The Complex-Step Derivative Approximation

JOAQUIM R. R. A. MARTINS
University of Toronto Institute for Aerospace Studies
and
PETER STURDZA and JUAN J. ALONSO
Stanford University

The complex-step derivative approximation and its application to numerical algorithms are presented. Improvements to the basic method are suggested that further increase its accuracy and robustness and unveil the connection to algorithmic differentiation theory. A general procedure for the implementation of the complex-step method is described in detail and a script is developed that automates its implementation. Automatic implementations of the complex-step method for Fortran and C/C++ are presented and compared to existing algorithmic differentiation tools. The complex-step method is tested in two large multidisciplinary solvers and the resulting sensitivities are compared to results given by finite differences. The resulting sensitivities are shown to be as accurate as the analyses. Accuracy, robustness, ease of implementation and maintainability make these complex-step derivative approximation tools very attractive options for sensitivity analysis.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms—*analysis of algorithms*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Automatic differentiation, forward mode, complex-step derivative approximation, overloading, gradients, sensitivities

1. INTRODUCTION

Sensitivity analysis has been an important area of engineering research, especially in design optimization. In choosing a method for computing sensitivities, one is mainly concerned with its accuracy and computational expense. In certain cases it is also important that the method be easily implemented.

One method that is very commonly used is finite differencing. Although it is not known for being particularly accurate or computationally efficient, the biggest advantage of this method resides in the fact that it is extremely easy to implement.

Authors' addresses: J. R. R. A. Martins, Institute for Aerospace Studies, University of Toronto, 4925 Dufferin Street, Toronto, Ont. M3H 5T6, Canada; email: martins@utias.utoronto.ca; P. Sturdza and J. J. Alonso, Department of Aeronautics and Aeronautics, Stanford University, Stanford, CA.

Analytic and semi-analytic methods for sensitivity analysis are much more accurate and can be classified in terms of both derivation and implementation. The continuous approach to deriving the sensitivity equations for a given system consists in differentiating the governing equations first and then discretizing the resulting sensitivity equations prior to solving them numerically. The alternative to this—the discrete approach—is to discretize the governing equations first and then differentiate them to obtain sensitivity equations that can be solved numerically. Furthermore, for both the discrete and continuous approaches, there are two ways of obtaining sensitivities: the direct method or the adjoint method.

In terms of implementation, the continuous approach can only be derived by hand, while the discrete approach to differentiation can be implemented automatically if the program that solves the discretized governing equations is provided. This method is known as algorithmic differentiation, computational differentiation or automatic differentiation. It is a well-known method based on the systematic application of the chain rule of differentiation to computer programs [Griewank 2000; Corliss et al. 2001]. This approach is as accurate as other analytic methods, and it is considerably easier to implement.

The use of complex variables to develop estimates of derivatives originated with the work of Lyness and Moler [1967] and Lyness [1967]. Their papers introduced several methods that made use of complex variables, including a reliable method for calculating the nth derivative of an analytic function. This theory was used by Squire and Trapp [1998] to obtain a very simple expression for estimating the first derivative. This estimate is suitable for use in modern numerical computing and has been shown to be very accurate, extremely robust and surprisingly easy to implement, while retaining a reasonable computational cost. The potential of this technique is now starting to be recognized and it has been used for sensitivity analysis in computational fluid dynamics (CFD) by Anderson et al. [1999] and in a multidisciplinary environment by Newman et al. [1998]. Further research on the subject has been carried out by the authors [Martins et al. 2000, 2001; Martins 2002].

The objective of this article is to shed new light on the theory behind the complex-step derivative approximation and to show its relation to algorithmic differentiation, further contributing to the understanding of this relatively new method. On the implementation side, we focus on developing *automatic* implementations, discussing the trade-offs between the complex-step method and algorithmic differentiation when programming in Fortran and C/C++. Finally, computational results corresponding to the application of these tools to large-scale algorithms are presented and compared with finite-difference estimates.

2. THEORY

2.1 First Derivative Approximations

Finite-differencing formulas are a common method for estimating the value of derivatives. These formulas can be derived by truncating a Taylor series expanded about a point x. A common estimate for the first derivative is the forward-difference formula

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h), \tag{1}$$

where h is the finite-difference interval. The truncation error is $\mathcal{O}(h)$, and therefore this represents a first-order approximation. Higher-order finite-difference approximations can also be derived by using combinations of alternate Taylor series expansions.

When estimating sensitivities using finite-difference formulas 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 [Gill et al. 1981].

We now show that an equally simple first derivative estimate for real functions can be obtained using complex calculus. Consider a function, $f = u + iv$, of the complex variable, $z = x + iy$. If f is analytic—that is, if it is differentiable in the complex plane—the Cauchy–Riemann equations apply and

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y}, \tag{2}$$

$$\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}. \tag{3}$$

These equations establish the exact relationship between the real and imaginary parts of the function. We can use the definition of derivative in the right-hand side of the first Cauchy–Riemann equation (2) to write

$$\frac{\partial u}{\partial x} = \lim_{h \to 0} \frac{v(x + i(y+h)) - v(x+iy)}{h}, \tag{4}$$

where h is a real number. Since the functions that we are interested in are originally real functions of real variables, $y = 0$, $u(x) = f(x)$ and $v(x) = 0$. Equation (4) can then be rewritten as

$$\frac{\partial f}{\partial x} = \lim_{h \to 0} \frac{\operatorname{Im}\left[f\,(x+ih)\right]}{h}. \tag{5}$$

For a small discrete h, this can be approximated by

$$\frac{\partial f}{\partial x} \approx \frac{\operatorname{Im}\left[f\,(x+ih)\right]}{h}. \tag{6}$$

We call this the *complex-step derivative approximation*. This estimate is not subject to subtractive cancellation errors, since it does not involve a difference operation. This constitutes a tremendous advantage over the finite-difference approximations.

