5.4 Algorithmic Differentiation

Algorithmic differentiation — also known as computational differentiation or automatic 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 an analytic method, it is potentially much easier to implement since this can be done automatically.

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*).

When using the forward mode, for each intermediate variable in the algorithm, a variation due to one input variable is carried through. This is very similar to the way the complex-step method works. To illustrate this, suppose we want to differentiate the multiplication operation, $f = x_1 x_2$, with respect to x_1. Table 1 compares how the differentiation would be performed using either algorithmic differentiation or the complex-step method.

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

Table 1: The differentiation of the multiplication operation $f = x_1 x_2$ with respect to x_1 using algorithmic differentiation and the complex-step derivative approximation.

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

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

The other mode — the *reverse mode* — has no equivalent in the complex-step method. When using the reverse mode, the code is executed forward and then backward to calculate derivatives of one output with respect to n inputs. The total number of operations is independent of n, but the memory requirements may be prohibitive, especially for the case of large iterative algorithms.

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

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1). \tag{1}$$

The algorithm that would calculate this function is shown below, together with the derivative calculation using the forward mode.

$$\begin{aligned}
t_1 &= x_1 & \Delta t_1 &= 1 \\
t_2 &= x_2 & \Delta t_2 &= 0 \\
t_3 &= t_1 t_2 & \Delta t_3 &= \Delta t_1 t_2 + t_1 \Delta t_2 \\
t_4 &= \sin(t_1) & \Delta t_4 &= \Delta t_1 \cos(t_1) \\
t_5 &= t_3 + t_4 & \Delta t_5 &= \Delta t_3 + \Delta t_4
\end{aligned}$$

The reverse mode is also based on the chain rule. Let t_j note all the intermediate variables in

an algorithm that calculates $f(x_i)$. We set $t_1, \ldots, t_n$ to $x_1, \ldots, x_n$ and the last intermediate variable, t_m to f. Then the chain rule can be written as,

$$\frac{\partial t_j}{\partial t_i} = \sum_{k \in \mathcal{K}_j} \frac{\partial t_j}{\partial t_k} \frac{\partial t_k}{\partial t_i}, \quad i = 1, 2, \ldots, m, \tag{2}$$

for $j = m+1, \ldots, n$ to obtain the gradients of the intermediate and output variables. $\mathcal{K}_j$ denotes the set of indices $k < j$ such that the variable t_j in the code depends explicitly on t_k. In order to know in advance what these indices are, we have to form the *graph* of the algorithm when it is first executed. This provides information on the interdependence of all the intermediate variables. A graph for our sample algorithm is shown in Figure 1.

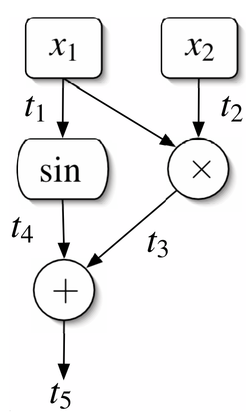

Figure 1: Graph of the algorithm that calculates $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$

The sequence of calculations shown below corresponds to the application of the reverse mode to our simple function.

$$
\begin{aligned}
\frac{\partial t_5}{\partial t_5} &= 1 \\
\frac{\partial t_5}{\partial t_4} &= \frac{\partial t_5}{\partial t_4} = 1 \\
\frac{\partial t_5}{\partial t_3} &= \frac{\partial t_5}{\partial t_4}\frac{\partial t_4}{\partial t_3} + \frac{\partial t_5}{\partial t_3}\frac{\partial t_3}{\partial t_3} + = 1 \cdot 0 + 1 \cdot 1 = 1 \\
\frac{\partial t_5}{\partial t_2} &= \frac{\partial t_5}{\partial t_3}\frac{\partial t_3}{\partial t_2} + \frac{\partial t_5}{\partial t_4}\frac{\partial t_4}{\partial t_2} = 1 \cdot t_1 + 1 \cdot 0 = t_1 \\
\frac{\partial t_5}{\partial t_1} &= \frac{\partial t_5}{\partial t_2}\frac{\partial t_2}{\partial t_1} + \frac{\partial t_5}{\partial t_3}\frac{\partial t_3}{\partial t_1} + \frac{\partial t_5}{\partial t_4}\frac{\partial t_4}{\partial t_1} = t_1 \cdot 0 + 1 \cdot t_2 + 1 \cdot \cos(t_1) = t_2 + \cos(t_1)
\end{aligned}
$$

The following matrix, helps to visualize the sensitivities of all the variables with respect to each other.

