Automatic Differentiation Applied to Economics

Emmanuel M. Tadjouddine [*]

Abstract—**This paper discusses the use of the Automatic Differentiation approach in evaluating derivatives of functions represented by computer programs. We then considered a Cournot oligopoly modeled by a system of stochastic differential equations. The setting is that of a set of self-interested firms striving to adjust their productions in the direction of higher profits subject to mistakes or random shocks. The stochastic differential equations are solved by a numerical method and the profits are calculated using a Monte Carlo simulation. Then, Automatic Differentiation is used to propagate sensitivities along each path in an automated fashion. Numerical results have confirmed the intuition one may have that noisy environments can lead to important profit differences between firms as well as higher sensitivities as opposed to less noisy ones.**

Keywords: Automatic Differentiation, sensitivity analysis, stochastic differential equations, Cournot oligopoly

1 Introduction

Solving nonlinear systems or nonlinear optimization often requires derivative computation. Derivatives can be obtained by hand when they are easy. However, if the functions are too complicated, we may use the popular finite differencing scheme, the complex-step derivative approximation [11] or the truncation-error-free derivative evaluation known as Automatic Differentiation (AD) [7, 14]. AD is a technique allowing for the evaluation of derivatives of a numerical function represented by a computer program. In here, we will use AD to evaluate sensitivities of expected profit functions by firms engaged in a competition wherein each firm is self-interested and is trying to setup its production so as to maximize its expected profit.

The economic model we have considered is known as Cournot oligopoly. It represents a process that evolves over time in a noisy environment giving rise to a *stochastic game*. This describes a sequential game (played in rounds) going from one state to another thanks to some probability transitions. At each round, firms simultaneously and independently select an action out of a finite set of actions according to their individual objective and the history of the game's interactions so far. Then, each firm receives an utility value based on an profit function mapping the outcomes to real numbers.

As discussed in [1, 6], these interactions can be modeled by a system of Stochastic Differential Equations (SDEs). To solve these equations, we have used the Euler-Maruyama scheme, see for example [9, 10]. Then, we have evaluated the profits of the firms by a Monte Carlo simulation.

Because of the impact small variations on actions can have on firms' profits, it is important to be able to assess the sensitivity of each firm's profit function with respect to its own action and the actions of its opponents. These sensitivities are useful as they may guide firms in their decision making process but also they can help the oligopoly designer through a feedback loop in the model. For example, if profit differences are small across a range of possible actions, then firms can make errors since the consequences for their expected profit are minor.

The remainder of this paper is organized as follows. Section 2 presents an overview of AD, points out its advantages and disadvantages, and states some of its challenges. Section 3 presents a Cournot oligopoly model to which we have used AD to evaluate numerical sensitivities. Eventually Section 4 presents some concluding remarks.

2 Automatic Differentiation (AD)

Automatic Differentiation of a computer code representing a function $F : R^n \mapsto R^m$ can be viewed as a program transformation in which the original code's statements that calculate real valued variables are augmented with additional statements to calculate their derivatives. This is carried out by regarding the original computer code as a sequence of elementary functions having at least a first derivative and then using the chain rule to automatically evaluate the function represented by the given code as well as its derivative. The main advantage of this technique is that it can handle codes of arbitrary complexity to provides accurate and fast derivatives while being reliable compared to hand-coding.

Assuming $\mathrm{d}x$ is the derivative associated with a variable x and x_1, x_2 the variables with respect to which we de-

[*]Department of Computer Science & Software Engineering, Xi'an Jiaotong-Liverpool University, 111 Ren Ai Road, SIP, Suzhou, Jiangsu Province, P.R. China 215123 Email: emmanuel.tadjouddine@xjtlu.edu.cn

sire derivatives, the function $F : R^2 \mapsto R : (x_1, x_2) \mapsto x_1 x_2/(1 + x_1^2)$ represented by the following code can be transformed as follows:

$$\begin{array}{ll} v_1 & = x_1 x_2 \\ v_2 & = 1 + x_1^2 \\ y & = v_1/v_2 \end{array} \quad \Rightarrow \quad \begin{array}{ll} \mathrm{d}v_1 & = x_2\,\mathrm{d}x_1 + x_1\,\mathrm{d}x_2 \\ v_1 & = x_1 x_2 \\ \mathrm{d}v_2 & = 2x_1\,\mathrm{d}x_1 \\ v_2 & = 1 + x_1^2 \\ \mathrm{d}y & = (v_2\,\mathrm{d}v_1 - v_1\,\mathrm{d}v_2)/v_2^2 \\ y & = v_1/v_2 \end{array} \tag{1}$$

In AD terminology, we define the *independent* variables to be those input variables with respect to which we need to compute the derivatives, the *dependent* variables to be those outputs whose derivatives are desired, and the *intermediate* variables to be those whose value depends on an independent and affects a dependent variable. An *active* variable is an independent, intermediate, or dependent variable.