In order to determine the error involved in this approximation, we repeat the derivation by Squire and Trapp [1998] which is based on a Taylor series expansion. Rather than using a real step h, to derive the complex-step derivative approximation, we use a pure imaginary step, ih. If f is a real function of a real variable and it is also analytic, we can expand it as a Taylor series about a real point x as follows:

$$f(x+ih) = f(x) + ihf'(x) - h^2\frac{f''(x)}{2!} - ih^3\frac{f'''(x)}{3!} + \cdots. \tag{7}$$

Taking the imaginary parts of both sides of this Taylor series expansion (7) and dividing it by h yields

$$f'(x) = \frac{\mathrm{Im}[f(x+ih)]}{h} + h^2\frac{f'''(x)}{3!} + \cdots. \tag{8}$$

Hence, the approximation is an $\mathcal{O}(h^2)$ estimate of the derivative of f. Notice that if we take the real part of the Taylor series expansion (7), we obtain the value of the function on the real axis, that is,

$$f(x) = Re[f(x+ih)] + h^2\frac{f''(x)}{2!} - \cdots, \tag{9}$$

showing that the real part of the result give the value of $f(x)$ correct to $\mathcal{O}(h^2)$.

The second-order errors in the function value (9) and the function derivative (8) can be eliminated when using finite-precision arithmetic by ensuring that h is sufficiently small. If ε is the relative working precision of a given algorithm, we need an h such that

$$h^2\left|\frac{f''(x)}{2!}\right| < \varepsilon|f(x)|, \tag{10}$$

to eliminate the truncation error of $f(x)$ in the expansion (9). Similarly, for the truncation error of the derivative estimate to vanish, we require that

$$h^2\left|\frac{f'''(x)}{3!}\right| < \varepsilon|f'(x)|. \tag{11}$$

Although the step h can be set to extremely small values—as shown in Section 2.2—it is not always possible to satisfy these conditions (10, 11), especially when $f(x)$, $f'(x)$ tend to zero.

2.2 A Simple Numerical Example

Since the complex-step approximation does not involve a difference operation, we can choose extremely small step sizes with no loss of accuracy.

To illustrate this point, consider the following analytic function:

$$f(x) = \frac{\mathrm{e}^x}{\sqrt{\sin^3 x + \cos^3 x}}. \tag{12}$$

The exact derivative at $x = 1.5$ is computed analytically to 16 digits and then compared to the results given by the complex-step formula (6) and the forward and central finite-difference approximations.

Figure 1 shows that 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

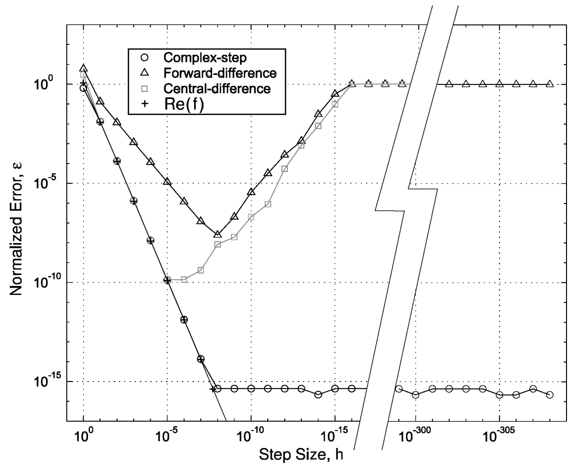

Fig. 1. Relative error in the sensitivity estimates given by the finite-difference and the complex-step methods using the analytic result as the reference; $\varepsilon = |f' - f'_{ref}|/|f'_{ref}|$.

the central difference, subtractive cancellation errors become significant and the resulting 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, for any step size below 10^{-8}, it achieves the accuracy of the function evaluation. Comparing the optimum accuracy of each of these approaches, we can see that, by using finite differences, we only achieve a fraction of the accuracy that is obtained by using the complex-step approximation.

Note that the real part of the perturbed function also converges quadratically to the actual value of the function, as predicted by the Taylor series expansion (9).

Although the size of the complex step can be made extremely small, there is a lower limit when using finite-precision arithmetic. The range of real numbers that can be handled in numerical computations is dependent on the particular compiler that is used. In this case, double precision arithmetic is used and the smallest nonzero number that can be represented is 10^{-308}. If a number falls below this value, underflow occurs and the representation of that number typically results in a zero value.

2.3 Function and Operator Definitions

In the derivation of the complex-step derivative approximation for a function f (6), we assume that f is analytic, that is, that the Cauchy–Riemann equations (2, 3) apply. It is therefore important to determine to what extent this assumption holds when the value of the function is calculated by a numerical algorithm. In addition, it is useful to explain how real functions and operators can be defined such that the complex-step derivative approximation yields the correct result when used in a computer program.

Relational logic operators such as "greater than" and "less than" are usually not defined for complex numbers. These operators are often used in programs, in conjunction with conditional statements to redirect the execution thread. The original algorithm and its "complexified" version must obviously follow the same execution thread. Therefore, defining these operators to compare only the real parts of the arguments is the correct approach. Functions that choose one argument such as the maximum or the minimum values are based on relational operators. Therefore, following the previous argument, we should also choose a number based on its real part alone.

Any algorithm that uses conditional statements is likely to be a discontinuous function of its inputs. Either the function value itself is discontinuous or the discontinuity is in the first or higher derivatives. When using a finite-difference method, the derivative estimate is incorrect if the two function evaluations are within h of the discontinuity location. However, when using the complex-step method, the resulting derivative estimate is correct right up to the discontinuity. At the discontinuity, a derivative does not exist by definition, but if the function is continuous up to that point, the approximation still returns a value corresponding to the one-sided derivative. The same is true for points where a given function has singularities.

