SIAM J. SCI. COMPUT.
Vol. 22, No. 1, pp. 39–62

ON EFFICIENT SOLUTIONS TO THE CONTINUOUS SENSITIVITY EQUATION USING AUTOMATIC DIFFERENTIATION*

JEFF BORGGAARD† AND ARUN VERMA‡

Abstract. Shape sensitivity analysis is a tool that provides quantitative information about the influence of shape parameter changes on the solution of a partial differential equation (PDE). These shape sensitivities are described by a continuous sensitivity equation (CSE). Automatic differentiation (AD) can be used to perform this sensitivity analysis without writing any additional code to solve the sensitivity equation. The approximate solution of the PDE uses a spatial discretization (mesh) that often depends on the shape parameters. Therefore, the straightforward application of AD introduces derivatives of the mesh. There are two drawbacks to this approach. First, extra computational effort (especially memory) is used in these calculations due to mesh sensitivities. Second, this mesh sensitivity information needs to be computed in order to obtain accurate results. In this work, we provide a methodology that avoids mesh sensitivities (and their drawbacks) by defining a modified PDE on a fixed domain (i.e., independent of the shape parameter) such that AD provides the desired approximation of the CSE. Using two examples, we demonstrate significant improvement in the computational effort, both in terms of floating point operations and memory requirements. We explain how these code modifications can be applied to a wide variety of practical problems with minimal changes to the original code. These changes are negligible when compared to the complexity of writing a separate solver for the sensitivity equation.

Key words. small shape optimization, automatic differentiation, continuous sensitivity equation, shape sensitivity, mesh sensitivity, boundary conditions, ADMAT, ADMIT

AMS subject classifications. 65K10, 65Y20, 35J99, 65N45

PII. S1064827599352136

1. Introduction.

Shape optimization problems frequently take the form of finding parameters that describe the shape of an object or a region in order to minimize a given design objective (such as weight or drag). In many practical situations, the behavior of *states* in the system can be modeled as the solution to a partial differential equation (PDE). Calculating the dependence of the state solution on these shape design parameters (the so-called *state sensitivity*) is of interest to design engineers who want to gain more insight into their problem. Furthermore, these sensitivity variables can be used to evaluate the gradient of the objective function (and other constraint functions) at a given design, which can be readily coupled with an optimization algorithm in an attempt to find the optimal parameter values.

The state sensitivity variables satisfy the continuous sensitivity equation (CSE), which can be derived formally by implicit differentiation of the state PDE and the corresponding boundary conditions [4]. This sensitivity equation is always *linear* and *shares structure* with the state PDE. In particular, these coupled PDEs share the same linearization and boundary condition type (Dirichlet, Neumann, etc.). Nearly all problems of interest require numerical techniques in order to approximate the

*Received by the editors February 23, 1999; accepted for publication (in revised form) December 3, 1999; published electronically June 13, 2000. This work was supported in part by the Air Force Office of Scientific Research under grant F49620-96-1-0329, the National Science Foundation under grants ASC-9704685, DMS-9508773, and DMS-9704509, and the Department of Energy under grant DE-FG02-90ER25013.A000.

http://www.siam.org/journals/sisc/22-1/35213.html

†Interdisciplinary Center for Applied Mathematics, Department of Mathematics, Virginia Tech, Blacksburg, VA 24061 (jborggaard@vt.edu).

‡Cornell Theory Center and Department of Computer Science, Cornell University, Ithaca, NY 14850 (verma@cs.cornell.edu).

solutions to these equations. Because these equations share the same structure, many computations which would be used to solve the state PDE alone can be reused in the solution of the sensitivity equation. Thus, obtaining the sensitivity variables can be performed for a fraction of the cost of computing the state variables.

In many cases, the software to solve the state PDE is already available, representing years of development and testing. This software needs to be modified in order to solve the coupled system. This is often impractical for a variety of reasons. Design engineers are often not experts on the simulation software, which may contain "legacy code" (where the software developer is no longer available to consult on code modifications) or "spaghetti code" (where the structure of the code is fragile or poor), making the necessary modification very difficult. In addition, the resources required to modify and debug an existing code may not be available. Automatic differentiation (AD) tools have been developed to simplify this process as they read in the original code and produce new software to solve for the state and sensitivity variables simultaneously [1, 17].

We point out that we have alluded to two fundamentally different approaches for computing the sensitivity variables. In the first, we derive the sensitivity equation and then apply solution techniques, i.e., we *differentiate-then-approximate*. In the second, we consider the traditional application of automatic differentiation. Essentially, we *approximate-then-differentiate* the state PDE. The operations of differentiation and approximation commute in many situations. For example, this fact is often exploited when developing algorithms for nonlinear problems [18]. However, when the shape of the boundary is parameter dependent, the discretization of the domain (finite difference mesh points, finite element nodes, etc.) depends upon these parameters. Derivatives of these discretization quantities, the so-called *mesh sensitivities*, appear when using the approximate-then-differentiate approach; however, they do not arise when differentiation occurs first. The result is that traditional application of AD in these problems introduces a number of "chain-rule"-like terms that contain the mesh sensitivities. Not only is there an increased overhead in calculating the mesh sensitivities themselves, but computing all of the additional terms that contain them adds significant computational effort. Some related issues in the ODE setting are discussed in [14].

Recent studies in the use of AD [11] have shown that the exploitation of the structure of the underlying algorithms can greatly enhance the performance of these tools and produce, in certain instances, code that approaches the performance of hand-coded algorithms. This requires that the design engineer expose the structure of the problem to the AD tools, and this may require minor code modifications. Finding ways to exploit the algorithm structure is currently an area of active research [11, 9, 10, 21].

In this work, we present a technique for exploiting structure at the problem level rather than at the algorithm level. This technique usually requires only minor code modification, specifically in how the boundary conditions and forcing functions are implemented. The modification is performed *before* handing the software over to AD tools. These changes essentially lead to approximating the CSE and avoid altogether the mesh derivatives and the chain-rule terms which contain them. The result is improved efficiency and a substantial reduction in memory requirements for the final differentiated code. Furthermore, eliminating mesh derivatives allows straightforward coupling of this technique with any adaptive mesh refinement strategy, a useful feature in automatic design optimization algorithms [5, 6, 8].

The remainder of this work is organized as follows. In the next section, we provide background on different techniques for computing the state sensitivities and the relationships between them. To facilitate this discussion, we introduce two example problems. The first is a nonlinear two-point boundary value problem solved by a finite element method and the second is the Laplace equation in two-dimensional coordinates solved by a finite difference method. These allow us to describe our present AD methodology in section 3. In section 4, this methodology is applied and the results demonstrate its effectiveness on these example problems. Finally, we present our conclusions and extensions in section 5.

2. Background.

2.1. Example: Nonlinear two-point boundary value problem. As our first example, we consider a nonlinear two-point boundary value problem in the interval $(0, a)$, $a > 1$,

$$\mathcal{A}_a(u(x;a)) \equiv \frac{\partial^2}{\partial x^2}u(x;a) + \frac{1}{8}\left[\frac{\partial}{\partial x}u(x;a)\right]^3 = 0 \tag{1}$$

with boundary conditions

$$u(x=0;a) = 0 \qquad \text{and} \qquad u(x=a;a) = 4.$$

The subscript on $\mathcal{A}_a$ indicates dependence of the operator on the parameter a, which occurs either through the boundary conditions or, by introducing a mapping function, through variable coefficients in the mapped operator (the operator obtained by mapping the problem to a fixed domain, e.g., (0,1) [7]).

This example was used in [7, 19] since it has sufficient complexity (nonlinear with a parameter-dependent domain) as well as a closed form solution:

$$u_{\text{ex}}(x;a) = 4\sqrt{x + \frac{(a-1)^2}{4}} - 2(a-1).$$

CSE. We define the state sensitivity for the problem (1) to be $s = \frac{\partial u}{\partial a}$. This quantity describes the behavior of the solution as the right end point (parameterized by a) is changed. As shown in [7], s solves the CSE

$$\mathcal{L}_a(u)s \equiv \frac{\partial^2}{\partial x^2}s(x;a) + \frac{3}{8}\left[\frac{\partial}{\partial x}u(x;a)\right]^2 s(x;a) = 0 \tag{2}$$

with boundary conditions

$$s(x=0;a) = 0 \qquad \text{and} \qquad s(x=a;a) = -\frac{\partial}{\partial x}u(x=a;a).$$

The term $\mathcal{L}_a(u)s$ is the linearization of $\mathcal{A}_a$ at u applied to s.

