Algorithmic Differentiation through Convergent Loops

Barak A. Pearlmutter[*] Hamilton E. Link[†]

Draft of August 22, 2002

Abstract

We consider an explicit iterate-to-fixedpoint operator and derive associated rules for both forward and reverse mode algorithmic differentiation. Like other AD transformation rules, these are exact and efficient. In this case, they generate code which itself invokes the iterate-to-fixedpoint operator. Loops which iterate until a variable changes less than some tolerance should be regarded as approximate iterate-to-fixedpoint calculations. After a convergence analysis, we contend that it is best both pragmatically and theoretically to find the approximate fixedpoint of the adjoint system of the actual desired fixedpoint calculation, rather than find the adjoint of the approximate primal fixedpoint calculation. Our exposition unifies and formalizes a number of techniques already known to the AD community, introduces a convenient and powerful notation, and opens the door to fully automatic efficient AD of a broadened class of codes.

1 Introduction

Algorithmic differentiation (Wengert, 1964; Speelpenning, 1980; Rall, 1981), sometimes known as automatic differentiation or computational differentiation, is fully automatic for straight line code and for loops which could in principle be expanded into straight line code, such as loops through the indices of an array. However, code in which a loop is iterated until some tolerance is reached has been problematic, requiring manual assistance and very rough approximations (Carle and Fagan, 1996; Cappelaere et al., 2001; Hoefkens et al., 2001; Giles, 2001; Gockenbach et al., 2001). The main point of this paper is the introduction of a mathematical abstraction and unified notation for loops that iterate until some tolerance is reached. As in Gilbert (1992), we regard these loops as approximations to a numeric fixedpoint operator, in that they iterate some function until its argument is nearly at a fixedpoint. Thinking of such loops as approximations of an ideal iterate-to-fixedpoint operator makes them amenable to efficient AD.

2 Notation

We use $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ to designate the source-to-source transformations of AD in the forward mode (Wengert, 1964; Kedem, 1980) and reverse mode (Speelpenning, 1980; Rall, 1981), respectively. The operations are applied to a sequence of assignment statements to yield the appropriate assignments for the adjoint variables. Table 1 shows how these operators apply to various sorts of statements. We overload these to apply not only to assignment statements, but also to functions, so $\overrightarrow{\mathcal{J}}\{f\}$ is the function that results from the forward-mode AD transformation of f. The transformed function takes both the original input and the adjoint input, which are written after the terminating brace, $\overrightarrow{\mathcal{J}}\{f\}(\mathbf{x})(\hat{\mathbf{x}})$, or simply $\overrightarrow{\mathcal{J}}\{f\}(\mathbf{x})\hat{\mathbf{x}}$. We also allow these operators to be applied to expressions, with a subscript to designate the inputs, so $\overrightarrow{\mathcal{J}}_a\{f(\mathbf{a},\mathbf{b})\}$ denotes the forward-mode AD transformation of $f(\mathbf{a},\mathbf{b})$ with $\mathbf{a}$ regarded as the input and $\mathbf{b}$ regarded as a constant. The result implicitly uses the input variable's value in the surrounding context, so it requires only the adjoint input. Therefore it is invoked as $\overrightarrow{\mathcal{J}}_{\mathbf{a}}\{f(\mathbf{a},\mathbf{b})\}\hat{\mathbf{a}}$, which yields $\hat{\mathbf{y}}$ with only a constant factor overhead in time and space, where $\hat{y}_j = \sum_i \hat{a}_i \, df_j(\mathbf{a},\mathbf{b})/da_i$.

The reverse mode AD transformation $\overleftarrow{\mathcal{J}}$ is similar, except the adjoint input it requires is the adjoint of the output of the argument, as in $\mathbf{y} = f(\mathbf{x}); \tilde{\mathbf{x}} = \overleftarrow{\mathcal{J}}\{f\}(\mathbf{x})\tilde{\mathbf{y}}$ or $\mathbf{z} = g(\mathbf{a},\mathbf{b}); \tilde{\mathbf{a}} = \overleftarrow{\mathcal{J}}_{\mathbf{a}}\{g(\mathbf{a},\mathbf{b})\}\tilde{\mathbf{z}}$. This yields $\tilde{\mathbf{a}} = \overleftarrow{\mathcal{J}}_{\mathbf{a}}\{f(\mathbf{a},\mathbf{b})\}\tilde{\mathbf{y}}$ where $\tilde{a}_i = \sum_j \tilde{y}_j \, df_j(\mathbf{a},\mathbf{b})/da_i$ with only a constant factor overhead in time, but potentially greater overhead in space.