Arithmetic functions and operators include addition, multiplication, and trigonometric functions, to name only a few. Most of these have a standard complex definition that is analytic, in which case the complex-step derivative approximation yields the correct result.

The only standard complex function definition that is non-analytic is the absolute value function. When the argument of this function is a complex number, the function returns the positive real number, $|z| = \sqrt{x^2 + y^2}$. The definition of this function is not derived by imposing analyticity and therefore it does not yield the correct sensitivity when using the complex-step estimate. In order to derive an analytic definition of the absolute value function, we must ensure that the Cauchy–Riemann equations (2, 3) are satisfied. Since we know the exact derivative of the absolute value function, we can write

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} = \begin{cases} -1, & \text{if} \quad x < 0, \\ +1, & \text{if} \quad x > 0. \end{cases} \tag{13}$$

From equation (3), since $\partial v/\partial x = 0$ on the real axis, we get that $\partial u/\partial y = 0$ on the same axis, and therefore the real part of the result must be independent of the imaginary part of the variable. Therefore, the new sign of the imaginary part depends only on the sign of the real part of the complex number, and an analytic "absolute value" function can be defined as

$$\mathtt{abs}(x + iy) = \begin{cases} -x - iy, & \text{if} \quad x < 0, \\ +x + iy, & \text{if} \quad x > 0. \end{cases} \tag{14}$$

Note that this is not analytic at $x = 0$ since a derivative does not exist for the real absolute value. In practice, the $x > 0$ condition is substituted by $x \geq 0$ so

Table I. The Differentiation of the Multiplication Operation $f = x_1x_2$ with Respect to x_1 using Algorithmic Differentiation in Forward Mode and the Complex-Step Derivative Approximation

Forward AD	Complex-Step Method
$\Delta x_1 = 1$	$h_1 = 10^{-20}$
$\Delta x_2 = 0$	$h_2 = 0$
$f = x_1x_2$	$f = (x_1 + ih_1)(x_2 + ih_2)$
$\Delta f = x_1\Delta x_2 + x_2\Delta x_1$	$f = x_1x_2 - h_1h_2 + i(x_1h_2 + x_2h_1)$
$\partial f/\partial x_1 = \Delta f$	$\partial f/\partial x_1 = \operatorname{Im} f/h_1$

that we can obtain a function value for $x = 0$ and calculate the correct right-hand-side derivative at that point.

2.4 The Connection to Algorithmic Differentiation

When using the complex-step derivative approximation, in order to effectively eliminate truncation errors, it is typical to use a step that is many orders of magnitude smaller than the real part of the calculation. When the truncation errors are eliminated, the higher-order terms of the derivative approximation (8) are so small that they vanish when added to other terms using finite-precision arithmetic.

We now observe that by linearizing the Taylor series expansion (7) of a complex function about x we obtain

$$f(x + ih) \equiv f(x) + ih\frac{\partial f(x)}{\partial x}, \tag{15}$$

where the imaginary part is exactly the derivative of f times h. The end result is a sensitivity calculation method that is equivalent to the forward mode of algorithmic differentiation, as observed by Griewank [2000, chap. 10, p. 227].

Algorithmic differentiation (AD) is a well-established method for estimating derivatives [Griewank 2000; Bischof et al. 1992]. The method is based on the application of the chain rule of differentiation to each operation in the program flow. For each intermediate variable in the algorithm, a variation due to one input variable is carried through. As a simple example, suppose we want to differentiate the multiplication operation, $f = x_1x_2$, with respect to x_1. Table I compares how the differentiation would be performed using either algorithmic differentiation in the forward mode or the complex-step method.

As we can see, algorithmic differentiation stores the derivative value in a separate set of variables while the complex-step method carries the derivative information in the imaginary part of the variables. In the case of this operation, we observe that the complex-step procedure performs one additional operation—the calculation of the term h_1h_2—which for the purposes of calculating the derivative is superfluous (and equal to zero in this case). The complex-step method nearly always includes these unnecessary computations and is therefore generally less efficient than algorithmic differentiation. The additional computations correspond to the higher-order terms in equation (8).

Although this example involves only one operation, both methods work for an algorithm with an arbitrary sequence of operations by propagating the variation of one input throughout the code. This means that the cost of calculating a given set of sensitivities is proportional to the number of inputs. This particular form of algorithmic differentiation is called the *forward mode*. The alternative—the *reverse mode*—has no equivalent in the complex-step method, but is analogous to an adjoint method.

Since the use of the complex-step method has only recently become widespread, there are some issues that seem unresolved. However, now that the connection to algorithmic differentiation has been established, we can look at the extensive research on the subject of algorithmic differentiation for answers. Important issues include how to treat singularities, differentiability problems due to `if` statements, and the convergence of iterative solvers, all of which have been addressed by the algorithmic differentiation research community.

The singularity issue—that is, what to do when the derivative is infinite—is handled automatically by the complex-step method at the expense of some accuracy. For example, the computation of $\sqrt{x+ih}$ differs substantially from $\sqrt{x}+ih/2\sqrt{x}$ as x vanishes, but this has not produced noticeable errors in the algorithms that we tested. Algorithmic differentiation tools usually deal with this and other exceptions in a more rigorous manner, as described by Bischof et al. [1991].

Regarding the issue of `if` statements, in rare circumstances, differentiability may be compromised by piecewise function definitions. Algorithmic differentiation is also subject to this possibility, which cannot be avoided in an automated manner. However, this problem does not occur if all piecewise functions are defined as detailed by Beck and Fischer [1994].