Boundary conditions for (2) are derived by setting the total (or material) derivative of u with respect to a to zero,

$$\frac{Du}{Da}(x;a) = \frac{\partial u}{\partial a}(x;a) + \frac{\partial u}{\partial x}(x;a)\Pi_a(x;a) = s(x;a) + \frac{\partial u}{\partial x}(x;a)\Pi_a(x;a) = 0,$$

where Π_a represents the derivative of the spatial location with respect to a. This term arises since the location where the boundary condition is evaluated is parameter dependent. For this problem $\Pi_a(x = 0; a) = 0$ and $\Pi_a(x = a; a) = 1$, leading to the boundary conditions in (2). An important observation is that Π_a only needs to be defined and evaluated on the boundary. Since we specify the parameterization of the shape, this quantity is well defined. It can be verified that differentiating u_{ex} with respect to a provides a closed form solution to (2) and satisfies the boundary conditions (using u_{ex}).

Note that the sensitivity equation is a linear PDE. For the case above where $\mathcal{A}_a$ is nonlinear, this equation requires the solution u in order to define $\mathcal{L}_a$. Additionally, for shape sensitivity problems ($\Pi_a \neq 0$), u is required to compute the boundary conditions. Therefore, the solution to (1) needs to be obtained *before* the sensitivity equation can be solved.

Finite element algorithm. To illustrate our AD methodology in section 3, we consider a finite element approximation to the above equations. The domain $(0, a)$ is subdivided into N elements which allow for either a piecewise linear or piecewise quadratic representation of the solution. The standard Galerkin procedure leads to a system of nonlinear algebraic equations which are solved using Newton's method. Convergence is declared when the ℓ_2-norm of the residual drops below 10^{-8}. Note that (1) and (2) share the same linearization and boundary condition types. Therefore, using a Newton method to solve the nonlinear finite element system for (1) produces the same system matrix as the finite element solution for (2) (using the previous iterate for u). The solution to the second linear system can be performed efficiently by solving the Newton system with an LU factorization and reusing this factorization to solve the second system. However, there are often computational advantages in using other linear system solvers, such as the conjugate gradient method. Thus, in many cases, code modifications to solve both linear systems may not be able to take advantage of this structure.

These algorithms are implemented in MATLAB. In particular, the Jacobian matrix is stored in the `sparse` data structure in order to minimize storage and CPU requirements. MATLAB vectorization is used in the Jacobian build to create a faster code. MATLAB features such as the sparse data structures and vectorization are exploited in the AD tools.

2.2. Example: Laplace equation in two dimensions. This example features the finite difference solution to Laplace's equation in a parameter-dependent region. The region resembles a unit square with the exception of the bottom edge, which is a multiple of the sine function (see Figure 1). Thus, we define the parameter-dependent portion of the boundary by the parametric curve

$$\Gamma_a = \{(x, a\sin(\pi x)) \mid x \in (0, 1)\}.$$

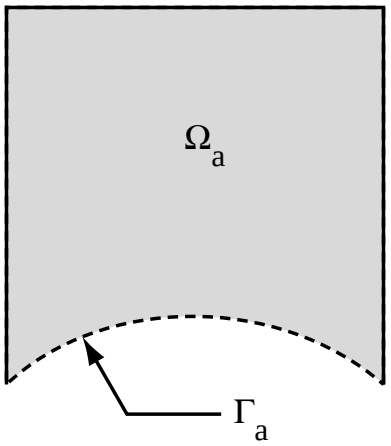

FIG. 1. *Parameter-dependent region.*

Let $\mathcal{A}_a : \mathrm{H}^1(\Omega_a) \to \mathrm{H}^{-1}(\Omega_a)$ be the Laplacian operator, $f \in \mathrm{H}^{-1}(\Omega_a)$ be a prescribed forcing function, and u be the solution to

$$\mathcal{A}_a(u(x,y;a)) \equiv \frac{\partial^2}{\partial x^2}u(x,y;a) + \frac{\partial^2}{\partial y^2}u(x,y;a) = f(x,y;a) \tag{3}$$

subject to

$$\begin{aligned} u(x=0,y;a) = u(x=1,y;a) &= 0, && y \in (0,1), \\ u(x,y=1;a) &= 0, && x \in (0,1), \quad \text{and} \\ u(x,y=a\sin(\pi x);a) &= \sin(\pi x) && x \in (0,1). \end{aligned}$$

Note that although $\mathcal{A}_a$ is linear, we use the notation $\mathcal{A}_a(u)$ to parallel the example in the previous section.

For this example, we define the function f such that u has the closed form solution

$$u_{\mathrm{ex}}(x,y;a) = \frac{1-y}{1-a\sin(\pi x)}\sin(\pi x).$$

Thus, f will be a nontrivial function of a in this example.

CSE. The CSE, which describes the state sensitivity, can be derived by performing implicit differentiation of (3) with respect to a and interchanging orders of differentiation (assuming necessary smoothness). We make an additional assumption that $\frac{\partial f}{\partial a} \in \mathrm{H}^{-1}(\Omega_a)$ so that the solution s has the same regularity as u (as we effectively solve this equation with the same approximation scheme as problem (3)).

$$\mathcal{L}_a(u)s(x,y;a) \equiv \frac{\partial^2}{\partial x^2}s(x,y;a) + \frac{\partial^2}{\partial y^2}s(x,y;a) = \frac{\partial}{\partial a}f(x,y;a) \tag{4}$$

subject to

$$\begin{aligned} s(0,y;a) = s(1,y;a) &= 0, && y \in (0,1),\\ s(x,1;a) &= 0, && x \in (0,1), \end{aligned}$$

and

$$s(x, y = a\sin(\pi x); a) = -\frac{\partial}{\partial y}u(x, y = a\sin(\pi x); a)\Pi_a(x, y = a\sin(\pi x); a), \quad x \in (0,1).$$

For this example, the function describing the influence of the parameter on the boundary, Π_a, is trivial on all sides except for Γ_a. On this side, $\Pi_a(x, y = a\sin(\pi x); a) = \sin(\pi x)$ and appears in the boundary condition above.

Note that $\mathcal{L}_a$ is a differential operator which represents the linearization of $\mathcal{A}_a$ with respect to u. As above, even though $\mathcal{A}_a$ is linear, we'll maintain this notation throughout the paper for easy generalization to nonlinear problems (as in section 2.1).

Finite difference algorithm. For purposes of applying finite differences to solve problem (3), we introduce the mapping $\mathcal{M}_a$, which is a bijection of the form

$$\mathcal{M}_a : \Omega_a \ni (x,y) \to (\xi,\eta) \in (0,1)\times(0,1) \equiv \tilde{\Omega}.$$

Transformation to the square simplifies the construction of high order difference stencils, which can now be set up on a lattice. This mapping procedure also covers the case where finite difference points are clustered in regions where more accurate differences are needed. To treat a wider class of problems with more irregularly shaped domains, domain decomposition can be used to create subdomains, each of which can be mapped in this way. In practice, algorithms for calculating this mapping may require the solution of another PDE [20].

The differential equation in Ω_a is then transformed to $\tilde{\Omega}$ by representing functions in the new coordinates, i.e.,

$$f(x,y;a) = \tilde{f}(\mathcal{M}_a(x,y;a);a) = \tilde{f}(\xi(x,y;a),\eta(x,y;a);a) = \tilde{f}(\xi,\eta;a),$$

and by replacing the differential operators in the obvious way, e.g.,

$$\frac{\partial u}{\partial x} = \frac{\partial \tilde{u}}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial \tilde{u}}{\partial \eta}\frac{\partial \eta}{\partial x}.$$