We distinguish at least two standard AD algorithms: the forward mode and the reverse or adjoint mode. The forward mode propagates directional derivatives along the control flow of the original program. The cost of evaluating $\nabla\mathbf{F}$ (see [4, 7] for more details) is bounded above as follows:

$$W(\nabla\mathbf{F}) \leq 3nW(\mathbf{F}) \tag{2}$$

The adjoint mode is composed of two passes: a forward pass that computes the function and a reverse pass that calculates the sensitivities of the dependent variables with respect to the intermediate and independent variables in the reverse order to their calculation in the function. In the reverse pass, we may need to recompute intermediate values required by the differentiation process or extract them from a storage data structure called the tape (essentially a LIFO stack) if they have been stored during the forward pass. The sensitivities of the dependent to the independent variables give the desired derivatives. The cost of calculating $\nabla\mathbf{F}$ (see [4, 7] for more details) is bounded above as follows:

$$W(\nabla\mathbf{F}) \leq 3mW(\mathbf{F}) \tag{3}$$

Note that the adjoint mode is particularly efficient in calculating gradients (m=1) since its complexity is not dependent on the number of inputs. However, the storage requirement may increase dramatically and powerful strategies to reduce the tape size are required in order to approach the complexity bound shown in (3).

The forward and adjoint modes of AD are implemented in various AD tools that usually use the following approaches:

- Augmenting the given computer code with extra statements calculating derivatives and output a transformed computer code (e.g., ADIFOR, TAF, TAPENADE).
- Providing a library that overloads the elementary operations to support derivatives calculation (e.g., ADOLC, MAD, ADIMAT). See `www.autodiff.org` for references and details on those softwares.

In this paper we are concerned with AD software for numerical codes written in the FORTRAN programming language. Examples of such AD tools are ADIFOR, TAF, and TAPENADEthat implement the two AD modes with variant strategies for performance purposes.

To differentiate a given source code using an AD package, we usually specify the independents, dependents, and the top-level routine, and then choose an AD algorithm (e.g., forward mode AD). However, the process is frequently not that straightforward for large-scale applications since current AD tools are limited by their language coverage. This limitation often forces the AD user to rewrite his or her input code before being transformed by the chosen software. This phase of code preparation may be aided by scripting languages (e.g., SED, PERL, or PYTHON) to automate the rewrite process. Examples of FORTRAN features not currently well handled by current AD tools include structured or derived data types, modules, dynamic allocation, pointers, or control structures such as `cycle` or `exit`. Consequently, an input code that uses such features may need to be rewritten prior to differentiation. In this preparation process, the AD user must validate each transformation by checking the semantic of the input code is preserved. This may become a difficult task when the input program is a legacy code that has been validated or developed in a third party location. Eventually, when the input code can be read and analyzed by the AD tool, a transformed code that computes the original function and its derivatives will be generated. The obtained computer program need be compiled and run to get derivative values. This leads us to seek ways of checking the correctness of derivative values hence validating the AD generated code and of improving its performance.

2.1 Validation

To ensure an AD generated code calculates the correct derivative values, it is important to check that the AD obtained values are in line with those from Finite-Fifferencing (FD) and that different AD algorithms give the same values to within the limits of the machine precision. We may proceed as follows:

1. Compute a single directional derivative $\dot{\mathbf{y}} = \nabla F(\mathbf{x})\dot{\mathbf{x}}$ for a random direction $\dot{\mathbf{x}}$ by using FD and the forward mode AD. Then check that the difference between the two values is about the square root of the machine precision ϵ.
2. Compute a single adjoint $\bar{\mathbf{x}} = \nabla F(\mathbf{x})^T\bar{\mathbf{y}}$ for a random $\bar{\mathbf{y}}$ via the reverse mode AD. Then check that

$$\bar{\mathbf{y}}\dot{\mathbf{y}} \equiv \bar{\mathbf{x}}\dot{\mathbf{x}}.$$

This validation process is crucial and needs careful attention. Note that AD makes the assumption that the input function $\mathbf{F}$ to be differentiated is composed of elemental functions ϕ that are continuously differentiable on their open domains [7]. However, this may not be the case for real-life applications. At a point on the boundary of an open domain, the function $\mathbf{F}$ can be continuous, but its derivative $\nabla\mathbf{F}$ may jump to a finite value or even infinity. A black box approach of AD can lead to wrong results in the presence of non-differentiable functions or iterative processes.

