$$

In general, we denote the *independent* variables as $t_1, t_2, \ldots t_n$. In this case $n = 2$. We also need to denote the *dependent* variables, which we write as $t_{n+1}, t_{n+2}, \ldots, t_m$. These are all the intermediate variables in the algorithm, including the outputs we are interested in.

The sequence of operations in the algorithm is then given by

$$t_i = T_i\left(t_1, t_2, \ldots t_{i-1}\right), \quad i = n+1, n+2, \ldots, m. \tag{4.18}$$

Each of these functions are either unary or binary operations.

We can write the example function as the following series of computations:

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

Thus in this case, $m = 12$.

The graph of the algorithm provides information on the interdependence of all the intermediate variables. A graph for our sample algorithm is shown in Figure 4.6.

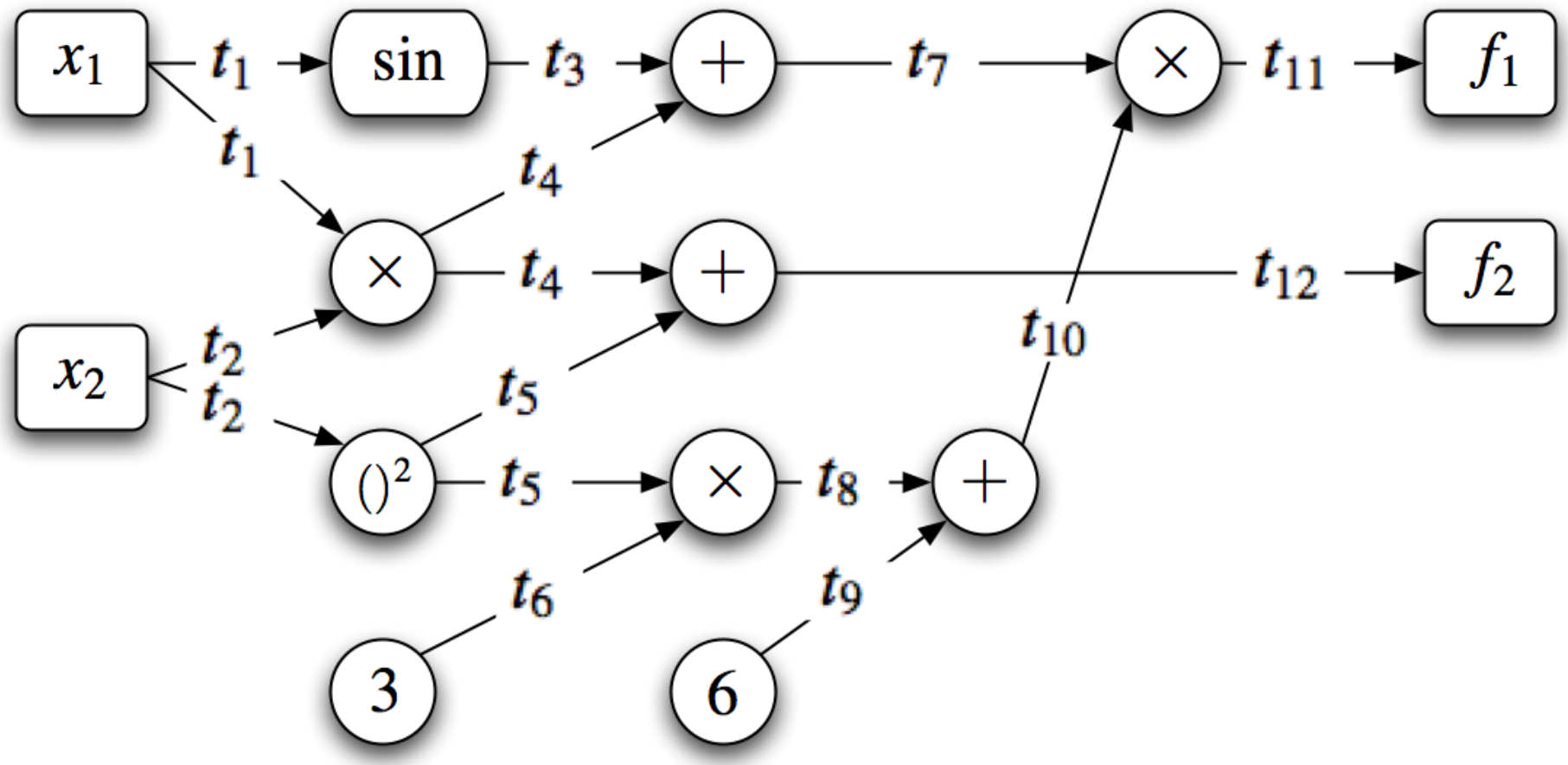


Figure 4.6: Graph showing dependency of independent, intermediate and dependent variables.

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

$$\frac{\partial t_i}{\partial t_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial T_i}{\partial t_k} \frac{\partial t_k}{\partial t_j}, \quad j \leq i \leq m \tag{4.19}$$

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 $i = 2, \ldots, m$ 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 $j = m - 1, \ldots, 1$ all the way to the independent variables.

Now let's use the forward mode to compute $\partial f_1/\partial x_1$ in our example, we set $j = 1$ and then $i = 1, \ldots, 12$:

$$\frac{\partial t_1}{\partial x_1} = 1, \qquad \frac{\partial t_2}{\partial x_1} = 0$$

$$\frac{\partial t_3}{\partial t_1} = \frac{\partial t_3}{\partial t_1}\frac{\partial t_1}{\partial t_1} = \cos t_1$$