Transforming problem (3) in this way leads to the equation

$$\tilde{\mathcal{A}}_a\left(\tilde{u}(\xi,\eta;a)\right) = \tilde{f}(\xi,\eta;a)$$

or

$$\begin{aligned} &\left(\left(\frac{\partial \xi}{\partial x}\right)^2 + \left(\frac{\partial \xi}{\partial y}\right)^2\right)\frac{\partial^2 \tilde{u}}{\partial \xi^2} + 2\left(\frac{\partial \xi}{\partial x}\frac{\partial \eta}{\partial x} + \frac{\partial \xi}{\partial y}\frac{\partial \eta}{\partial y}\right)\frac{\partial^2 \tilde{u}}{\partial \xi \partial \eta} + \left(\left(\frac{\partial \eta}{\partial x}\right)^2 + \left(\frac{\partial \eta}{\partial y}\right)^2\right)\frac{\partial^2 \tilde{u}}{\partial \eta^2}\\ &\qquad + \left(\frac{\partial^2 \xi}{\partial x^2} + \frac{\partial^2 \xi}{\partial y^2}\right)\frac{\partial \tilde{u}}{\partial \xi} + \left(\frac{\partial^2 \eta}{\partial x^2} + \frac{\partial^2 \eta}{\partial y^2}\right)\frac{\partial \tilde{u}}{\partial \eta} = f\left(\mathcal{M}_a^{-1}(\xi,\eta);a\right), \end{aligned} \tag{5}$$

subject to

$$\begin{aligned} \tilde{u}(\xi=0,\eta;a) = \tilde{u}(\xi=1,\eta;a) &= 0, && \eta \in (0,1), \\ \tilde{u}(\xi,\eta=1;a) &= 0, && \xi \in (0,1), \quad \text{and} \\ \tilde{u}(\xi,\eta=0;a) &= \sin\left(\pi \mathcal{M}_a^{-1}(\xi,\eta=0)\right), && \xi \in (0,1). \end{aligned}$$

Once the mapping is defined, the PDE in $\tilde{\Omega}$ can be readily approximated since finite difference approximations in $\tilde{\Omega}$ are performed on a lattice,

$$\frac{\partial \tilde{u}}{\partial \xi}(\xi,\eta;a) \approx \frac{u(\xi+\Delta\xi,\eta;a) - u(\xi-\Delta\xi,\eta;a)}{2\Delta\xi}, \text{ etc.}$$

A finite difference method is applied to (5) in the usual manner, considering second order central differences for all of the derivative terms. The resulting numerical approximation to the PDE in $\tilde{\Omega}$ is denoted by $\tilde{u}^N$, representing $\tilde{u}$ at the discrete points of the lattice. The finite difference approximation defines an algebraic equation

$$\tilde{A}_a(\tilde{u}^N) = \tilde{b}. \tag{6}$$

Note that $\tilde{b}$ contains point evaluations of the forcing function and the nonhomogeneous values of the boundary conditions. The solution to (6) can be found by solving the linear system and this solution can be mapped back to Ω_a ($u^N(x,y;a) = \tilde{u}^N(\mathcal{M}(x,y;a);a)$).

2.3. Discrete sensitivity equations and mesh sensitivities. In this section, we introduce the concept of mesh sensitivities and emphasize the differences and similarities between the differentiate-then-approximate and approximate-then-differentiate approaches to obtain state sensitivities. To simplify this discussion, we restrict our attention to the finite difference algorithm from the previous section.

Rather than deriving the sensitivity equation and finding an approximation for it, many practitioners apply implicit differentiation to the discrete form of the equations, e.g., (6), in order to determine the dependence of the solution on the parameter a. Straightforward differentiation leads to

$$\nabla_u \tilde{A}_a(\tilde{u}^N) \frac{\mathrm{D}\tilde{u}^N}{\mathrm{D}a} = \nabla_a \tilde{b} - \tilde{B}_a(\tilde{u}^N). \tag{7}$$

The Jacobian matrix on the left-hand side of the equation above is the linearization of $\tilde{A}_a$ and may be available in factored form if a direct solver is used to solve (6). The term $\nabla_a \tilde{b}$ contains terms arising from differentiating the boundary conditions and forcing functions in $\tilde{b}$. The expression $\tilde{B}_a$ contains terms which involve derivatives of the mapping $\mathcal{M}_a$ with respect to a. In other words, these are terms from the differentiation of the variable coefficients in $\tilde{A}$ that include

$$\frac{\partial \xi}{\partial a}, \ldots, \frac{\partial^2 \xi}{\partial a \partial x}, \ldots, \frac{\partial^3 \xi}{\partial a \partial x^2}, \ldots, \text{etc.}$$

All of the above terms are collectively referred to as the mesh sensitivities, since they represent how the discrete points in Ω (or the mesh) depend on a. It is these quantities and the evaluation of $\tilde{B}_a$ that we wish to avoid altogether.

Of course, it is also possible to apply the finite difference technique directly to the sensitivity equation (4). Thus, we transform the sensitivity equation to $\tilde{\Omega}$ as for (3): s to $\tilde{s}$, the differential operator $\mathcal{L}_a$ to $\tilde{\mathcal{L}}_a$, etc. Linearization of $\mathcal{A}_a$ with respect to u

commutes with the approximation scheme (as alluded to earlier, this is the premise of deriving many Newton algorithms). Thus, $\tilde{L}_a(\tilde{u}^N) = \nabla_u \tilde{A}_a(\tilde{u}^N)$ and the linear system for $\tilde{s}^N$ is

$$\nabla_u \tilde{A}_a(\tilde{u}^N)\tilde{s}^N = \tilde{c}. \tag{8}$$

The right-hand side vector contains the forcing function and boundary conditions (analogous to the construction of $\tilde{b}$).

Note that the linear systems for $\frac{\mathrm{D}\tilde{u}^N}{\mathrm{D}a}$ in (7) and for $\tilde{s}^N$ in (8) share the same system matrix. The differences in the right-hand sides reflect differences in the meaning of $\frac{\mathrm{D}\tilde{u}^N}{\mathrm{D}a}$ and $\tilde{s}^N$. As implied by our notation "D" in (7), when differentiation is performed on the discretized equations, the total (or material) derivative of $\tilde{u}^N$ is computed. Thus, these sensitivities capture the "convective" behavior of the solution as a is varied. This behavior is represented by the first two terms on the right-hand side below,

$$\frac{\mathrm{D}\tilde{u}^N}{\mathrm{D}a} = \frac{\partial \tilde{u}^N}{\partial \xi}\frac{\partial \xi}{\partial a} + \frac{\partial \tilde{u}^N}{\partial \eta}\frac{\partial \eta}{\partial a} + \frac{\partial \tilde{u}^N}{\partial a}.$$

This should be compared to $\tilde{s}^N$, which is the approximation $(\widetilde{\frac{\partial u}{\partial a}})^N$ (the operations on u here are differentiation, then transformation, then approximation). By comparing (7) and (8), we see that differentiating the approximation introduces terms to the calculations from the following identity:

$$\nabla_u \tilde{A}(\tilde{u}^N)\left[\frac{\partial \tilde{u}^N}{\partial \xi}\frac{\partial \xi}{\partial a} + \frac{\partial \tilde{u}^N}{\partial \eta}\frac{\partial \eta}{\partial a}\right] = \nabla_a \tilde{b} - \tilde{c} - \tilde{B}(\tilde{u}^N).$$

In the next section, we outline a methodology that bypasses the above calculations resulting in more efficient sensitivity calculations.

We point out that this methodology will not produce exactly the same end result that would be obtained by performing AD in the traditional fashion. In other words,

$$\left(\widetilde{\frac{\partial u}{\partial a}}\right)^N \neq \frac{\partial \tilde{u}^N}{\partial a},$$

as the first expression contains truncation errors from approximating the sensitivity equation, while the second expression contains the derivative of the truncation errors from approximating the state PDE. These errors will not be the same. However, in many cases, they both vanish as the approximations are refined. This observation motivates the notion of asymptotic consistency [2, 3], justifying the use of the CSE to obtain gradients in an optimal design algorithm.

2.4. AD strategies. AD is a chain-rule-based technique for evaluating the derivatives of functions defined by a high-level language computer program. AD relies on the fact that all computer programs, no matter how complicated, use a finite set of *elementary functions*. The `function` computed by the program is simply a composition of these elementary functions. The partial derivatives of these elementary functions are known, and thus derivatives of the `function` can be computed by propagating these derivatives via the chain rule. An introduction to AD can be found in [16].

Abstractly, the program to evaluate the solution u as a function of a (generally an m-vector) has the form

$$\begin{aligned} a &\equiv (a_1, a_2, \ldots, a_m), \\ &\downarrow \\ z &\equiv (z_1, z_2, \ldots, z_p), \quad p >> m+n, \\ &\downarrow \\ u &\equiv (u_1, u_2, \ldots, u_n), \end{aligned}$$

where the intermediate variables z are related through a series of these elementary functions which may be unary,

$$z_k = f^k_{\text{elem}}(z_i), \qquad i < k,$$

consisting of operations such as ($-$, pow$(\cdot)$, sin$(\cdot)$, $\ldots$), or binary,

$$z_k = f^k_{\text{elem}}(z_i, z_j), \qquad i < k, \quad j < k,$$

such as ($+$, $/$, $\ldots$).