When using the complex-step method for iterative solvers, the experience of the authors has been that the imaginary part converges at a rate similar to the real part, although somewhat lagged. Whenever the iterative process converged to a steady state solution, the derivative also converged to the correct result. Proof of convergence of the derivative given by algorithmic differentiation of certain classes of iterative methods has been obtained by Beck [1994] and Griewank et al. [1993].

3. IMPLEMENTATION

In this section, existing algorithmic differentiation implementations are first described and then the automatic implementation of the complex-step derivative approximation is presented in detail for Fortran and C/C++. Some notes for other programming languages are also included.

3.1 Algorithmic Differentiation

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

In the implementation of algorithmic differentiation by source transformation, the 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 that makes it difficult to read. In the authors' opinion, this constitutes an implementation disadvantage as it becomes impractical to debug this new extended source code.

In order to use derived types, we need languages that support this feature, such as Fortran 90 or C++. Using this feature, algorithmic differentiation can be implemented by creating a new structure that contains both the value of the variable and its derivative. All of the existing operators are then redefined (overloaded) for the new type. The new operator exhibits 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 program.

3.1.1 *Fortran.*

Many tools for automatic algorithmic differentiation of Fortran programs exist. These tools have been extensively developed and some of them provide the user with great functionality by including the option for using the reverse mode, for calculating higher-order derivatives, or for both. Tools that use the source transformation approach include: ADIFOR [Bischof et al. 1992], DAFOR [Berz 1987], GRESS [Horwedel 1991], Odyssée [Faure and Papegay 1997], PADRE2 [Kubota 1996], and TAMC [Giering 1997]. The necessary changes to the source code are made automatically.

The derived datatype approach is used in the following tools: AD01 [Pryce and Reid 1998], ADOL-F [Shiriaev 1996], IMAS [Rhodin 1997] and OPTIMA90 [Brown 1995; Bartholomew-Biggs 1995]. Although it is in theory possible to develop a script to make the necessary changes in the source code automatically, none of these tools have this ability and the changes must be performed manually.

3.1.2 *C/C++.*

Well established tools for automatic algorithmic differentiation also exist for C/C++. These include include ADIC [Bischof et al. 1997], an implementation mirroring ADIFOR, ADOL-C [Griewank et al. 1996], and FADBAD [Bendtsen and Stauning 1996]. Some of these packages use operator overloading and can operate in the forward or reverse mode and compute higher-order derivatives.

3.2 Complex-Step Derivative Approximation

The general procedure for the implementation of the complex-step method for an arbitrary computer program can be summarized as follows:

(1) Substitute all `real` type variable declarations with `complex` declarations. It is not strictly necessary to declare *all* variables complex, but it is much easier to do so.
(2) Define all functions and operators that are not defined for complex arguments.
(3) Add a small complex step (e.g., $h = 1 \times 10^{-20}$) to the desired x, run the algorithm that evaluates f, and then compute $\partial f / \partial x$ using equation (6).

The above procedure is independent of the programming language. We now describe the details of our Fortran and C/C++ implementations.

3.2.1 *Fortran.* In Fortran 90, intrinsic functions and operators (including comparison operators) can be overloaded. This means that if a particular function or operator does not accept complex arguments, one can extend it by writing another definition that does. This feature makes it much easier to implement the complex-step method since once we overload the functions and operators, there is no need to change the function calls or conditional statements.

The complex function and operators needed for implementing the complex-step method are defined in the `complexify` Fortran 90 module. The module can be used by any subroutine in a program, including Fortran 77 subroutines and it redefines all intrinsic complex functions using definition (15), the equivalent of algorithmic differentiation. Operators—like for example addition and multiplication—that are intrinsically defined for complex arguments cannot be redefined, according to the Fortran 90 standard. However, the intrinsic complex definitions are suitable for our purposes.

In order to automate the implementation, we developed a script that processes Fortran source files automatically. The script inserts a statement that ensures that the complex functions module is used in every subroutine, substitutes all the real type declarations by complex ones and adds `implicit complex` statements when appropriate. The script is written in Python [Lutz 1996] and supports a wide range of platforms and compilers. It is also compatible with the message-passing interface (MPI) standard for parallel computing. For MPI-based code, the script converts all real types in MPI calls to complex ones. In addition, the script can also take care of file input and output by changing `format` statements. Alternatively, a separate script can be used to automatically change the input files themselves. The latest versions of both the script and the Fortran 90 module are available from a dedicated web page [Martins 2003].

This tool for implementing the complex-step method represents, in our opinion, a good compromise between ease of implementation and algorithmic efficiency. While pure algorithmic differentiation is numerically more efficient, the complex-step method requires far fewer changes to the original source code, due to the fact that complex variables are a Fortran intrinsic type. The end result is improved maintainability. Furthermore, practically all the changes are performed automatically by the use of the script.

3.2.2 *C/C++.* The C/C++ implementations of the complex-step method and algorithmic differentiation are much more straightforward than in Fortran. Two different C/C++ implementations are presented and used in this article and are available on the World Wide Web [Martins 2003].

The first procedure is analogous to the Fortran implementation, that is, it uses complex variables and overloaded complex functions and operators. An include file, `complexify.h`, defines a new variable type called `cmplx` and all the functions that are necessary for the complex-step method. The inclusion of this file and the replacement of `double` or `float` declarations with `cmplx` is nearly all that is required.

The remaining work involves dealing with input and output routines. The usual casting of the inputs to `cmplx` and printing of the outputs using the `real()` and `imag()` functions works well. For ASCII files or terminal input and output, the use of the C++ iostream library is also possible. In this case, the "$\gg$" operator automatically reads a real number and properly casts it to `cmplx`, or reads in a complex number in the Fortran parenthesis format, for example, "`(2.3,1.e-20)`". The "$\ll$" operator outputs in the same format.