$$\frac{\partial t_4}{\partial t_1} = \frac{\partial t_4}{\partial t_1}\frac{\partial t_1}{\partial t_1} + \frac{\partial t_4}{\partial t_2}\frac{\partial t_2}{\partial t_1} = t_2$$

$$\frac{\partial t_5}{\partial t_1} = 0, \qquad \frac{\partial t_6}{\partial t_1} = 0$$

$$\frac{\partial t_7}{\partial t_1} = \frac{\partial t_7}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial t_7}{\partial t_4}\frac{\partial t_4}{\partial t_1} = \cos t_1 + t_2$$

$$\frac{\partial t_8}{\partial t_1} = 0, \qquad \frac{\partial t_9}{\partial t_1} = 0, \qquad \frac{\partial t_{10}}{\partial t_1} = 0$$

$$\frac{\partial f_1}{\partial x_1} = \frac{\partial t_{11}}{\partial t_1} = \frac{\partial t_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_1} + \frac{\partial t_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_1} = t_{10}\frac{\partial t_7}{\partial t_1} = \left(3x_2^2 + 6\right)(\cos x_1 + x_2)$$

$$\frac{\partial f_2}{\partial x_1} = \frac{\partial t_{12}}{\partial t_1} = \frac{\partial t_{12}}{\partial t_4}\frac{\partial t_4}{\partial t_1} + \frac{\partial t_{12}}{\partial t_5}\frac{\partial t_5}{\partial t_1} = x_2$$

Note that although we did not set out to compute $\partial f_2/\partial x_1$, this was obtained with little

additional cost.

Now the reverse mode, we set $i = 11$ and loop $j = 10, \ldots, 1$:

$$\frac{\partial f_1}{\partial t_{11}} = 1$$

$$\frac{\partial f_1}{\partial t_{10}} = \frac{\partial t_{11}}{\partial t_{11}}\frac{\partial t_{11}}{\partial t_{10}} = t_7$$

$$\frac{\partial f_1}{\partial t_9} = \frac{\partial t_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_9} = t_7 \cdot 1 = t_7$$

$$\frac{\partial f_1}{\partial t_8} = \frac{\partial t_{11}}{\partial t_{10}}\frac{\partial t_{10}}{\partial t_8} = t_7 \cdot 1 = t_7$$