2.1.1 About non-differentiability

A real-life application may contain mathematical functions that are not differentiable in some points in their domain. A computer code that models such an application may contain intrinsic functions (e.g. `abs`, or `arccos`) or branching constructs used to treat physical constraints for instance non physical values of model parameters. We now describe three situations, which may cause non-differentiability problems.

First, let us consider the case related to non-differentiable intrinsic functions. For instance, the derivative of cos^{-1} is not defined at $x = 0$ since

$$\frac{\mathrm{d}\cos^{-1}(x=1)}{\mathrm{d}x} = \infty.$$

Moreover, consider the function `abs`. Its derivative evaluated at the point $x = 0$ has more than one possible values including $-1, 0, 1$. Choosing one of these values depends upon the numerical application. This suggests that there is no "automatic" way of treating such a pathological case and that code insight is crucial in guiding sensible choices. To date, the best thing an AD tool can do is to provide an exception handling mechanism that can be turned on in order to track down intrinsic related non-differentiable points. ADIFOR is a primary example for such a mechanism and to our knowledge, at the time of the writing, it is unique in that respect.

Second, let us consider pathological cases related to branching constructs. Differentiating blindly such a construct may give point-valued derivatives. Therefore a function $\mathbf{F}$ that is mathematically differentiable may become non-differentiable when AD is applied. This happens namely when the test used in the branching construct involves active variables as in the following example:

$$\text{if } x == 0 \text{ then } \ y = 1.0 \ \text{ else } \ y = x + 1.0 \qquad (4)$$

An AD tool will give a derivative value zero at x=0 in lieu of the value one. Although in this example, the branching construct is not needed, there may be cases where such constructs may be used to prevent unphysical values. Currently AD tools do not handle or even detect these cases. Tracking down such pathological cases may require developing robust program analysis algorithms or rely on the user's insight of the given computer code.

Third, consider an engineering application in which the independent or dependent variables are real-valued but complex-valued data have been used for computation purposes. Using the equivalence between R^2 and C, a complex function $h : a + ib \mapsto f(a, b) + ig(a, b)$ of a complex variable $a + ib$, where a, b are real values and f, g are real-valued functions, is differentiable if and only if h is *analytic* meaning $\frac{\partial f}{\partial a}=\frac{\partial g}{\partial b}$ and $\frac{\partial f}{\partial b}= -\frac{\partial g}{\partial a}$. It follows that the conjugate operator $z \mapsto \bar{z}$ is not differentiable. The application of AD into such complex-valued functions is discussed in [13]. Unlike real-valued functions, complex-valued functions may be many-to-one mappings. For example, the function sqrt maps a complex number x=$a + ib$ to two complex numbers z and $-z$ with z=$\sqrt{1/2(a + \sqrt{a^2 + b^2})} + i\sqrt{1/2(-a + \sqrt{a^2 + b^2})}$. This may raise subtle issues for the application of AD. It also turns out that the modulus function `abs` raises issues when evaluated at the origin. To handle possible singularities, the application of the chain rule must be robustly used by the AD tools [13].

2.1.2 Iterative Numerical Solvers

An important question in using AD concerns differentiating through iterative processes. Typically, AD augments the given iteration with statements calculating derivatives. Empirically, AD provides the desired derivatives. However, questions remained as to whether the AD generated iteration converges and what it converges to. Consider Fischer's example as discussed in [3]. The iterative constructor $x_{k+1} = g_k(x_k)$ with

$$g_k(x) = x \exp(-kx^2) \qquad (5)$$

converges to $g \equiv 0$ when $k \to \infty$ whilst its derivative $g_k^{'}(x) \to 0$ but $g^{'}(0) = 1$. The issues of derivative convergence for iterative solvers in relation to AD are discussed in detail in [5, 8] for the forward mode AD and in [2] for the adjoint mode. In [8], it is been shown that the mechanical application of AD to a fixpoint iteration gives a derivative fixpoint iteration that converges R-linearly to the desired derivative for a large class of nicely contractive iterates or secant updating methods.