AD has two basic modes of operation, the forward mode and the reverse mode. In the forward mode the derivatives are propagated along with the computation, e.g., in the elementary step $z_k = f^k_{\text{elem}}(z_i, z_j)$, the intermediate derivative, $\frac{dz_k}{da}$, can be propagated in the forward mode as

$$\frac{dz_k}{da} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dz_i}{da} + \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{dz_j}{da}.$$

This propagation is done for all the intermediate variables z and for the output variables u. This process eventually produces the desired derivative $\frac{du}{da}$.

The reverse mode propagates the derivatives $\frac{du}{dz_k}$ for all intermediate variables backward (i.e., in the reverse order) through the computation. For example, in the elementary step $z_k = f^k_{\text{elem}}(z_i, z_j)$, the derivatives are propagated as

$$\frac{du}{dz_i} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{du}{dz_k} \qquad \text{and} \qquad \frac{du}{dz_j} = \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{du}{dz_k}. \tag{9}$$

At the end of this procedure we obtain the derivative $\frac{du}{da}$. However, the reverse mode requires saving the entire function computation, since the propagation is done backward through the computation. Hence, the partials $\frac{\partial f^k_{\text{elem}}}{\partial z_j}, \frac{\partial f^k_{\text{elem}}}{\partial z_i}$ need to be stored for the derivative computation in (9) making the reverse mode potentially prohibitive due to memory requirements.[1]

An application of AD to the discretized PDE solution results in the computation of additional terms involving the mesh. In the reverse mode these terms are stored and subsequently used for the derivative computation. This adds to the computational and memory requirements and adversely affects the performance in the reverse mode. These extra memory requirements are not necessary in the forward mode. The usage of reverse mode of AD is important for calculating gradients; the *cheap gradient*

[1] There are techniques that make the reverse mode efficient; e.g., a checkpointing technique due to Griewank [15] dramatically reduces the memory requirements with little increase in the computational requirements.

theorem [16] tells us that the gradient of any nonlinear function can be calculated for less than the cost of five function evaluations in the reverse mode of AD.

ADMAT (automatic differentiation for MATLAB) is an AD tool that can be applied to functions written for MATLAB [12]. ADMIT-1 is a MATLAB toolbox, which uses a generic AD plug-in tool (e.g., ADMAT or ADOL-C [17]) to compute the gradient and the (possibly sparse) Jacobian and Hessian matrices. The requirements from the users are minimal: the user is simply required to supply the code for the function computation and identify the variables. For complete information on using ADMIT-1, we refer the interested reader to the ADMIT-1 user manual [13]. ADMAT and ADMIT-1 are used to obtain the numerical results in section 4.

3. Methodology: Differentiation of an equivalent problem. In this section, we consider an AD methodology that avoids the need for mesh sensitivities. To facilitate this, we introduce a simple example: Let $f(a)$ be a function that is defined for $a \in \mathcal{P}$. Furthermore, assume f is differentiable over the set $\mathcal{P}$ and define a new function g with independent variables $a \in \mathcal{P}$ and $\alpha \in (-\epsilon, \epsilon)$ as

$$g(a, \alpha) = f(a) + \alpha \frac{df}{da}(a).$$

The following statements are obviously true:

$$\begin{array}{rcll} g(a, \alpha = 0) & = & f(a) & \forall a \in \mathcal{P}, \\ \frac{\partial g}{\partial a}(a, \alpha = 0) & = & \frac{df}{da}(a) & \forall a \in \mathcal{P}. \end{array}$$

Therefore, we have shifted the task of finding the derivative of f with respect to a to finding the derivative of g with respect to α at $\alpha = 0$.

In the sections below, we introduce a new parameter α and define a modified equivalent problem to aid our sensitivity analysis. While the example function g seems contrived, since we need to find $\frac{df}{da}(a)$ in order to construct g in the first place, it motivates how we can construct an equivalent problem to compute the derivatives we desire. This idea is used to construct a modified problem (PDE). Typically, we need only to define modified forcing functions and boundary conditions using the technique above. Since the modified problem is defined at the abstract level before any approximations are considered, it may be possible to perform minor modifications to an existing algorithm to produce a solution to this modified problem. Moreover, unlike the simple contrived example above, this problem definition does not require the sensitivity solution a priori but rather leads to the sensitivity solution. AD of this modified algorithm with respect to the parameter α produces an algorithm for approximating the CSE. The advantages of obviating differentiation of the algorithm with respect to a is that mesh sensitivities are not required. This has significant computational advantages as we demonstrate in section 4.

The procedure is illustrated in Figure 2. The typical application of AD follows the upper right path; first the PDE is approximated and then AD is applied. This is different from the continuous sensitivity approach along the lower left path, which first derives a linear PDE (the CSE) for the sensitivity variables and then performs approximation. The methodology below creates a modified PDE such that the approximation of this modified PDE yields the same result as approximating the original PDE. However, when AD is applied, it produces an approximation to the CSE (and inherits all of the advantages and disadvantages of doing so).

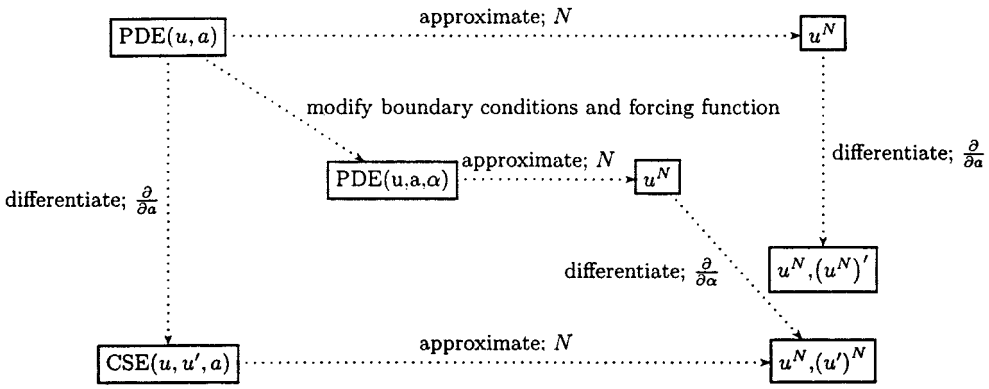

FIG. 2. *Schematic of proposed differentiation methodology.*

3.1. Differentiation methodology. The general form of the two examples from section 2 is to seek a solution u that satisfies

$$\mathcal{A}_a(u) = f \qquad \text{on } \Omega_a \tag{10}$$

subject to the boundary conditions

$$u = \hat{u} \quad \text{on } \Gamma_{\mathrm{d}} \qquad \text{and} \qquad \nabla u \cdot \hat{n} = \hat{q} \quad \text{on } \Gamma_{\mathrm{n}}$$

with $\Gamma_{\mathrm{d}} \cup \Gamma_{\mathrm{n}} = \partial\Omega_a$ and $\Gamma_{\mathrm{d}} \cap \Gamma_{\mathrm{n}} = \emptyset$. The general form of the sensitivity equation is

$$\mathcal{L}_a(u)s = g(u) \qquad \text{on } \Omega_a \tag{11}$$

subject to

$$s = \hat{s}(u) \quad \text{on } \Gamma_{\mathrm{d}} \qquad \text{and} \qquad \nabla s \cdot \hat{n} = \hat{q}_s(u) \quad \text{on } \Gamma_{\mathrm{n}}.$$

Note that the operator $\mathcal{A}_a(u)$ may depend on the parameter a. If this is the case, it results in adding a forcing function to the CSE in addition to $\frac{\partial f}{\partial a}$. This additional forcing function may also be a function of u, and the combined forcing function $g(u)$ reflects this dependence.

Next, we use the parameter α and introduce a differential equation for $v = v(x, y; a, \alpha)$ as follows. First of all, to construct a forcing function, we add α times the forcing function in (11) to the forcing function in (10). Next, we add α times the boundary conditions of the sensitivity equation (11) to the corresponding boundary conditions in the PDE (10). This leads to the following PDE with a solution v:

$$\mathcal{A}_a(v) = f + \alpha g \qquad \text{on } \Omega_a$$

with

$$v = \hat{u} + \alpha\hat{s}(v) \quad \text{on } \Gamma_{\mathrm{d}} \qquad \text{and} \qquad \nabla v \cdot \hat{n} = \hat{q} + \alpha\hat{q}_s(v) \quad \text{on } \Gamma_{\mathrm{n}}.$$

Thus, if we consider applying this to the example in section 2.2, we obtain the PDE

$$\mathcal{A}_a(v) = f + \alpha \frac{\partial f}{\partial a} \tag{12}$$

with

$$\begin{aligned} v(0,y;a,\alpha) = v(1,y;a,\alpha) &= 0, && y \in (0,1), \\ v(x,1;a,\alpha) &= 0, && x \in (0,1), \end{aligned}$$

and

$$v(x, y = a\sin(\pi x); a, \alpha) = \sin(\pi x) - \alpha \tfrac{\partial}{\partial y} v(x, y = a\sin(\pi x); a, \alpha)\sin(\pi x), \quad x \in (0,1).$$

From our assumptions on f and $\frac{\partial f}{\partial a}$, the right-hand side of (12) is in $\mathcal{H}^{-1}$. Observe the following fact.