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
\frac{\partial t_3}{\partial t_1} & \frac{\partial t_3}{\partial t_2} & 1 & 0 & 0 \\
\frac{\partial t_4}{\partial t_1} & \frac{\partial t_4}{\partial t_2} & \frac{\partial t_4}{\partial t_3} & 1 & 0 \\
\frac{\partial t_5}{\partial t_1} & \frac{\partial t_5}{\partial t_2} & \frac{\partial t_5}{\partial t_3} & \frac{\partial t_5}{\partial t_4} & 1
\end{bmatrix}
\tag{3}
$$

In the case of the example we are considering we have:

$$
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
t_2 & t_1 & 1 & 0 & 0 \\
\cos(t_1) & 0 & 0 & 1 & 0 \\
t_2 + \cos(t_1) & t_1 & 1 & 1 & 1
\end{bmatrix}
\tag{4}
$$

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.

Tools for Algorithmic Differentiation

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

To implement algorithmic 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 constitutes 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.

In order to use derived types, we need languages that support this feature, such as Fortran 90 or C++. To implement algorithmic 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 algorithmic 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 [3], TAMC, DAFOR, GRESS, Odyssée and PADRE2. 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 algorithmic differentiation also exist for C/C++[1]. 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.

5.5 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.

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, $k = 1, \ldots, N_\mathcal{R}$

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

$$f = f(x_n, y_i), \tag{5}$$

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

$$\mathcal{R}_k\left(x_n, y_i\left(x_n\right)\right) = 0. \tag{6}$$

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.

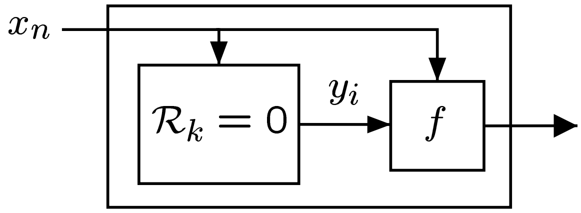

Figure 2: Schematic representation of the governing equations ($\mathcal{R}_k = 0$), design variables or inputs (x_j), state variables (y_i) and the function of interest or output (f).

A graphical representation of the system of governing equations is shown in Figure 2, 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

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} + \frac{\partial f}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n}, \tag{7}$$

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 (7) 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 (6) 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,

$$\frac{\mathrm{d}\mathcal{R}_k}{\mathrm{d}x_n} = \frac{\partial \mathcal{R}_k}{\partial x_n} + \frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = 0, \tag{8}$$

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 (8) as

$$\frac{\partial \mathcal{R}_k}{\partial y_i}\frac{\mathrm{d}y_i}{\mathrm{d}x_n} = -\frac{\partial \mathcal{R}_k}{\partial x_n}, \tag{9}$$

we can solve for $\mathrm{d}y_i/\,\mathrm{d}x_n$ and substitute this result into the total derivative equation (7), to obtain

$$\frac{\mathrm{d}f}{\mathrm{d}x_n} = \frac{\partial f}{\partial x_n} - \underbrace{\frac{\partial f}{\partial y_i}\overbrace{\left[\frac{\partial \mathcal{R}_k}{\partial y_i}\right]^{-1}\frac{\partial \mathcal{R}_k}{\partial x_n}}^{-\,\mathrm{d}y_i/\,\mathrm{d}x_n}}_{-\Psi_k}. \tag{10}$$

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.

Direct Sensitivity Equations

The approach where we first calculate $\mathrm{d}y_i/\,\mathrm{d}x_n$ using equation (9) and then substitute the result in the expression for the total sensitivity (10) is called the *direct* method. Note that solving for $\mathrm{d}y_i/\,\mathrm{d}x_n$ requires the solution of the matrix equation (9) *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.

Adjoint Sensitivity Equations

Returning to the total sensitivity equation (10), 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*

$$\frac{\partial\mathcal{R}_k}{\partial y_i}\Psi_k = -\frac{\partial f}{\partial y_i}. \tag{11}$$

The vector Ψ_k is usually called the *adjoint vector* and is substituted into equation (10) 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.

Direct vs. Adjoint

We can now see that the choice of the solution procedure (direct vs. adjoint) to obtain the total sensitivity (10) 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 5.5. 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 (9) and (11) 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 (9) and (11) 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 3 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.