The second method is a version of the forward mode of algorithmic differentiation, that is, the application of definition (15). The method can be implemented by including a file called `derivify.h` and by replacing declarations of type `double` with declarations of type `surreal`. The `derivify.h` file redefines all relational operators, the basic arithmetic formulas, trigonometric functions, and other formulas in the math library when applied to the `surreal` variables. These variables contain "value" and "derivative" parts analogous to the real and imaginary parts in the complex-step method. This works just as the complex-step version, except that the step size may be set to unity since there is no truncation error.

One feature available to the C++ programmer is worth mentioning: templates. Templates make it possible to write source code that is independent of variable type declarations. This approach involves considerable work with complicated syntax in function declarations and requires some object-oriented programming. There is no need, however, to modify the function bodies themselves or to change the flow of execution, even for pure C programs. The distinct advantage is that variable types can be decided at run time, so the very same executable can run either the real-valued, the complex or the algorithmic differentiation version. This simplifies version control and debugging considerably since the source code is the same for all three versions of the program.

3.2.3 *Other Programming Languages.* In addition to the Fortran and C/C++ implementations described above, there was some experimentation with other programming languages.

Matlab. As in the case of Fortran, one must redefine functions such as `abs`, `max` and `min`. All differentiable functions are defined for complex variables. The results shown in Figure 1 are actually computed using Matlab. The standard transpose operation represented by an apostrophe (`'`) poses a problem as it takes the complex conjugate of the elements of the matrix, so one should use the non-conjugate transpose represented by "dot apostrophe" (`.'`) instead.

Java. Complex arithmetic is not standardized at the moment but there are plans for its implementation. Although function overloading is possible, operator overloading is currently not supported.

Python. A simple implementation of the complex-step method for Python was also developed in this work. The `cmath` module must be imported to gain access to complex arithmetic. Since Python supports operator overloading, it is possible to define complex functions and operators as described earlier.

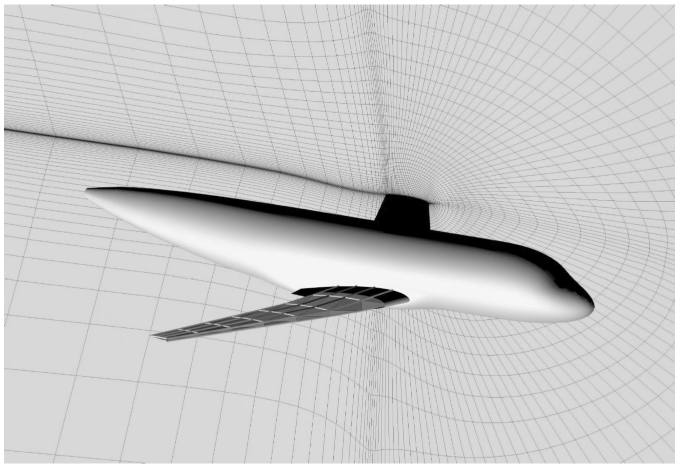
Fig. 2. Aero-structural model and solution of a transonic jet configuration, showing a slice of the grid and the internal structure of the wing.

Algorithmic differentiation by overloading can be implemented in any programming language that supports derived datatypes and operator overloading. For languages that do not have these features, the complex-step method can be used wherever complex arithmetic is supported.

4. RESULTS

4.1 Three-Dimensional Aero-Structural Solver

The tool that we have developed to implement the complex-step method automatically in Fortran has been tested on a variety of programs. One of the most complicated examples is a high-fidelity aero-structural solver, which is part of an MDO framework created to solve wing aero-structural design optimization problems [Reuther et al. 1999b; Martins et al. 2002; Martins 2002]. The framework consists of an aerodynamic analysis and design module (which includes a geometry engine and a mesh perturbation algorithm), a linear finite-element structural solver, an aero-structural coupling procedure, and various pre-processing tools that are used to setup aero-structural design problems. The multidisciplinary nature of this solver is illustrated in Figure 2 where we can see the aircraft geometry, the flow mesh and solution, and the primary structure inside the wing.

The aerodynamic analysis and design module, SYN107-MB [Reuther et al. 1999b], is a multiblock parallel flow solver for both the Euler and the Reynolds averaged Navier–Stokes equations that has been shown to be accurate and efficient for the computation of the flow around full aircraft configurations [Reuther et al. 1997; Yao et al. 2001]. An aerodynamic adjoint solver is also included in this package, enabling aerodynamic shape optimization in the absence of aero-structural interaction.

To compute the flow for this configuration, we use the CFD mesh shown in Figure 2. This is a multiblock Euler mesh with 60 blocks and a total of 229,500

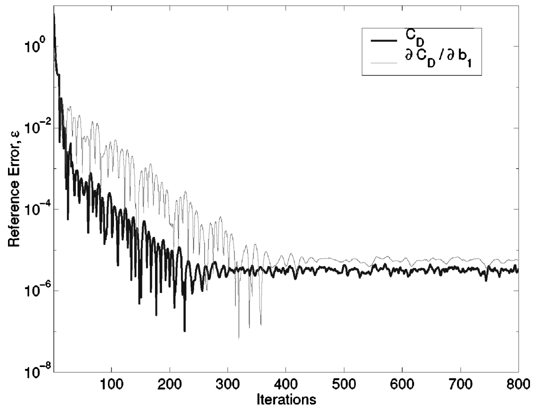

Fig. 3. Convergence of C_D and $\partial C_D/\partial b_1$ for the 3D aero-structural solver; $\varepsilon = (|f - f_{ref}|)/|f_{ref}|$; reference values from the 801st iteration.

mesh points. The structural model of the wing consists of 6 spars, 10 ribs and skins covering the upper and lower surfaces of the wing box. A total of 640 finite elements are used in the construction of this model.