THEOREM 3.1. *The derivative of the solution of* (3) *with respect to* a *is the same function as the derivative of the solution of* (12) *with respect to* α *at* $\alpha = 0$. *In other words,*

$$\frac{\partial u}{\partial a}(\cdot,\cdot;\cdot) = \frac{\partial v}{\partial \alpha}(\cdot,\cdot;\cdot,\alpha = 0).$$

Proof. Note that when $\alpha = 0$, (12) reduces to (3) (the same operator on the same domain with the same boundary conditions) so they have the same solution, i.e.,

$$u(\cdot,\cdot;\cdot) = v(\cdot,\cdot;\cdot,\alpha = 0).$$

Consider the sensitivity equation for problem (12) with parameter α; then $r = \frac{\partial v}{\partial \alpha}$ satisfies the following PDE:

$$\mathcal{L}_a(u) r = \frac{\partial f}{\partial a} \tag{13}$$

with

$$\begin{aligned} r(0,y;a,\alpha) = r(1,y;a,\alpha) &= 0, && y \in (0,1), \\ r(x,1;a,\alpha) &= 0, && x \in (0,1), \quad \text{and} \\ r(x, y = a\sin(\pi x); a, \alpha) &= -\tfrac{\partial}{\partial y} v(x, y = a\sin(\pi x); a, \alpha)\sin(\pi x), && x \in (0,1). \end{aligned}$$

We now show that the sensitivity of v in (12) with respect to α (our r above) evaluated at $\alpha = 0$ is the same as s in (4). It is easy to see that the derivative of the forcing function in (12) with respect to α gives the forcing function in (4). We now consider the boundary condition expressions in (12). Differentiating the Dirichlet boundary condition in (12) with respect to α leads to

$$r(x,y;a,\alpha) = -\frac{\partial}{\partial y} v(x,y;a,\alpha)\sin(\pi x) - \alpha \frac{\partial}{\partial y} r(x,y;a,\alpha)\sin(\pi x).$$

When $\alpha = 0$ this is precisely the boundary condition that appears in (4) (since $u = v$ as noted above):

$$r(x,y;a,\alpha = 0) = -\frac{\partial}{\partial y} v(x,y;a,\alpha = 0)\sin(\pi x) = -\frac{\partial}{\partial y} u(x,y;a)\sin(\pi x).$$

A similar argument holds for the Neumann boundary condition. □

It is important to point out that the parameter α has no influence on the geometry of the problem, only on the forcing function and the boundary conditions that are applied. Therefore, we propose modifying the software so that it approximates (12) instead of approximating (3). Then, application of AD tools results in software that produces an approximation to the CSE (11) and all of the unnecessary calculations associated with the mesh sensitivities are avoided.

3.2. Implementation details. We now discuss the implementation of this AD methodology for the two examples in section 2. Each example requires that we modify the approximation scheme for the PDE so that it approximates an augmented PDE. Fortunately, these modifications are only necessary in the routines that calculate forcing functions and boundary conditions. These routines typically account for a small percentage of the code.

The modification of the forcing function is straightforward. If we can explicitly differentiate the forcing function with respect to a, then this function can be modified to accept another argument, α, and return the value of $f + \alpha(\frac{\partial f}{\partial a})$. In the event that the code to evaluate f is complicated, AD can be used to compute $\frac{\partial f}{\partial a}$. A new routine, say `f_new`, can be created to call `f_original` (for f) and the differentiated version of this function (to compute $\frac{\partial f}{\partial a}$), then calculate the appropriate linear combination. For example, with the ADMIT toolbox, AD can be performed in a nested fashion to carry this out:

```
function f_new(x,y,a,alpha)
   return f_original(x,y,a) + alpha*f_original(x,y,D(a));
```

where `f_original(x,y,D(a))` is the AD of the function `f_original` with respect to `a`.

The treatment of the boundary conditions is more complex and will be specified for each example shortly. For now, we describe where this complexity arises. If we consider a boundary condition of the form

$$r = \frac{\partial \hat{u}}{\partial a} - \nabla v \cdot \Pi_a$$

(as appears in (13)), this cannot be implemented by differentiating a program with the boundary condition

$$v = \hat{u} + \alpha \left(\frac{\partial \hat{u}}{\partial a} - \nabla v \cdot \Pi_a \right).$$

Since most differentiated code solves for v and r simultaneously, we can't expect v (and ∇v in particular) to be available before we calculate r. Current AD tools generally don't have information about the structure of the problem (where the equations for v and r can be decoupled), so this type of structure needs to be exposed to the AD tools. Therefore, the boundary condition routine needs to be modified so that it imposes the condition

$$v + \alpha \nabla v \cdot \Pi_a = \hat{u} + \alpha \frac{\partial \hat{u}}{\partial a}$$

even though α will eventually be set to zero. Using this modified boundary condition allows us to impose

$$r + \alpha \nabla r \cdot \Pi_a = \frac{\partial \hat{u}}{\partial a} - \nabla v \cdot \Pi_a$$

in the differentiated code (which is our desired boundary condition when $\alpha = 0$).

Application to the nonlinear two-point boundary value problem. We consider the implementation for the problem in section 2.1. In order to approximate the sensitivity equation using our AD tools, we consider the following two-point boundary value problem instead:

$$\frac{\partial^2}{\partial x^2} v(x;a,\alpha) + \frac{1}{8}\left[\frac{\partial}{\partial x} v(x;a,\alpha)\right]^3 = 0 \tag{14}$$

with boundary conditions

$$v(x=0;a,\alpha) = 0 \qquad \text{and} \qquad v(x=a;a,\alpha) + \alpha \frac{\partial}{\partial x} v(x=a;a,\alpha) = 4.$$

Note that changes in the algorithm to compute an approximation to v rather than to u occur only in the implementation of the boundary conditions and that

$$u(x;a) = v(x;a,\alpha=0)$$

and

$$s(x;a) \equiv \frac{\partial u}{\partial a}(x;a) = \frac{\partial v}{\partial \alpha}(x;a,\alpha=0) \equiv r(x;a,\alpha=0).$$

The finite element approximation of (1) leads to a system of nonlinear algebraic equations. These equations can be solved, e.g., using Newton's method. In this case, an update of the solution δu is constructed to improve the current guess of the solution. The boundary conditions for the update are

$$\delta u(x=0;a) = 0 - u(x=0;a)$$

and

$$\delta u(x=0;a) = 4 - u(x=a;a).$$

In the algebraic system, the last boundary condition takes the form

$$\begin{aligned} L(N,N) &= 1, \\ b(N) &= 4 - u(N), \end{aligned}$$

where L is considered to be an approximation to $\mathcal{L}_a(u)$, or the Jacobian of the nonlinear algebraic equations.

For the modified problem, the boundary conditions for the update of v are

$$\delta v(x=0;a,\alpha) = 0 - v(x=0;a,\alpha)$$

and

$$\delta v(x=a;a,\alpha) + \alpha \frac{\partial}{\partial x} \delta v(x=a;a,\alpha) = 4 - \left[v(x=a;a,\alpha) + \alpha \frac{\partial}{\partial x} v(x=a;a,\alpha) \right].$$

The second boundary condition can be implemented as

$$\begin{aligned} L(N,N-1) &= -\frac{\alpha}{\Delta x}, \\ L(N,N) &= 1 + \frac{\alpha}{\Delta x}, \\ b(N) &= 4 - \left(v(N) + \frac{\alpha}{\Delta x}(v(N) - v(N-1)) \right). \end{aligned}$$

Thus, AD can be applied to (14) with respect to α, rather than to problem (1) with respect to a. The required modification only occurs in a few lines of MATLAB code.