$$\frac{\partial f_1}{\partial t_7} = \frac{\partial t_{11}}{\partial t_{11}}\frac{\partial t_{11}}{\partial t_7} = t_{10}$$

$$\frac{\partial f_1}{\partial t_6} = \frac{\partial t_{11}}{\partial t_8}\frac{\partial t_8}{\partial t_6} = t_7 t_5$$

$$\frac{\partial f_1}{\partial t_5} = \frac{\partial t_{11}}{\partial t_8}\frac{\partial t_8}{\partial t_5} = t_7 t_6$$

$$\frac{\partial f_1}{\partial t_4} = \frac{\partial t_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_4} = t_{10} \cdot 1 = t_{10}$$

$$\frac{\partial f_1}{\partial t_3} = \frac{\partial t_{11}}{\partial t_7}\frac{\partial t_7}{\partial t_3} = t_{10} \cdot 1 = t_{10}$$

$$\frac{\partial f_1}{\partial t_2} = \frac{\partial t_{11}}{\partial t_4}\frac{\partial t_4}{\partial t_2} + \frac{\partial t_{11}}{\partial t_5}\frac{\partial t_5}{\partial t_2} = t_{10}t_1 + t_7 t_6 2t_2 = \left(3x_2^2 + 6\right) x_1 + 6x_2 \left(\sin x_1 + x_1 x_2\right)$$

$$\frac{\partial f_1}{\partial t_1} = \frac{\partial t_{11}}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial t_{11}}{\partial t_4}\frac{\partial t_4}{\partial t_1} = t_{10}\cos t_1 + t_{10}t_2 = \left(3x_2^2 + 6\right)\left(\cos x_1 + x_2\right)$$

Although we started this procedure with the objective of computing $\partial f_z / \partial x_1$ only, we observe that with little additional effort we can compute $\partial f_z / \partial x_2$ as well.

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.

Recall chain rule (4.19) that forms the basis for both the forward and reverse modes

$$\frac{\partial t_i}{\partial t_j} = \sum_{k=1}^{i-1} \frac{\partial f_i}{\partial t_k} \frac{\partial t_k}{\partial t_j}, \quad j = 1, 2, \ldots, n \tag{4.20}$$

This can be written as a matrix equation,

$$\underbrace{L}_{m \times m} \underbrace{D}_{m \times n} = \underbrace{E}_{m \times n} \tag{4.21}$$

where L is the $m \times m$ lower-triangular matrix of partial derivatives of each function with respect to the intermediate variables involved in the function, i.e. these are explicit derivatives. The unit diagonal multiplied by D represents the left-hand side of the chain rule, while the nonzero off-diagonal terms come from $\partial f_i / \partial t_k$ in the chain rule and therefore, $L_{ij} = -\partial f_i / \partial t_k$. For each row of L, only one or two off-diagonal terms are nonzero.

D is the matrix of sensitivities we must solve for. Each column of D is the m-vector of derivatives $\partial t_k / \partial x_j$, $k = 1, 2, \ldots, m$. We seek the the bottom $n \times n$ block of this matrix.

Each column of E is the j^{th} unit vector.

In our example, this matrix equation is,

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -\sqrt{2}/2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -2 & -\pi/4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & -4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -18 & 0 & 0 & -2.28 & 1 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2} \\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2} \\ \frac{\partial t_3}{\partial x_1} & \frac{\partial t_3}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial t_9}{\partial x_1} & \frac{\partial t_9}{\partial x_2} \\ \frac{\partial t_{10}}{\partial x_1} & \frac{\partial t_{10}}{\partial x_2} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{4.22}$$

The solution of this system is

$$D = \begin{bmatrix} \frac{\partial t_1}{\partial x_1} & \frac{\partial t_1}{\partial x_2} \\ \frac{\partial t_2}{\partial x_1} & \frac{\partial t_2}{\partial x_2} \\ \frac{\partial t_3}{\partial x_1} & \frac{\partial t_3}{\partial x_2} \\ \vdots & \vdots \\ \frac{\partial t_{10}}{\partial x_1} & \frac{\partial t_{10}}{\partial x_2} \\ \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.71 & 0 \\ 2 & 0.79 \\ 0 & 4 \\ 0 & 0 \\ 2.71 & 0.79 \\ 0 & 12 \\ 0 & 0 \\ 0 & 12 \\ 48.73 & 41.47 \\ 2 & 4.79 \end{bmatrix} \tag{4.23}$$

The solution can be obtained in two different ways.