Since an analytic method for calculating sensitivities has been developed for this analysis code, the complex-step method is an extremely useful reference for validation purposes [Martins et al. 2002; Martins 2002]. To validate the complex-step results for the aero-structural solver, we chose the derivative of the drag coefficient, C_D, with respect to a set of 18 wing shape design variables. These variables are shape perturbations in the form of Hicks–Henne *bump* functions [Reuther et al. 1999a], which not only control the aerodynamic shape, but also change the shape of the underlying structure.

Since the aero-structural solver is an iterative algorithm, it is useful to compare the convergence of a given function with that of its derivative, which is contained in its complex part. This comparison is shown in Figure 3 for the drag coefficient and its derivative with the respect to the first shape design variable, b_1. The drag coefficient converges to the precision of the algorithm in about 300 iterations. The drag sensitivity converges at the same rate as the coefficient and it lags slightly, taking about 100 additional iterations to achieve the maximum precision. This is expected, since the calculation of the sensitivity of a given quantity is dependent on the value of that quantity [Beck 1994; Griewank et al. 1993].

The minimum error in the derivative is observed to be slightly lower than the precision of the coefficient. When looking at the number of digits that are converged, the drag coefficient consistently converges to six digits, while the derivative converges to five or six digits. This small discrepancy in accuracy can be explained by the increased round-off errors of certain complex arithmetic operations such as division and multiplication due to the larger number of operations that is involved [Olver 1983]. When performing multiplication, for example, the complex part is the result of two multiplications and one addition, as shown in Table I. Note that this increased error does not affect the real part when such small step sizes are used.

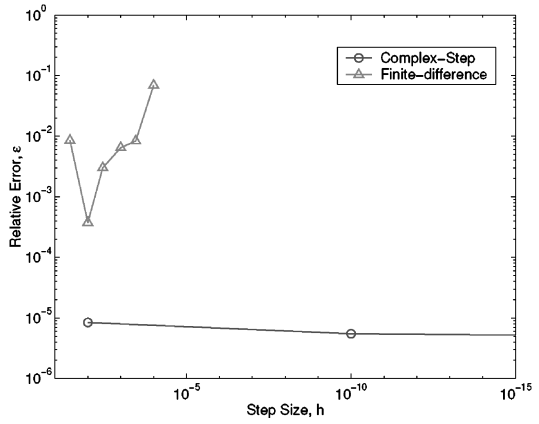

Fig. 4. Sensitivity estimate errors for $\partial C_D/\partial b_1$ given by finite-difference and the complex step for different step sizes; $\varepsilon = (|f - f_{ref}|)/|f_{ref}|$; reference is complex-step estimate at $h = 10^{-20}$.

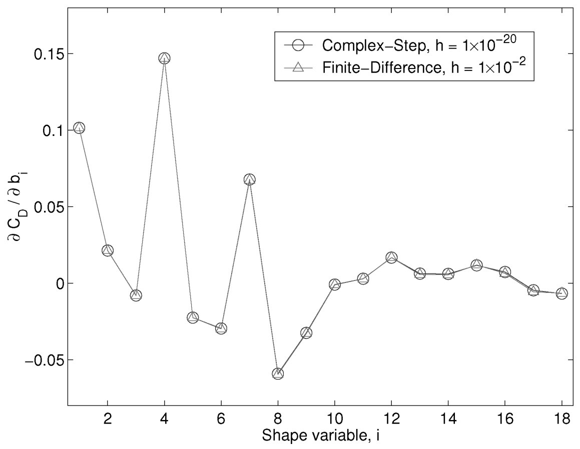

Fig. 5. Comparison of the estimates for the shape sensitivities of the drag coefficient, $\partial C_D/\partial b_i$.

The plot shown in Figure 4 is analogous to that of Figure 1, where the sensitivity estimates given by the complex-step and forward finite-difference methods are compared for a varying step sizes. In this case, the finite-difference result has an acceptable precision only for one step size ($h = 10^{-2}$). Again, the complex-step method yields accurate results for a wide range of step sizes, from $h = 10^{-2}$ to $h = 10^{-200}$ in this case.

The results corresponding to the complete shape sensitivity vector are shown in Figure 5. Although many different sets of finite-difference results were obtained, only the set corresponding to the optimum step is shown. The plot shows no discernible difference between the two sets of results. Note that these sensitivities account for both aerodynamic and structural effects: a variation in the shape of the wing affects the flow and the underlying structure directly. There is also an indirect effect on the flow due to the fact that the wing exhibits a new displacement field when its shape is perturbed.

Table II. Normalized Computational Cost Comparison for the Calculation of the Complete Shape Sensitivity Vector

Computation Type	Normalized Cost
Aero-structural Solution	1.0
Finite difference	14.2
Complex step	34.4

A comparison of the relative computational cost of the two methods was also performed for the aerodynamic sensitivities, namely for the calculation of the complete shape sensitivity vector. Table II lists these costs, normalized with respect to the solution time of the aero-structural solver.

The cost of a finite-difference gradient evaluation for the 18 design variables is about 14 times the cost of a single aero-structural solution for computations that have converged to six orders of magnitude in the average density residual. Notice that one would expect this method to incur a computational cost equivalent to 19 aero-structural solutions (the solution of the baseline configuration plus one flow solution for each design variable perturbation.) The cost is lower than this value because the additional calculations start from the previously converged solution.