Application to the Laplace equation in two dimensions. To use our AD methodology on the problem in section 2.2, we need to modify the finite difference algorithm so that it approximates

$$\frac{\partial^2}{\partial x^2}v(x,y;a,\alpha)+\frac{\partial^2}{\partial y^2}v(x,y;a,\alpha)=f(x,y;a)+\alpha\frac{\partial f}{\partial a}(x,y;a) \tag{15}$$

with boundary conditions

$$\begin{aligned} v(0,y;a,\alpha)=v(1,y;a,\alpha)&=0, \qquad y\in(0,1),\\ v(x,1;a,\alpha)&=0, \qquad x\in(0,1), \end{aligned}$$

and

$$v(x,y=a\sin(\pi x);a,\alpha)+\alpha\tfrac{\partial}{\partial y}v(x,y=a\sin(\pi x);a,\alpha)\sin(\pi x)=\sin(\pi x), \quad x\in(0,1),$$

instead of the system (3) and evaluates this at $\alpha=0$. The code that evaluates f can be modified in a straightforward manner by passing an additional argument for α and returning the new forcing function. In this problem, $f(x,y;a)$ is given by an explicit formula; thus $\frac{\partial f}{\partial a}$ is also known explicitly. Therefore the modification is straightforward as illustrated for the one-dimensional (1-D) problem.

Once again, the modification of the boundary conditions is more challenging. Dirichlet boundary conditions in a finite difference approximation of (3) may be implemented by creating rows in the system matrix that impose

$$\tilde{u}(\xi,\eta=0)=\sin(\pi x(\xi,\eta=0)) \qquad \text{for } \xi=\Delta\xi,2\Delta\xi,\ldots,1-\Delta\xi.$$

The boundary condition for (15) can be implemented by imposing

$$\tilde{v}(\xi,\eta=0)+\alpha\left(\frac{\partial\xi}{\partial y}(\xi,\eta=0)\frac{\partial\tilde{v}}{\partial\xi}(\xi,\eta=0)+\frac{\partial\eta}{\partial y}(\xi,\eta=0)\frac{\partial\tilde{v}}{\partial\eta}(\xi,\eta=0)\right)\sin(\pi x(\xi,\eta=0))$$
$$\text{for } \xi=\Delta\xi,2\Delta\xi,\ldots,1-\Delta\xi,$$

where the terms $\frac{\partial\tilde{v}}{\partial\xi}$ and $\frac{\partial\tilde{v}}{\partial\eta}$ are implemented via finite differences, a central difference for $\frac{\partial\tilde{v}}{\partial\xi}$ (as described in section 2.2), and a one-sided difference for $\frac{\partial\tilde{v}}{\partial\eta}$.

This modification requires the addition of nonzero elements in the system matrix $\tilde{A}$ as in the previous example (in general, we would also need to make additions to $\tilde{b}$, but this could be handled in the same way as the forcing function). By modifying $\tilde{A}$ in this way, the proper boundary conditions are realized when AD is performed with respect to α.

4. Numerical results. In this section, we present numerical results for the methodology presented above. AD is performed on both the 1-D nonlinear two-point boundary value problem (1) and the two-dimensional (2-D) Laplace equation (3) as well as the corresponding modified problems, (14) and (15). Since the modified problems avoid the computation of mesh sensitivities, comparing these results (in both the forward and reverse modes of AD) illustrate the gains that can be made with our modification. We use the following three metrics to compare the performance of these differentiated codes:

- FLOPs for the forward mode. In the modified problem, the required floating point operations (FLOPs) are expected to be less than in the original problem since the computation of mesh sensitivities is avoided. Hence, this measure indicates the amount of computation that can be saved using the modified problem. FLOP1 in Tables 1 and 2 represents the FLOPs for the modified problem and FLOP2 represents the FLOPs for the original problem. These calculations are made using the forward mode of AD. The problem size, N, (the solution is of size N in the 1-D problem and of size $N \times N$ in the 2-D problem) is varied to demonstrate how these savings scale.

- Space complexity for the reverse mode. This quantity measures the memory requirements for the reverse mode of AD. Recall that the reverse mode requires storing the function trace for the derivative computation. Hence, this measure indicates how much space is saved by avoiding the mesh sensitivity terms. In Tables 1 and 2, SPACE1 represents the space complexity (the number of floating point numbers that are stored) for the modified problem and SPACE2 is the space complexity for the original problem.

- FLOPs for the reverse mode. This measure indicates how much computation can be saved by avoiding the mesh sensitivities in the reverse mode. In Tables 1 and 2, RFLOP1 represents the FLOPs for the modified problem in the reverse mode, while RFLOP2 represents the corresponding FLOPs for the original problem.

4.1. Nonlinear two-point boundary value problem. We begin by discussing results for the nonlinear two-point boundary value problem introduced in sections 2.1 and 3.2. AD is applied to the original finite element code for (1) as well as to the modified code to approximate (14). The values of our three metrics are listed in Table 1 for various problem sizes. We observe significant gains in terms of both FLOPs (for both modes) and space requirements. All three performance metrics are more or less independent of problem size. This can be explained by the fact that the complexity of the linear system in the finite element code is linear in the problem size, N (every nonlinear equation solved converged in five iterations). The amount of time required to define the mesh and the finite element matrix is also linear in the problem size, hence, the FLOPs gain remains essentially constant. In addition, the amount of space allocated to define the mesh is proportional to N, causing the memory savings to be independent of N.

The FLOPs in the forward mode and the FLOPs and space complexity in the reverse mode are displayed in Figures 3, 5, and 4, respectively. Note that in each case, these metrics behave linearly in N.

The sensitivity solution. AD of the modified problem computes the sensitivity solution correctly. Figure 6 displays the ℓ_2-norm of the error between the sensitivity solution computed using the methodology of section 3 and the correct analytic sensitivity. The analytic sensitivity is obtained by differentiating u_{ex} and evaluating it at the nodes of the finite element mesh.

In the next section we produce the results for the 2-D Laplace equation problem, where the gains are more dramatic, especially in terms of memory requirements.

TABLE 1
1-*D problem comparisons.*

N	FLOP1	FLOP2	% gain	SPACE1	SPACE2	% gain	RFLOP1	RFLOP2	% gain
8	18924	25245	25.04	9698	10920	11.19	45800	52744	13.17
16	37352	49865	25.09	18890	20808	9.22	84984	97184	12.55
32	74624	99585	25.07	37274	40584	8.16	164008	186720	12.16
64	149168	199025	25.05	74042	80136	7.60	322056	365792	11.96
128	298256	397905	25.04	147578	159240	7.32	638152	723936	11.85

TABLE 2
2-*D problem comparisons.*

N	FLOP1	FLOP2	% gain	SPACE1	SPACE2	% gain	RFLOP1	RFLOP2	% gain
10×10	4.06×10^4	5.44×10^4	36.9	2.49×10^3	3.09×10^4	91.9	7.19×10^4	1.18×10^5	39.0
20×20	4.15×10^5	5.30×10^5	21.7	1.00×10^4	1.45×10^5	93.1	6.65×10^5	8.85×10^5	24.9
40×40	6.01×10^6	6.52×10^6	7.7	4.01×10^5	6.29×10^6	93.6	8.93×10^6	9.89×10^6	9.7
80×80	8.97×10^7	9.18×10^7	2.3	1.60×10^6	2.61×10^7	93.9	1.32×10^8	1.37×10^8	3.6
100×100	2.18×10^8	2.21×10^8	1.4	2.50×10^6	3.63×10^7	93.8	3.22×10^8	3.28×10^8	1.8