Usually, current AD tools generate derivative code using the same number of iterations as the original solver. However, if the initial guess is close to the solution, then this adjoint solver does no longer converge to the adjoint of the solution. For example, let us consider the following implicit iterative solver:

$$z_0 = z_0(x, y), \quad z_i = g(x, y, z_{i-1}) \ \text{ for } \ i = 1 \ldots l, \qquad (6)$$

for l a non negative integer and the function g defined as:

$$\begin{aligned} g:\quad & R^3 \to R \\ & (x,y,z) \mapsto (y^2+z^2)/x \end{aligned}$$

$z_0 = z_0(x,y)$ is meant z_0 is initialized for some values of x and y. For given values $x = 3, y = 2$ and an initial guess $z = 0.5$, the implicit equation

$$z = g(x,y,z)$$

has a solution $z_*{=}z_*(x,y){=}1$ and $\nabla g(x,y,z_*){=}(-1,1)$. When the code in equation 6 is mechanically differentiated using for example Tapenade, we observed:

- if the initial guess is within a radius of the solution that leads to convergence, then the AD generated iteration converged to the correct derivative.
- if the initial guess is closer to the solution, say the initial value of $z{=}1$, then the derivative iteration converges in one iteration to $\nabla g(x,y,z_*){=}(-1/3,1/3)$, which is wrong.

This means the assumption made by most AD tools to use the same number of iterations taken by the original iterative process for the derivative one is fair but may lead to wrong derivatives in certain cases. As suggested in [2], the AD tool ought to augment the convergence criterion to account for derivative convergence.

In summary, validating derivative calculation via AD can be difficult in the presence of non-differentiable functions and iterative solvers. It is hoped future AD tools will help spotting such anomalies and raising warnings to the AD user since, to our knowledge, there is no automatic ways of solving these issues.

2.2 About Efficiency

In theory, the fundamental idea of AD is simple: consider a computer code as a composition of elementary functions and differentiate it using the chain rule. The main advantage of this technique is that it provides algorithms that exhibit better accuracy and performance compared to finite differencing and that it is more reliable than hand-coded derivatives, which are error prone. In practice, there is a scope for AD tools to improve further in producing fast and reliable derivatives. Most of these improvements concern code optimization. New algorithms have been developed and implemented into some AD software. These include the followings:

- Dependency analyses which determine the set of active variables and procedures. This eliminates many redundant calculations e.g., creating derivative objects that are a priori zero or adding/multiplying zero.
- In-out analyses which determine sets of active variables or required variables for the reverse sweep of the adjoint mode at subroutine level.
- Providing 'directives' facilities for the user to exploit certain insights of the code to be differentiated (e.g., parallel loops).
- Use of graph elimination based techniques for local Jacobians, see for example [12, 15, 4].

However, hand-tuning the AD generated code can give further performance.

3 Cournot Oligopoly

Let us consider a Cournot *oligopoly* (a competition between a small number of sellers) with n firms competing for the production of a single good. The cost to firm i of producing x_i units of the good is an increasing function $C_i(x_i)$ while the firms' total output is sold at a single price, which is a decreasing function so that firm i's revenue is $x_i(a - b\sum_{j=1}^n x_j)$, wherein a and b are nonnegative real numbers. Thus, firm i's profit is

$$\pi_i(\mathbf{x}) = x_i(a - b\sum_{j=1}^{n} x_j) - C_i(x_i) \tag{7}$$

This is usually modeled as a game wherein players are the set of firms and the actions for a player i are the set of nonnegative numbers x_i. Over time each firm i will take actions x_i so as to maximize $\pi_i(\mathbf{x})$. This gives rise to a dynamical system wherein firms produce certain amounts of the good to maximize their profits but also, observed profits lead to adjustments for the production quantity. However, these decisions on how much to produce are taken under uncertainties. These uncertainties may cause errors or random shocks in each firm's production. Random shocks may arise from mistakes in judgment, preference discrepancies or emotions. Moreover, a small variation of a firm's production could impact its profit and those of its opponents.