We consistently use a hat (circumflex) over a symbol to indicate the forward mode (covariant) adjoint of a variable, and a tilde for the reverse mode (contravariant) adjoint. It is important to note that this notation is potentially ambiguous. For instance, we might do a $\overrightarrow{\mathcal{J}}$ transformation on some code that is itself the result of a previous application of $\overrightarrow{\mathcal{J}}$. The forward-mode adjoint of a variable x

[*]Depts of Computer Science and Neurosciences, Univ. New Mexico, Albuquerque, NM 87131, bap@cs.unm.edu. *Corresponding author.*

[†]Sandia National Labs, B836/MS0455, Albuquerque, NM 87135

created by the first application must be distinguished from the adjoint of the same variable x created during the second application. It is thus crucial in an actual implementation to distinguish between these, probably by creating a new token each time a new adjoint variable is created. The examples we give in this manuscript do not suffer from this issue, so for expository simplicity we use the potentially ambiguous adjoint variable notation described above.

This notation was chosen for both conciseness and mnemonic value. We use $\mathcal{J}$ for AD operators because the Jacobian is usually denoted $\mathbf{J}$, and the result of applying an AD operator is a function that can calculate the product with the Jacobian or the Jacobian transpose, for the forward and reverse modes respectively. We use an arrow above the $\mathcal{J}$ pointing forward for forward mode and backwards for reverse mode. The arrow shows not just the direction of accumulation, but also the side of the Jacobian on which the vector is multiplied: right for the right hand side, left for the left hand side. Since the result returned by $\mathcal{J}$ is not just a function but a linear function we allow it to take an argument without surrounding parenthesis, by analogy with matrix multiplication.

3 The Fixedpoint Operator $\mathcal{F}$

Consider a loop of the form

```
for $t = 1, \ldots, \infty$ {
    $\mathbf{a}_t \leftarrow g(\mathbf{a}_{t-1}, \mathbf{b})$;
    if $||\mathbf{a}_t - \mathbf{a}_{t-1}|| \leq \varepsilon$ break; }
$\mathbf{z} = \mathbf{a}_t$;                                   (1)
```

This is an approximation for finding the actual numeric fixedpoint, $\mathbf{z} = \mathbf{a}_\infty$, assuming that $g(\cdot, \mathbf{b})$ has appropriate convergence properties.[1] We will denote this operation

$$\mathbf{z} = \mathcal{F}^{\varepsilon}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\} \tag{2}$$

In general we omit the superscript ε and treat the error introduced by stopping the loop when tolerance ε is reached as negligible. In other words, we treat the fixedpoint operator $\mathcal{F}$ as exact, even though numeric and efficiency issues introduce some error.

Although the loop (1) requires an initial value $\mathbf{a}_0$, we omit this as it does not enter our calculations.[2]

[1]Convergence could be checked using interval arithmetic (Moore, 1966) or online estimation of the maximal eigenvalue of the transfer function (Simard et al., 1991). However, we simply ignore the possibility that the iteration is unstable, *i.e.* we assume the programmer has ensured convergence. This is necessarily the case for correct numeric code. Methods for designing algorithms whose loops can be shown to converge are beyond the scope of this paper.

[2]In general $\mathbf{z}$ should be piecewise constant in the initial value $\mathbf{a}_0$.

4 AD of $\mathcal{F}$

It is common in mathematical physics to consider a function defined implicitly, such as the solution of a differential equation or the minimum of a functional. A common problem is to calculate how such a function changes as various system parameters are modified—this is often called perturbation theory. For instance, how do the positions of the atoms in a crystal lattice, and therefore its density, change as the pressure, temperature, and electric field are varied? Various qualitatively different kinds of changes can be found, such as phase transitions of various orders. But in the very simplest form of this question we ask for the gradient, or more generally the Jacobian, of an implicitly defined function.

In the early twentieth century, such gradients were calculated numerically by the method of finite differences. This is both inaccurate and inefficient, as it not only suffers from roundoff errors but also increases the computational burden by $O(n)$ in approximating the gradient of an $\mathbb{R}^n \to \mathbb{R}$ implicit function. In his undergraduate thesis, Feynman (1939) exhibited a method that reduced the increase in computational burden to a constant factor via the solution of a linear equation involving a matrix that linearizes the fixedpoint. This method was rediscovered in the neural network community (Pineda, 1987; Almeida, 1987) in the context of relaxation networks, which can be regarded as implicitly defined functions, and also in the AD community (Christianson, 1994).