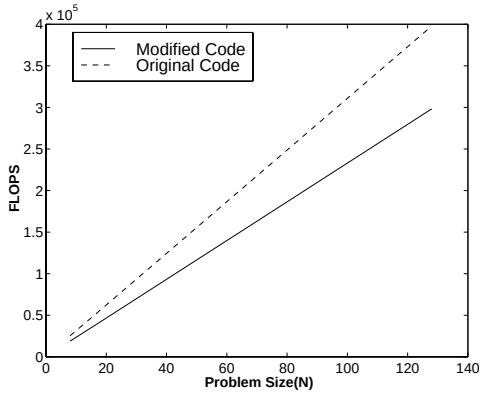

FIG. 3. *Forward mode FLOPs for the* 1*-D problem.*

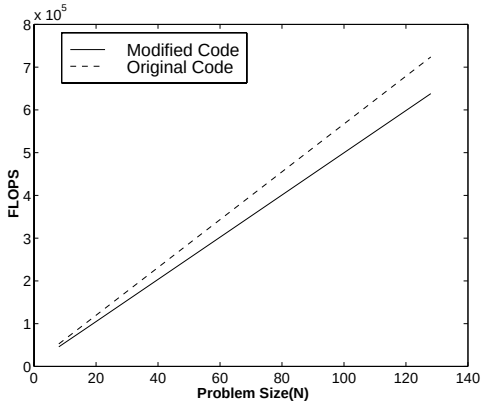

FIG. 4. *Reverse mode FLOPs for the* 1*-D problem.*

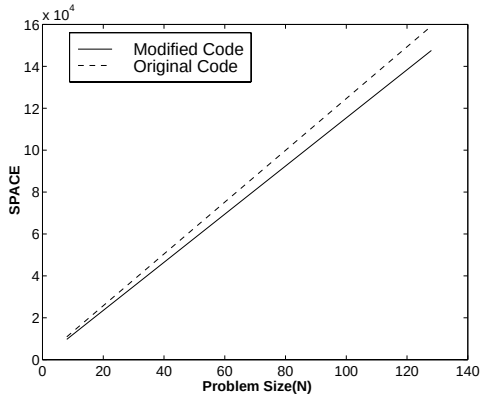

FIG. 5. *Space complexity for the* 1*-D problem.*

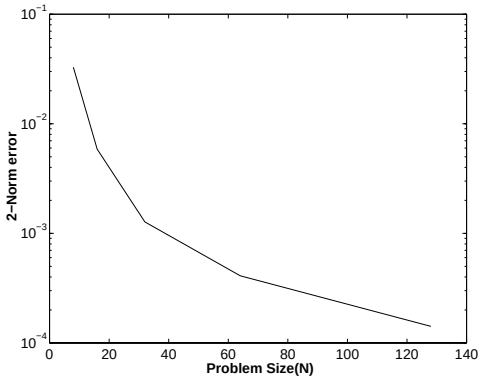

FIG. 6. *The ℓ_2-norm of the error in the sensitivity solution.*

TABLE 3
FLOPs overhead in solving the CSE.

N	Approximating the CSE	Proposed AD methodology	% overhead
10×10	3.44×10^4	4.06×10^4	18.0
20×20	4.01×10^5	4.15×10^5	3.5
40×40	5.86×10^6	6.01×10^6	2.6
80×80	8.79×10^7	8.97×10^7	2.1
100×100	2.14×10^8	2.18×10^8	1.8

4.2. Laplace equation in two dimensions. Table 2 provides computational metrics for AD of the 2-D problem introduced in sections 2.2 and 3.2. Results are provided for various problem sizes. In this case, the trend is that the improvement in the FLOPs requirement diminishes with increasing problem size. This is explained by the fact that the most expensive step in the computation is the solution of the linear system. This step dominates defining the mesh, the system matrix, etc. Therefore, the difference in the FLOPs required to compute the different right-hand sides becomes negligible.

The FLOPs requirement in the reverse mode depends on both the space complexity metric (since all stored derivatives need to be accessed again in the reverse pass) as well as the FLOPs to perform the derivative calculations themselves. Hence, in the reverse mode, the FLOPs gain is greater than that in the forward mode. However, the dependence of the space complexity metric on the FLOPs count is insignificant compared to the FLOPs required to perform the linear system solve. Figures 7 and 9 display how the FLOPs in the forward mode and reverse mode compare for the two AD methodologies. The substantial order-of-magnitude gain in space complexity can be seen in Figure 8.

The efficiency of the code generated by our AD methodology compares favorably with software written specifically to solve the CSE. We display the FLOPs requirements for solving the CSE using AD of the modified problem as well as traditional application of AD in Table 3. Note that there is very little overhead involved with this AD approach over a direct approximation. These results correspond to the same approximation of the CSE that is used for the PDE.

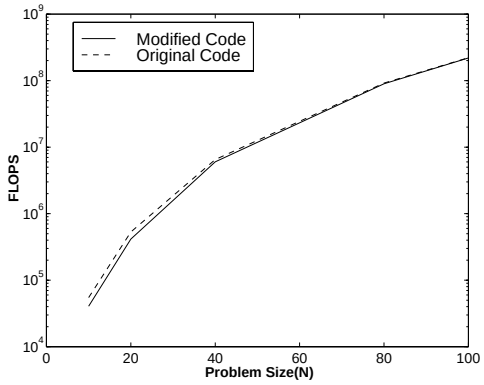

FIG. 7. *Forward mode FLOPs for the* 2*-D problem.*

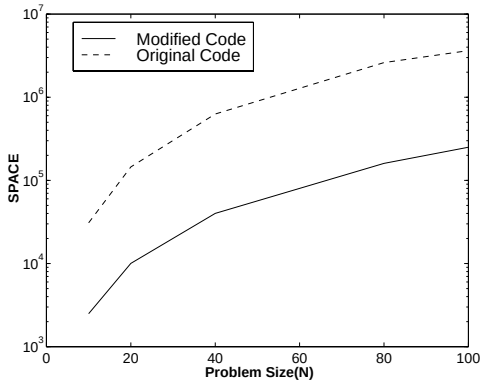

FIG. 8. *Space complexity for the* 2*-D problem.*

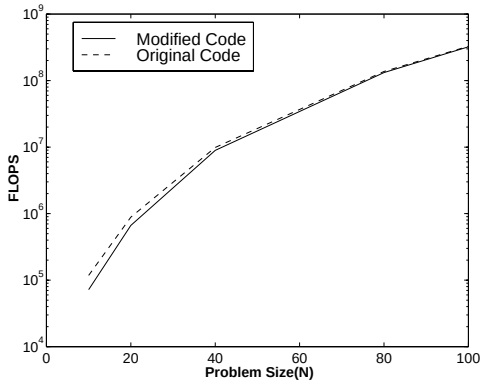

FIG. 9. *Reverse mode FLOPs for* 2*-D problem.*

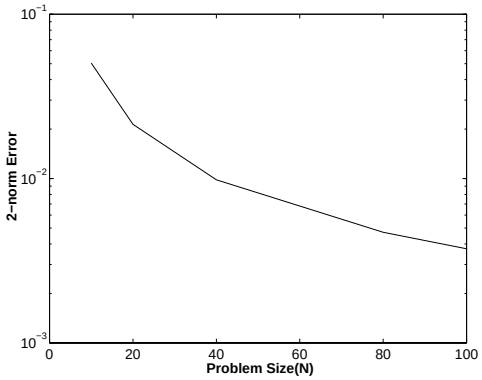

FIG. 10. *The ℓ_2-norm of the error in the sensitivity solution.*

The sensitivity solution. The ℓ_2-norm of the error in sensitivity calculations versus problem size is plotted in Figure 10. As in section 4.1, the exact solution is defined by evaluating the analytic sensitivity at the $N \times N$ finite difference nodes. Again, we see that accurate sensitivity solutions are calculated using AD of the modified problem.

4.3. Analysis of the numerical results. These results show that a sometimes dramatic, and usually substantial, gain in terms of FLOPs and memory requirements can be achieved by using the modified problem. To develop an a priori estimate of the potential gains of this AD methodology, we present the following recipe. Consider the following PDE solution code which can be broadly divided into two main parts: the code to define the mesh and the code for computing the solution. Consider a general template (here a denotes the independent variables, e.g., the shape parameters):

function $u =$ **pdesolution**(a)

Compute the Mesh, $M(a)$. (Complexity: memory $= S_m$, FLOPs $= F_m$)

Compute the Solution, $u = F(a, M(a))$. (Complexity: memory $= S_s$, FLOPs $= F_s$)

end

For generality, we assume a multiparameter case. Assume the number of parameters is given by p. In the above framework, the various complexities can now be estimated by

- FLOPs for the forward mode. For AD of the modified problem, the FLOPs will be $\mathsf{FLOP1} = F_m + (p+1)F_s$, while for the original problem it will be $\mathsf{FLOP2} = (p+1)(F_m + F_s)$. The gain is
$$\frac{\mathsf{FLOP2} - \mathsf{FLOP1}}{\mathsf{FLOP2}} \times 100\% = \frac{pF_m}{\mathsf{FLOP2}} \times 100\%.$$
If F_m is substantial, i.e., the mesh definition requires a lot of computation as in elliptic grid generation [20], then a significant gain can be expected. *In the 1-D problem, F_m is a significant portion of the computational effort as N is increased; thus the gain remains significant as the problem size increases.*
- Space complexity for the reverse mode. For the modified problem the space complexity will be $\mathsf{SPACE1} = S_m + (p+1)S_s$ while for the original problem

it will be $\mathsf{SPACE2} = (p+1)(S_m + S_s)$. The gain is

$$\frac{\mathsf{SPACE2} - \mathsf{SPACE1}}{\mathsf{SPACE2}} \times 100\% = \frac{pS_m}{\mathsf{SPACE2}} \times 100\%.$$