Let us now assume the oligopoly is composed of $n \geq 2$ firms and let us denote $x_i(t) \in [\underline{x}, \bar{x}]$ the action of firm i over time. Let $u_i(\mathbf{x}(t), t)$ be the expected profit for firm i when firms take action $\mathbf{x}$. Assuming that the profit function u_i is differentiable and denoting $g_i(\mathbf{x}(t)) = u_i^{'}(\mathbf{x}(t), t) = \frac{\partial u_i(\mathbf{x})}{\partial x_i}$, actions $x_i(t)$ for an firm i evolve according to the following system of independent SDE (Stochastic Differential Equations)

$$\mathrm{d}x_i(t) = g_i(\mathbf{x}(t), t)\,\mathrm{d}t + \sigma_i\,\mathrm{d}w_i(t), \quad i = 1, \ldots, n \tag{8}$$

wherein the differential $\mathrm{d}w_i(t)$ of the standard Wiener process $w_i(t)$ represents the random shocks and σ_i a parameter related to each firm, see [1, 6] for more details. Equation (8) expresses the idea that a player i's decision will increase as its expected profit increases at $\mathbf{x}(t)$

proportionally to the slope (derivative) of the profit function plus a stochastic Brownian motion with a parameter σ_i representing the importance of random shocks in the system.

If the production cost function C_i admits a continuous first derivative, then the profit function π_i for firm i is differentiable and its derivative with respect to the output of firm i is

$$\frac{\partial \pi_i(\mathbf{x})}{\partial x_i} = a - b\sum_{j=1}^{n} x_j - bx_i - C_i'(x_i) \qquad (9)$$

Actions x_i taken by the firms in the competition are independent and their evolution can be described by the system of independent SDEs in equation (8) since each firm is aiming to maximize its profit. We will use this example in the solution of equation (8) using the Monte Carlo method.

3.1 Monte-Carlo Simulation

We have used the Euler-Maruyama scheme, see for example [9, 10] to solve the SDEs in equation (8). Then, we have estimated each firm's profit by using the Monte-Carlo method to evaluate an average profit over P Brownian paths. For each path, we consider only the decision $x_i^{(j)}(T)$ taken by firm i along the Brownian path j at time T. The associated profit over P paths can be estimated as

$$u_i(x_i(T)) = \frac{1}{P}\sum_{j=1}^{P}\frac{1}{T}g_i(\mathbf{x}^{(j)}(T)) \qquad (10)$$

Notice that although the equations describing the dynamics of the decision making for each player are independent, the profit of each player relies on solving the n SDEs.

For a given firm i in the oligopoly, we would like to estimate

$$\frac{\partial u_i(\mathbf{x}(T))}{\partial x_j(0)},$$

the sensitivity of firm i's profit with respect to the initial decision $x_j(0)$ of a participant j. Such a sensitivity information may help firm i in its decision making process but also it may assist an oligopoly designer in inventing protocols that have some desirable properties. For example, if initial actions have small effects on an firm's profit, then firms can make errors since the consequences for their expected profit are minor.

If all components $x_k(t), k = 1, \ldots, n$ of the vector $\mathbf{x}$ are differentiable in $x_j(0)$ and if u_i is differentiable in $x_i(t)$, then we can write

$$\frac{\partial u_i(\mathbf{x}(T))}{\partial x_j(0)} = \sum_{i=1}^{n}\frac{\partial u_i(\mathbf{x}(T))}{\partial x_i(T)}\frac{\partial x_i(T)}{\partial x_j(0)} \qquad (11)$$

Notice that this gives us pathwise derivatives, not only we get sensitivities with respect to an initial decision $x_j(0)$ but we can even compute sensitivities with respect to $x_j(mh)$, the action of player j at the discrete time mh in the evolution of the stochastic game.

3.2 Numerical Results

We have solved the equations (8) wherein $g(\mathbf{x})$ represents the derivative of the firm i's profit π_i in equation (9) and $C_i(x_i)$, a monotonically increasing function on $[0+\infty[$,

$$\begin{aligned} g(\mathbf{x}) &= \tfrac{\partial \pi_i(\mathbf{x})}{\partial x_i}, \\ C_i(x_i) &= c_i x_i^2, \end{aligned}$$

whith $c_i > 0$. The EM solutions for two firms, say 1 and 2, in an olygopoly composed of 30 firms are depicted in Figure 1. The parameter σ_i representing the diffusion term for the SDEs in equation (8) was set so that $\sigma_1 > \sigma_2$. We have observed a higher variability for the decisions of firm 1 compared to those of firm 2. It is as if the parameter σ_i in equation (8) represents some kind of variance of the variable x_i, which encodes obviously the actions by firm i.

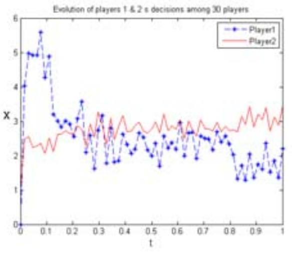

Figure 1: Simulation of the evolution of decisions for firms 1 & 2 in a oligopoly of 30 firms; in this simulation, we have chosen $\sigma_1 > \sigma_2$.