Here we give a general formulation which unifies the concepts of performing AD through a fixedpoint process with some perturbation methods of mathematical physics. The adjoint systems we construct for both forward and reverse mode AD through a fixedpoint-finding loop have the usual property of AD: they do not increase the computational burden beyond a constant factor. In forward mode, the space requirements are increased by at most a factor of two. In reverse mode, the space requirements are increased by at most the amount of space needed by standard reverse mode AD operating on the function being iterated.

We will now derive rules to perform AD through a numeric fixedpoint process. For concreteness, we call the function that performs one iteration of the fixedpoint-finding loop g, and we write the loop itself as

$$\mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\} \tag{3}$$

where $\mathbf{a}$ is the variable being iterated and $\mathbf{b}$ is the controlling input.

4.1 Forward mode AD of $\mathcal{F}$

We wish to perform forward accumulation AD through the numeric fixedpoint process (3). This case, includ-

original	$\overrightarrow{\mathcal{J}}$ **adjoint**	$\overleftarrow{\mathcal{J}}$ **adjoint**
x is input	$\hat{x}$ is adjoint input	$\tilde{x}$ is adjoint output
y is output	$\hat{y}$ is adjoint output	$\tilde{y}$ is adjoint input
constant	0	n/a
$S_1; S_2; \cdots; S_n$	$\overrightarrow{\mathcal{J}}\{S_1\}; \overrightarrow{\mathcal{J}}\{S_2\}; \cdots; \overrightarrow{\mathcal{J}}\{S_n\}$	$\overleftarrow{\mathcal{J}}\{S_n\}; \cdots; \overleftarrow{\mathcal{J}}\{S_2\}; \overleftarrow{\mathcal{J}}\{S_1\}$
$z = a + b$	$\hat{z} = \hat{a} + \hat{b}$	$\tilde{a} = \tilde{z}; \tilde{b} = \tilde{z}$
$z = ab$	$\hat{z} = \hat{a}b + a\hat{b}$	$\tilde{a} = \tilde{z}b; \tilde{b} = \tilde{z}a$
$z = f(a)$	$\hat{z} = \hat{a}f'(a)$	$\tilde{a} = \tilde{z}f'(a)$
$z = f(a_1, \ldots)$	$\hat{z} = \sum_i \hat{a}_i (d/da_i) f(a_1, \ldots)$	$\tilde{a}_i = \tilde{z}\,(d/da_i) f(a_1, \ldots)$
$(z_1, z_2, z_3) = (a, a, a)$	$\hat{z}_1 = \hat{z}_2 = \hat{z}_3 = \hat{a}$	$\tilde{a} = \tilde{z}_1 + \tilde{z}_2 + \tilde{z}_3$
$(z_1, \ldots, z_n) = f(a)$	$\hat{z}_i = (df_i(a)/da)\hat{a}$	$\tilde{a} = \sum_i (df_i(a)/da)\tilde{z}_i$
$\mathbf{z} = f(\mathbf{a})$	$\hat{\mathbf{z}} = \overrightarrow{\mathcal{J}}\{f\}(\mathbf{a})\hat{\mathbf{a}}$	$\tilde{\mathbf{a}} = \overleftarrow{\mathcal{J}}\{f\}(\mathbf{a})\tilde{\mathbf{z}}$
$\mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}$	$\hat{\mathbf{z}} = \mathcal{F}_{\hat{\mathbf{a}}}\{\ \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{a}}$ $+ \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{b}}\ \}$	$\tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\} \cdot$ $\mathcal{F}_{\tilde{\mathbf{a}}}\{\overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}}\}$

Table 1: Standard rules for creation of AD adjoint systems, in $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ notation, augmented with rules for the iterate-to-fixedpoint operator $\mathcal{F}$. The $\overrightarrow{\mathcal{J}}$ adjoint of a variable z is $\hat{z} \equiv \sum_i \hat{x}_i\, dz/dx_i$, where x_i are the inputs. The $\overleftarrow{\mathcal{J}}$ adjoint of a variable z is $\tilde{z} \equiv \sum_i \tilde{y}_i\, dy_i/dz$, where y_i are the outputs. For $\overleftarrow{\mathcal{J}}$, each variable is assumed to appear on the right hand side of at most one equation, and explicit fanout statements $(z_1, z_2, z_3) = (a, a, a)$ can be inserted to ensure this. In the absence of this guarantee, the $\overleftarrow{\mathcal{J}}$ adjoint of a variable is the sum of the values it would be given according to its various occurrences.

ing the convergence analysis, was exhaustively treated by Griewank et al. (1993), and applied in the context of Neural Networks by Pearlmutter (1994).