Forward mode: Pick one column j of D and the corresponding column of E and solve for the derivative of all variables with respect to one independent variable by forward substitution,

$$LD_{.j} = E_{.j} \Rightarrow Ld_j = e_j \tag{4.24}$$

Reverse mode: We solve for the chosen row of D. This can be done by writing

$$L^{-1} = L_m^{-1} L_{m-1}^{-1} \dots L_{n+1}^{-1} \tag{4.25}$$

A row k of D can then be computed from

$$D_{k\cdot} = e_k^T L_m^{-1} L_{m-1}^{-1} \dots L_{n+1}^{-1} E \tag{4.26}$$

4.6.2 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.

Many tools for automatic differentiation of programs in different languages exist. They have been extensively developed and provide the user with great functionality, including the calculation of higher-order derivatives and reverse mode options.

Fortran: Tools that use the source transformation approach include: ADIFOR, TAMC, DAFOR, GRESS, TAPENADE. 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.

4.6.3 The Connection to Algorithmic Differentiation

The new function definitions in these examples can be generalized:

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

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

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

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

Example 4.4: Complex-Step Method Terms

Consider a more involved function, e.g., $f = (xy + \sin x + 4)(3y^2 + 6)$,

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

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

4.6.4 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.

4.7 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.

4.7.1 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, $i = 1, \ldots, N_{\mathcal{R}}$
Ψ_k	adjoint vector, $k = 1, \ldots, N_{\mathcal{R}}$

4.7.2 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

$$f = f(x_n, y_i), \tag{4.27}$$

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

$$\mathcal{R}_k\left(x_n, y_i\left(x_n\right)\right) = 0. \tag{4.28}$$

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.

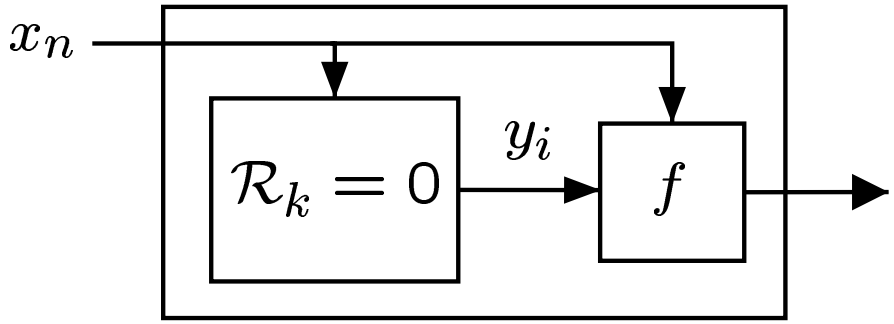


Figure 4.7: Schematic representation of the governing equations ($\mathcal{R}_k = 0$), design variables or inputs (x_j), state variables (y_i) and the function of interest or output (f).

A graphical representation of the system of governing equations is shown in Figure 4.7, 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

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} + \frac{\partial f}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n}, \tag{4.29}$$

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 (4.29) 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 (4.28) 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,

$$\frac{\mathrm{d}\mathcal{R}_k}{\mathrm{d}x_n} = \frac{\partial \mathcal{R}_k}{\partial x_n} + \frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = 0, \tag{4.30}$$

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 (4.30) as

$$\frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = -\frac{\partial \mathcal{R}_k}{\partial x_n}. \tag{4.31}$$

We can solve for $\mathrm{d}y_i/\,\mathrm{d}x_n$ and substitute this result into the total derivative equation (4.29), to obtain

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} - \underbrace{\frac{\partial f}{\partial y_i}\overbrace{\left[\frac{\partial \mathcal{R}_k}{\partial y_i}\right]^{-1}}^{-\mathrm{d}y_i/\,\mathrm{d}x_n}}_{-\Psi_k}\frac{\partial \mathcal{R}_k}{\partial x_n}\,. \tag{4.32}$$

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.