We have also computed the profit $\pi_i(\mathbf{x})$ for each firm i by using the Monte-Carlo estimation in equation (10) and then calculated the Jacobian $\nabla\pi(\mathbf{x})$ by using both FD (Finite-Differencing) and the AD technique. The AD sensitivities have been obtained using the MAD tool, which implements the forward mode AD to calculate the derivative of a function written in MATLAB. Because $\nabla\pi(\mathbf{x})$ is a square matrix and given the complexities of the forward and reverse modes AD, it is natural to use here the forward mode and MAD is a good choice.

Furthermore, by varying the parameter σ_i in equation (8), we observed that the more noise in the system, the more sensitive the profits become and that in a less noisy environment, these sensitivities are getting even smaller. We interpret this by the ability by a player to learn by its mistakes and to improve its profit over time in a less noisy environment than in a more noisy one. Figure 2 shows the performance of sensitivity evaluations using both FD and AD.

In our experiments, the function $\pi(\mathbf{x})$ is been evaluated over $10,000$ Brownian paths using a Monte Carlo simulation. We have first checked that both FD and AD gave

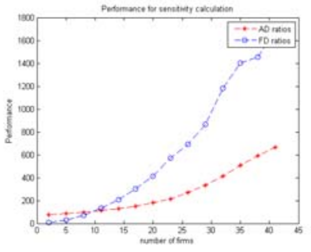

Figure 2: CPU ratios between the Jacobian and function evaluation for both FD and AD

comparable results; the difference between the AD results and FD results were about 10^{-7}. As Figure 2 shows, the FD method is likely to outperform AD for small calculations. When it comes to large-scale computations, the AD is usually faster, see for example [4, 12, 15] for more information about the performance of AD in evaluating derivatives.

4 Conclusions

In this paper, we have presented a thorough overview of the AD technique showing its strengths and weaknesses. We have used it to evaluate sensitivities of expected profit functions in the setting of a Cournot oligopoly model. Numerical results have confirmed the intuition one may have that for each firm, the impact of initial decisions is higher when the level of noise in the system is higher in comparison with less noisy environments. The profit differences between players and the sensitivities decrease when the coefficient σ of the Brownian term is small. Sensitivities may be important in decision making since when errors in decisions imply huge loses, firms' behaviors may be altered accordingly. The use of Automatic Differentiation allows us to get accurate and fast sensitivities. The accuracy of these sensitivities may be of interest in modeling scenarios whereby one has to choose parameters so as to achieve a desired outcome in the system. We believe the AD technique is important and that engineers and practitioners in scientific computing should be aware of in order to avoid spending months to produce hand-coded derivatives.

References

[1] S. P. Anderson, J. K. Goeree, and C. A. Holt. Stochastic game theory: Adjustment to equilibrium under noisy directional learning. *Virginia Economics Online Papers*, 327, 1997.

[2] B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.

[3] H. Fischer. Special problems in automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 43–50. SIAM, Philadelphia, PA, 1991.

[4] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software*, 30(3):266–299, Sep. 2004.

[5] J. C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[6] J. K. Goeree and C. A. Holt. Stochastic game theory: For playing games not just for doing theory. *Proceedings of the National Academy of Sciences of USA*, 96(19):10564–10567, September 2003.

[7] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[8] A. Griewank, C. Bischof, G. Corliss, A. Carle, and K. Williamson. Derivative convergence for iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[9] D. J. Higham. An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM Review*, 43(3):525–546, 2001.

[10] D. J. Higham. *An introduction to financial option valuation: mathematics, stochastics, and computation.* Cambridge University Press, Cambridge, UK, 2004.

[11] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. The complex-step derivative approximation. *ACM Trans. Math. Softw.*, 29(3):245–262, 2003.

[12] J. D. Pryce and E. M. Tadjouddine. Fast automatic differentiation jacobians by compact LU factorization. *SIAM J. Scientific Computing*, 30(4):1659–1677, 2008.

[13] G. D. Pusch, C. Bischof, and A. Carle. On automatic differentiation of codes with COMPLEX arithmetic with respect to real variables. Technical Memorandum ANL/MCS-TM-188, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Casss Avenue, Argonne, IL 60439, June 1995.

[14] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[15] E. M. Tadjouddine. Vertex-ordering algorithms for automatic differentiation of computer codes. *The Computer Journal*, 51(6):688–699, 2008.