The cost of the complex-step procedure is more than twice of that of the finite-difference procedure since the function evaluations require complex arithmetic. We feel, however, that the complex-step calculations are worth this cost penalty since there is no need to find an acceptable step size *a priori*, as in the case of the finite-difference approximations.

Again, we would like to emphasize that while there was considerable effort involved in obtaining reasonable finite-difference results by optimizing the step sizes, no such effort was necessary when using the complex-step method.

4.2 Supersonic Natural Laminar Flow Analysis

The second example illustrates how the complex-step method can be applied to an analysis for which it is very difficult to extract accurate finite-difference gradients. This code was developed for supporting design work of the supersonic natural laminar flow (NLF) aircraft concept [Sturdza et al. 1999]. It is a multidisciplinary framework, which uses input and output file manipulations to combine five computer programs including an iterative Euler solver and a boundary layer solver with transition prediction. In this framework, Python is used as the gluing language that joins the many programs. Gradients are computed with the complex-step method and with algorithmic differentiation in the Fortran, C++ and Python programming languages.

The sample sensitivity was chosen to be that of the skin-friction drag coefficient with respect to the root chord. A comparison between finite-central-difference and complex-step sensitivities is shown in Figure 6. The plot shows the rather poor accuracy of the finite-difference gradients for this analysis.

There are several properties of this analysis that make it difficult to extract useful finite-difference results. The most obvious is the transition from laminar to turbulent flow. It is difficult to truly smooth the movement of the transition

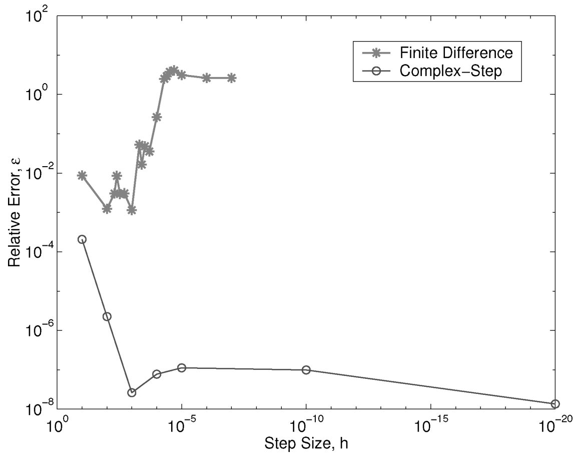

Fig. 6. Convergence of gradients as step size is decreased. $\varepsilon = (|f - f_{ref}|)/|f_{ref}|$; reference is complex-step result at $h = 10^{-30}$.

front when transition prediction is computed on a discretized domain. Since transition has such a large effect on skin friction, this difficulty is expected to adversely affect finite-difference drag sensitivities. Additionally, the discretization of the boundary layer by computing it along 12 arcs that change curvature and location as the wing planform is perturbed is suspected to cause some noise in the laminar solution as well [Sturdza et al. 1999].

5. CONCLUSIONS

The complex-step method and its application to real-world numerical algorithms was presented. We established that this method is equivalent to the forward mode of algorithmic differentiation. This enables the application of a substantial body of knowledge—that of the algorithmic differentiation literature—to the complex-step method.

The implementation of the complex-step derivative approximation has successfully been automated using a simple Python script in conjunction with operator overloading for two large solvers, including an iterative one. The resulting derivative estimates were validated by comparison with finite-difference results.

We have shown that the complex-step method, unlike finite differencing, has the advantage of being step-size insensitive and for small enough steps, the accuracy of the sensitivity estimates is only limited by the numerical precision of the algorithm. This is the main advantage of this method in comparison with finite differencing: there is no need to compute a given sensitivity repeatedly to find out the optimal step that yields the minimum error in the approximation.

The examples presented herein illustrate these points clearly and put forward two excellent uses for the method: validating a more sophisticated gradient calculation scheme and providing accurate and smooth gradients for analyses that accumulate substantial computational noise.

Despite the fact that algorithmic differentiation tools are much more sophisticated in terms of both functionality and computational efficiency, the complex-step method remains a simple and elegant alternative for the purposes mentioned above. Users that are interested only in the functionality of the forward mode of algorithmic differentiation will find the complex-step a compelling alternative due to its simplicity of implementation and the potential to obtain sensitivity information with very low turnaround time. In the authors' experience with a broad range of sensitivity analysis methods, the complex-step method is the simplest derivative calculation method to implement and run, particularly in projects that mix programming languages, require multidisciplinary analysis, or both.

REFERENCES

ANDERSON, W. K., NEWMAN, J. C., WHITFIELD, D. L., AND NIELSEN, E. J. 1999. Sensitivity analysis for the Navier–Stokes equations on unstructured meshes using complex variables. AIAA Paper 99–3294.

BARTHOLOMEW-BIGGS, M. 1995. *OPFAD—A Users Guide to the OPtima Forward Automatic Differentiation Tool*. Numerical Optimization Centre, University of Hertfordsshire.

BECK, T. 1994. Automatic differentiation of iterative processes. *J. Comput. Appl. Math. 50*, 109–118.

BECK, T. AND FISCHER, H. 1994. The if-problem in automatic differentiation. *J. Comput. Appl. Math. 50*, 119–131.

BENDTSEN, C. AND STAUNING, O. 1996. FADBAD, A flexible C++ package for automatic differentiation—using the forward and backward methods. Tech. Rep. IMM-REP-1996-17, Technical University of Denmark, DK-2800 Lyngby, Denmark.

BERZ, M. 1987. The differential algebra FORTRAN precompiler DAFOR. Tech. Rep. AT–3: TN–87–32, Los Alamos National Laboratory, Los Alamos, N.M.