4.7.3 Direct Sensitivity Equations

The approach where we first calculate $\mathrm{d}y_i/\,\mathrm{d}x_n$ using equation (4.31) and then substitute the result in the expression for the total sensitivity (4.32) is called the *direct* method. Note that solving for $\mathrm{d}y_i/\,\mathrm{d}x_n$ requires the solution of the matrix equation (4.31) *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.

4.7.4 Adjoint Sensitivity Equations

Returning to the total sensitivity equation (4.32), we observe that there is an alternative option for computing the total sensitivity $\mathrm{d}f/\,\mathrm{d}x_n$. The auxiliary vector Ψ_k can be obtained by solving the *adjoint equations*

$$\frac{\partial \mathcal{R}_k}{\partial y_i}\Psi_k = -\frac{\partial f}{\partial y_i}. \tag{4.33}$$

The vector Ψ_k is usually called the *adjoint vector* and is substituted into equation (4.32) 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.

4.7.5 Direct vs. Adjoint

We can now see that the choice of the solution procedure (direct vs. adjoint) to obtain the total sensitivity (4.32) 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, Ψ_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.

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 4.7.5. 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 (4.31) and (4.33) 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 (4.31) and (4.33) respectively. The cost involved in this step when computing the same set of sensitivities is the same for both methods.

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

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 4.8 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.

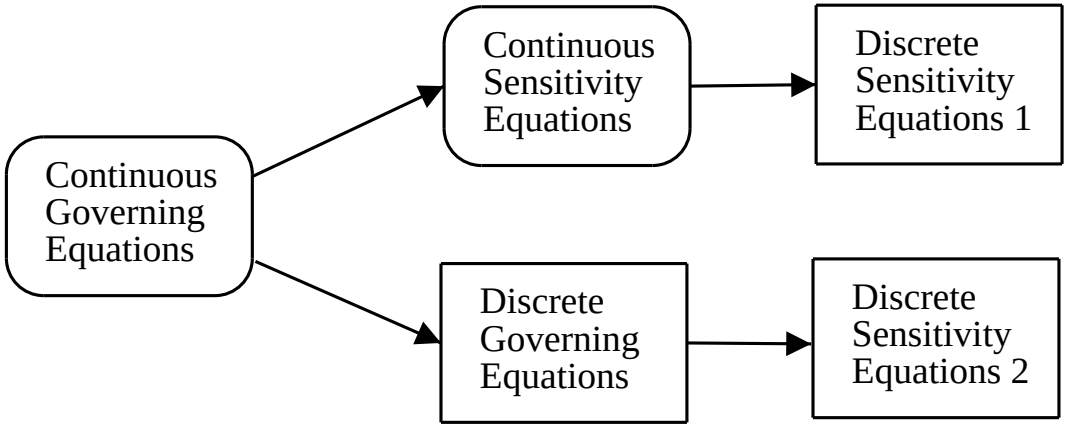


Figure 4.8: The two ways of obtaining the discretized sensitivity equations

The resulting sensitivity equations should be equivalent, but are not necessarily the same. 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.

4.7.6 Example: Structural Sensitivity Analysis

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

$$\mathcal{R}_k = K_{ki} u_i - F_k = 0, \tag{4.34}$$

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,

$$\sigma_m = S_{mi} u_i. \tag{4.35}$$

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 (4.32).