If the space requirement for the mesh setup, S_m, is substantial compared to the solution step, S_s, then a significant gain is expected in the space complexity. *In the 2-D problem, the memory required to store the mesh and its derivatives, S_m, remains a significant portion of the overall total memory required (since the system matrix is sparse); thus the gain remains substantial as the problem size is increased.*

- FLOPs for the reverse mode. For the modified problem, the FLOPs required for the reverse mode of AD will be $\mathsf{RFLOP1} = \mathsf{FLOP1} + \mu\mathsf{SPACE1}$ while for the original problem it will be $\mathsf{RFLOP2} = \mathsf{FLOP2} + \mu\mathsf{SPACE2}$, where μ is constant which determines the FLOPs required to save and access one unit of memory (stored for accessing in the reverse pass). The gain is

$$\frac{\mathsf{RFLOP2} - \mathsf{RFLOP1}}{\mathsf{RFLOP2}} \times 100\% = \frac{(1+\mu)\, pF_m}{\mathsf{rFLOP2}} \times 100\%.$$

5. Comments and conclusions. To summarize, recall that when a problem has a parameter-dependent domain, differentiating the discrete form of the equations (the traditional use of AD) can lead to mesh sensitivities. These terms lead to additional computations and storage in the differentiated code. This extra computation can be avoided by using AD to generate code to approximate the CSE. Minor modifications to the original software were outlined for two example problems so that AD of the modified code produced the desired result. Numerical experiments indicated that the number of FLOPs and the required memory were both reduced when this methodology was applied. In some cases, the savings were substantial (up to 25% improvement in the FLOPs requirements for a 1-D finite element problem and better than 10-fold improvement in the memory requirements for a 2-D finite difference problem). These are savings over and above the calculation of the mesh sensitivity terms themselves. In the previous section, we provided formulae that can be used to estimate the amount of savings that this AD methodology will produce.

The methodology is useful in problems where the domain is fixed, yet $\mathcal{M}_a$ is still parameter dependent (e.g., this occurs when adaptive mesh refinement strategies are used). To handle this case with AD, the mesh is simply specified as a fixed constant (it is possible to do this in most AD software). The CSE are solved on this mesh using the technique described in section 3. Since the approximation takes place after the differentiation, sensitivities are found without mesh derivatives [5]. Note that fixing the mesh is also possible in the approximate-then-differentiate approach if the domain is not parameter dependent.

This methodology has no benefit in problems where $\mathcal{M}_a = \mathcal{M}$, however, as the operations of differentiation and approximation commute in this case. For this situation, it is better to apply AD to the original code (with respect to a) than to apply the proposed AD methodology (with respect to α).

In some instances, the modification of the boundary conditions can be very difficult. Programs which are set up to handle zero Dirichlet boundary conditions may not have the capability of treating nontrivial data without some significant program modification. These cases require extending the solution vector to include the boundary points and increasing the size of the system matrix. These changes may not be

straightforward when this occurs and the benefits reported in section 4 need to be weighed against the resources required to perform the necessary changes. It's possible that the traditional application of AD is more attractive in this case.

Our methodology was presented for the situation where either the boundary conditions, forcing functions, or the shape of the domain was parameter dependent. However, it can be extended to a more general case when, for example, $\mathcal{A}_a$ has other dependencies on a, such as parameter-dependent coefficients. When this situation occurs, the CSE contains additional functions of u. These extra terms can be combined with the function $g(u)$ in problem (4). The implementation of this methodology can be more difficult in this case since we need to extract the necessary code out of the computation of $A_a(u^N)$ in order to build the modified right-hand side.

This approach is applicable to a wide variety of PDE approximation schemes. In this work, we have demonstrated the applicability of this approach for PDEs that are approximated with either finite differences or finite element methods. However, one should take the point of view that AD is being used to approximate the sensitivity equation in these problems. Therefore the success of this approach comes from the fact that the approximation scheme can be changed so that it approximates a modified PDE (and that these modifications are small, occurring in a small portion of the code). The application of AD to this modified code will yield the desired sensitivity solution.

REFERENCES

[1] C. Bischof, A. Carle, A. Griewank, and P. Hovland, *ADIFOR: Generating Derivative Codes From Fortran Programs*, Technical Report MCS-P263-0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, and Center for Research on Parallel Computation, Rice University, Houston, TX, 1991.

[2] J. Borggaard and J. Burns, *Asymptotically consistent gradients in optimal design*, in Multidisciplinary Design Optimization: State of the Art, N. Alexandrov and M. Hussaini, eds., SIAM, Philadelphia, PA, 1997, pp. 303–314.

[3] J. Borggaard and J. Burns, *A PDE sensitivity equation method for optimal aerodynamic design*, J. Comput. Phys., 136 (1997), pp. 366–384.

[4] J. Borggaard, J. Burns, E. Cliff, and M. Gunzburger, *Sensitivity calculations for a 2D, inviscid, supersonic forebody problem*, in Identification and Control in Systems Governed by Partial Differential Equations, H.T. Banks, R. Fabiano, and K. Ito, eds., SIAM, Philadelphia, PA, 1993, pp. 14–24.

[5] J. Borggaard and D. Pelletier, *Computing design sensitivities using an adaptive finite element method*, in Proceedings of the 27th AIAA Fluid Dynamics Conference, New Orleans, LA, 1996.

[6] G. Bugeda and J. Oliver, *A general methodology for structural shape optimization problems using automatic adaptive remeshing*, Internat. J. Numer. Methods Engrg., 36 (1993), pp. 3161–3185.

[7] J. Burns, L. Stanley, and D. Stewart, *Computational methods for design sensitivities*, in Optimal Control: Theory, Algorithms Appl., Appl. Optim. 15, W.H. Hager and P.M. Pardalos, eds., Kluwer, Dordrecht, 1998, pp. 40–66.

[8] G.C. Buscaglia, R.A. Feijóo, and C. Padra, *A posteriori error estimation in sensitivity analysis*, Structural Optimization, 9 (1995), pp. 194–199.

[9] T. Coleman and A. Verma, *Structure and efficient Hessian calculation*, in Advances in Nonlinear Programming, Ya-Xiang Yuan, ed., Kluwer, Dordrecht, The Netherlands, 1996, pp. 57–72.

[10] T. Coleman and A. Verma, *Structure and efficient Jacobian calculation*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bishof, G. Corliss, and A. Griewank, eds., SIAM, Philadelphia, PA, 1996, pp. 149–159.

[11] T.F. Coleman, F. Santosa, and A. Verma, *Semi-automatic differentiation*, in Computational Methods for Optimal Design and Control, J. Borggaard, J. Burns, E. Cliff, and S. Schreck, eds., Birkhäuser, Boston, 1998, pp. 113–126.

[12] A. Verma, *ADMAT: Automatic differentiation in MATLAB using object oriented methods*, in Proceedings of the 1998 SIAM Workshop on Object Oriented Methods for Interoperable Scientific and Engineering Computing, M.E. Henderson, C.R. Anderson, and S.L. Lyons, eds., SIAM, Philadelphia, 1999, pp. 174–183.

[13] T.F. Coleman and A. Verma, *ADMIT*-1: *Automatic differentiation and MATLAB interface toolbox*, ACM Trans. Math. Software, to appear.

[14] P. Eberhard and C. Bischof, *Automatic Differentiation of Numerical Integration Algorithms*, Technical Report MCS-P621-1196, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1996.

[15] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optim. Methods Softw., 1 (1992), pp. 35–54.

[16] A. Griewank, *Some bounds on the complexity of gradients, Jacobians, and Hessians*, in Complexity in Nonlinear Optimization, P. Pardalos, ed., World Scientific Publishers, River Edge, NJ, 1993, pp. 131–167.

[17] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, 2 (1996), pp. 131–167.

[18] J.M. Ortega and W.C. Rheinboldt, *On discretization and differentiation of operators with application to Newton's method*, SIAM J. Numer. Anal., 3 (1966), pp. 143–156.

[19] D. Stewart, *Numerical Methods for Accurate Computation of Design Sensitivities*, Ph.D. thesis, Virginia Tech, Blacksburg, VA, 1998.

[20] J.F Thompson, Z.U.A. Warsi, and C.W. Mastin, *Numerical Grid Generation: Foundations and Applications*, North-Holland Publishing Company, New York, 1985.

[21] A. Verma, *Structured Automatic Differentiation*, Ph.D. thesis, Cornell University, Ithaca, NY, 1998.