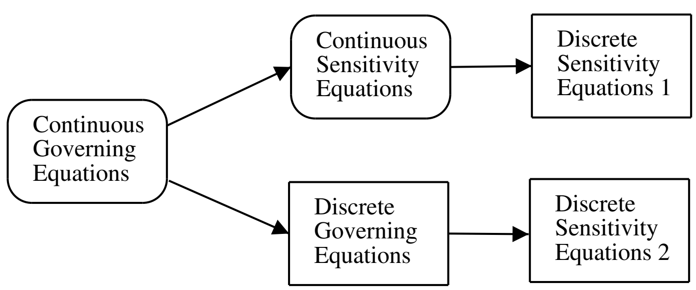

Figure 3: The two ways of obtaining the discretized sensitivity equations

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.

Example: Structural Sensitivity Analysis

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

$$\mathcal{R}_k = K_{ki}u_i - F_k = 0, \tag{12}$$

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,

$$\sigma_m = S_{mi}u_i. \tag{13}$$

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 (10).

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.,

$$\frac{\partial \mathcal{R}_k}{\partial y_i} = \frac{\partial (K_{ki}u_i - F_k)}{\partial u_i} = K_{ki}. \tag{14}$$

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,

$$\frac{\partial \mathcal{R}_k}{\partial x_j} = \frac{\partial (K_{ki}u_i - F_k)}{\partial A_j} = \frac{\partial K_{ki}}{\partial A_j}u_i \tag{15}$$

The partial derivative of the stress with respect to the displacements is simply given by the matrix in equation (13), i.e.,

$$\frac{\partial f_m}{\partial y_i} = \frac{\partial \sigma_m}{\partial u_i} = S_{mi} \tag{16}$$

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

$$\frac{\partial f_m}{\partial x_j} = \frac{\partial \sigma_m}{\partial A_j} = 0. \tag{17}$$

Substituting these into the generalized total sensitivity equation (10) we get:

$$\frac{d\sigma_m}{dA_j} = -\frac{\partial \sigma_m}{\partial u_i} K_{ki}^{-1} \frac{\partial K_{ki}}{\partial x_j} u_i \tag{18}$$

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

$$K_{ki}\frac{du_i}{dA_j} = -\frac{\partial K_{ki}}{\partial A_j}u_i \tag{19}$$

and then substitute the result in,

$$\frac{d\sigma_m}{dA_j} = \frac{\partial \sigma_m}{\partial A_j} + \frac{\partial \sigma_m}{\partial u_i}\frac{du_i}{dA_j} \tag{20}$$

to calculate the desired sensitivities.

The adjoint method could also be used, in which case we would solve equation (11) for the structures case,

$$K_{ki}^T \psi_k = \frac{\partial \sigma_m}{\partial u_i}. \tag{21}$$

Then we would substitute the adjoint vector into the equation,

$$\frac{d\sigma_m}{dx_j} = \frac{\partial \sigma_m}{\partial x_j} + \psi_k^T \left(-\frac{\partial K_{ki}}{\partial A_j} u_i \right). \tag{22}$$

to calculate the desired sensitivities.

References

[1] `http://www.sc.rwth-aachen.de/Research /AD/subject.html`

[2] Adelman, H. M., R. T. Haftka, "Sensitivity Analysis of Discrete Structural Systems", *AIAA Journal*, Vol. 24, No. 5, May 1986.

[3] Bischof, C., A. Carle, G. Corliss, A. Grienwank, P. Hoveland, "ADIFOR: Generating Derivative Codes from Fortran Programs", *Scientific Programming*, Vol. 1, No. 1, 1992, pp. 11-29.

[4] Barthelemy, B., R. T. Haftka and G. A. Cohen, "Physically Based Sensitivity Derivatives for Structural Analysis Programs", *Computational Mechanics*, pp. 465-476, Springer-Verlag, 1989.

[5] Belegundu, A. D. and J. S. Arora, "Sensitivity Interpretation of Adjoint Variables in Optimal Design", *Computer Methods in Applied Mechanics and Engineering*, Vol. 48, pp. 81-90, 1985.

[6] Sobieszczanski-Sobieski, J., "Sensitivity of Complex, Internally Coupled Systems", *AIAA Journal*, Vol. 28, No. 1., January 1990.

[7] Hajela, P., C. L. Bloebaum and J. Sobieszczanski-Sobieski, "Application of Global Sensitivity Equations in Multidisciplinary Aircraft Synthesis", *Journal of Aircraft*, Vol. 27, No. 12., pp. 1002-1010, December 1990.