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.,

$$\frac{\partial \mathcal{R}_k}{\partial y_i} = \frac{\partial (K_{ki} u_i - F_k)}{\partial u_i} = K_{ki}. \tag{4.36}$$

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,

$$\frac{\partial \mathcal{R}_k}{\partial x_j} = \frac{\partial (K_{ki} u_i - F_k)}{\partial A_j} = \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.37}$$

The partial derivative of the stress with respect to the displacements is simply given by the matrix in equation (4.35), i.e.,

$$\frac{\partial f_m}{\partial y_i} = \frac{\partial \sigma_m}{\partial u_i} = S_{mi} \tag{4.38}$$

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

$$\frac{\partial f_m}{\partial x_j} = \frac{\partial \sigma_m}{\partial A_j} = 0. \tag{4.39}$$

Substituting these into the generalized total sensitivity equation (4.32) we get:

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = -\frac{\partial \sigma_m}{\partial u_i} K^{-1}_{ki} \frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.40}$$

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

$$K_{ki} \frac{\mathrm{d}u_i}{\mathrm{d}A_j} = -\frac{\partial K_{ki}}{\partial A_j} u_i \tag{4.41}$$

and then substitute the result in,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = \frac{\partial \sigma_m}{\partial u_i}\frac{\mathrm{d}u_i}{\mathrm{d}A_j} \tag{4.42}$$

to calculate the desired sensitivities.

The adjoint method could also be used, in which case we would solve equation (4.33) for the structures case,

$$K_{ki}^T \psi_k = \frac{\partial \sigma_m}{\partial u_i}. \tag{4.43}$$

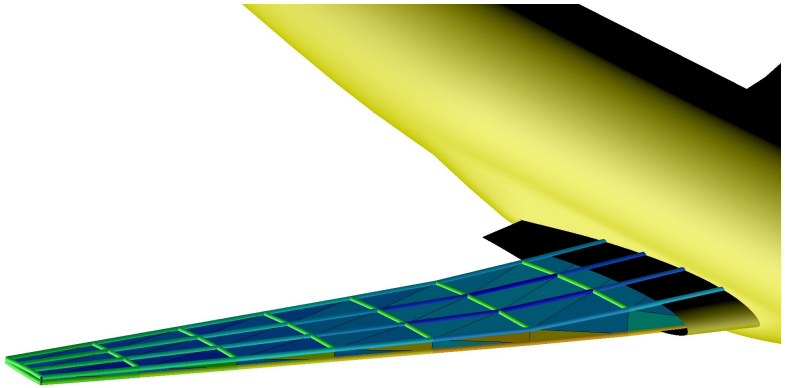
Then we would substitute the adjoint vector into the equation,

$$\frac{\mathrm{d}\sigma_m}{\mathrm{d}A_j} = \frac{\partial \sigma_m}{\partial A_j} + \psi_k^T \left(-\frac{\partial K_{ki}}{\partial A_j} u_i \right). \tag{4.44}$$

to calculate the desired sensitivities.

Example 4.5: 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.

References

[1] Tools for automatic differentiation.

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

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

[4] C. Bendtsen and O. 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.

[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.

[6] J. A. Fike and J. J. Alonso. The development of hyper-dual numbers for exact second-derivative calculations. In *49th AIAA Aerospace Sciences Meeting*, number AIAA–2011–886, Orlando, Florida, Jan. 2011.

[7] A. Griewank. *Evaluating Derivatives*. SIAM, Philadelphia, 2000.

[8] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical report 300, INRIA, 2004.

[9] 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.

[10] J. N. Lyness and C. B. Moler. Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis*, 4(2):202–210, 1967.

[11] J. R. R. A. Martins. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization*. PhD thesis, Stanford University, Stanford, CA, 2002.

[12] J. R. R. A. Martins. A guide to the complex-step derivative approximation, 2003.

[13] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004.

[14] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. A coupled-adjoint sensitivity analysis method for high-fidelity aero-structural design. *Optimization and Engineering*, 6(1):33–62, March 2005.

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

[16] S. Nadarajah and A. 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.

[17] 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.

[18] J. D. Pryce and J. 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.

[19] W. Squire and G. Trapp. Using complex variables to estimate derivatives of real functions. *SIAM Review*, 40(1):110–112, 1998.

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