BISCHOF, C., CARLE, A., CORLISS, G., GRIENWANK, A., AND HOVELAND, P. 1992. ADIFOR: Generating derivative codes from Fortran programs. *Sci. Prog. 1*, 1, 11–29.

BISCHOF, C., CORLISS, G., AND GRIENWANK, A. 1991. ADIFOR exception handling. Tech. Rep. MCS-TM-159, Argonne Technical Memorandum.

BISCHOF, C. H., ROH, L., AND MAUER-OATS, A. J. 1997. ADIC: An extensible automatic differentiation tool for ANSI-C. *Softw. — Prac. Exp. 27*, 12, 1427–1456.

BROWN, S. 1995. *OPRAD—A Users Guide to the OPtima Reverse Automatic Differentiation Tool*. Numerical Optimization Centre, University of Hertfordsshire.

CORLISS, G., FAURE, C., GRIEWANK, A., HASCOET, L., AND NAUMANN, U., EDS. 2001. *Automatic Differentiation: From Simulation to Optimization*. Springer.

FAURE, C. AND PAPEGAY, Y. 1997. *Odyssée Version 1.6. The Language Reference Manual*. INRIA. Rapport Technique 211.

GIERING, R. 1997. *Tangent Linear and Adjoint Model Compiler Users Manual*. Max-Planck Institut für Meteorologie.

GILL, P. E., MURRAY, W., AND WRIGHT, M. H. 1981. *Practical Optimization*, Chapter 2, pp. 11–12. Academic Press, San Diego, Calif.

GRIEWANK, A. 2000. *Evaluating Derivatives*. SIAM, Philadelphia, Pa.

GRIEWANK, A., BISCHOF, C., CORLISS, G., CARLE, A., AND WILLIAMSON, K. 1993. Derivative convergence for iterative equation solvers. *Optim. Meth. Softw. 2*, 321–355.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. 22*, 2 (June), 131–167.

HORWEDEL, J. E. 1991. *GRESS Version 2.0 User's Manual*. ORNL. TM-11951.

KUBOTA, K. 1996. *PADRE2—FORTRAN Precompiler for Automatic Differentiation and Estimates of Rounding Errors*. SIAM, Philadelphia, Pa., pp. 367–374.

LUTZ, M. 1996. *Programming Python*. O'Reilly & Associates, Inc., Cambridge, Mass.

Lyness, J. N. 1967. Numerical algorithms based on the theory of complex variable. In *Proceedings of the ACM National Meeting* (Washington, D.C.). ACM New York, pp. 125–133.

Lyness, J. N. and Moler, C. B. 1967. Numerical differentiation of analytic functions. *SIAM J. Numer. Anal. 4*, 2 (June), 202–210.

Martins, J. R. R. A. 2002. *A Coupled-Adjoint Method for High-Fidelity Aero-Structural Optimization*. Ph.D. dissertation. Stanford University, Stanford, CA 94305.

Martins, J. R. R. A. 2003. A Guide to the Complex-Step Derivative Approximation. http://mdolab.utias.utoronto.ca.

Martins, J. R. R. A., Alonso, J. J., and Reuther, J. J. 2002. Complete configuration aero-structural optimization using a coupled sensitivity analysis method. AIAA Paper 2002-5402 (Sept.).

Martins, J. R. R. A., Kroo, I. M., and Alonso, J. J. 2000. An automated method for sensitivity analysis using complex variables. AIAA Paper 2000-0689 (Jan.).

Martins, J. R. R. A., Sturdza, P., and Alonso, J. J. 2001. The connection between the complex-step derivative approximation and algorithmic differentiation. AIAA Paper 2001-0921 (Jan.).

Newman, J. C., Anderson, W. K., and Whitfield, L., D. 1998. Multidisciplinary sensitivity derivatives using complex variables. Tech. Rep. MSSU-COE-ERC-98-08 (July), Computational Fluid Dynamics Laboratory.

Olver, F. W. J. 1983. *Error Analysis of Complex Arithmetic*. Reidel, Dordrecht, Holland, pp. 279–292.

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

Reuther, J., Alonso, J. J., Jameson, A., Rimlinger, M., and Saunders, D. 1999a. Constrained multipoint aerodynamic shape optimization using an adjoint formulation and parallel computers: Part I. *J. Aircraft 36*, 1, 51–60.

Reuther, J., Alonso, J. J., Martins, J. R. R. A., and Smith, S. C. 1999b. A coupled aero-structural optimization method for complete aircraft configurations. AIAA Paper 99–0187.

Reuther, J., Alonso, J. J., Vassberg, J. C., Jameson, A., and Martinelli, L. 1997. An efficient multiblock method for aerodynamic analysis and design on distributed memory systems. AIAA Paper 97–1893 (June).

Rhodin, A. 1997. IMAS—Integrated Modeling and Analysis System for the solution of optimal control problems. *Comput. Phys. Commun. 107* (Dec.), 21–38.

Shiriaev, D. 1996. ADOL–F automatic differentiation of Fortran codes. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. H. Bischof, G. F. Corliss, and A. Griewank, Eds., SIAM, Philadelphia, Pa., pp. 375–384.

Squire, W. and Trapp, G. 1998. Using complex variables to estimate derivatives of real functions. *SIAM Rev. 40*, 1 (Mar.), 110–112.

Sturdza, P., Manning, V. M., Kroo, I. M., and Tracy, R. R. 1999. Boundary layer calculations for preliminary design of wings in supersonic flow. AIAA Paper 99–3104 (June).

Yao, J., Alonso, J. J., Jameson, A., and Liu, F. 2001. Development and validation of a massively parallel flow solver for turbomachinery flow. *J. Prop. Power 17*, 3 (June), 659–668.

Received November 2001; revised February 2003; accepted April 2003