Consider the effect on $\mathbf{z}$ of an infinitesimal perturbation $\hat{\mathbf{b}}$ of $\mathbf{b}$. Since $\mathbf{z} = g(\mathbf{z}, \mathbf{b})$, we must find $\hat{\mathbf{z}}$ such that $\mathbf{z} + \hat{\mathbf{z}} = g(\mathbf{z} + \hat{\mathbf{z}}, \mathbf{b} + \hat{\mathbf{b}})$. Taking a first order expansion and subtracting $\mathbf{z}$ from each side yields

$$\hat{\mathbf{z}} = \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{z}} + \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{b}} \qquad (4)$$

Although exact solutions are possible (see Section 6), we consider iterating (4) until $\hat{\mathbf{z}}$ stabilizes, at which point the equation is solved. Using the fixedpoint operator, this can be written

$$\begin{gathered} \overrightarrow{\mathcal{J}}\{\mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}\} \\ \Downarrow \\ \hat{\mathbf{z}} = \mathcal{F}_{\hat{\mathbf{a}}}\{\ \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{a}} + \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\}\hat{\mathbf{b}}\ \} \end{gathered} \qquad (5)$$

which is the final formula for forward accumulation AD through a fixedpoint process. As shown in Section 5.2, this will converge if the original fixedpoint process converged, and at the same rate.

4.2 Reverse mode AD of $\mathcal{F}$

We assume we know the sensitivities $\tilde{\mathbf{z}}$ of the output $\mathbf{z}$, and wish to find the corresponding sensitives $\tilde{\mathbf{b}}$ induced in $\mathbf{b}$ which controls the fixedpoint process (3). By consideration of the meaning of $\mathcal{F}$ in that equation we see that

$$\mathbf{z} = \mathbf{a} \qquad (6)$$

where $\mathbf{a}$ was iterated to a fixedpoint, so

$$\mathbf{a} = g(\mathbf{a}, \mathbf{b}). \qquad (7)$$

Since $\mathbf{a}$ appears on the right hand size of both (6) and (7), reverse mode AD yields

$$\tilde{\mathbf{a}} = \overleftarrow{\mathcal{J}}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}} \qquad (8)$$

while $\mathbf{b}$ appears only in (7) so

$$\tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{a}, \mathbf{b})\}\tilde{\mathbf{a}} \qquad (9)$$

We iterate (8) to a solution,

$$\tilde{\mathbf{a}} = \mathcal{F}_{\tilde{\mathbf{a}}}\{\ \overleftarrow{\mathcal{J}}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}}\ \} \qquad (10)$$

Substituting this into (9) and applying (6) yields

$$\begin{gathered}\overleftarrow{\mathcal{J}}\{\ \mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a},\mathbf{b})\}\ \} \\ \Downarrow \\ \tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z},\mathbf{b})\} \cdot \mathcal{F}_{\tilde{\mathbf{a}}}\{\ \overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}}\ \}\end{gathered} \tag{11}$$

which is our final formula for reverse mode AD through a fixedpoint process. This is similar to the "final iteration" rule of Christianson (1994), and in practice we should use the stopping criterion derived in that work for stopping the dual fixedpoint process here. As shown in Section 5.3, this converges if the original fixedpoint process converged, and at the same rate.

5 Convergence

We will show that the asymptotic convergence rate of the primal fixedpoint process determines the convergence rate of both the forward and reverse mode AD transformations. In fact, these iterated functions essentially share a common Jacobian at their respective fixedpoints. This means that, assuming the tolerance on the primal fixedpoint iteration brings one into the asymptotic regime, the expected error in the primal introduced by the tolerance ε is identical in magnitude to that introduced in the transformed code by the same tolerance. (Although the convergence rate is identical, this does not mean that the number of iterations is the same. One might expect the transformed loop to require a small constant number of extra iterations because its starting point is not carefully chosen, while the starting point of the primal loop is usually chosen carefully by the programmer.)

5.1 Convergence of the primal

Consider the asymptotic convergence of equation (3), $\mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a},\mathbf{b})\}$. Labeling the intermediate values in the convergence to a fixedpoint, we have $\mathbf{a}_{t+1} = g(\mathbf{a}_t, \mathbf{b})$ and $\mathbf{a}_t \to \mathbf{z}$. Asymptotic convergence thus requires that any infinitesimal perturbation about the fixedpoint $\mathbf{z}$ be damped. If we define the matrix $\mathbf{J}$ in the first-order Taylor expansion

$$\mathbf{J}\hat{\mathbf{z}} = g(\mathbf{z}+\hat{\mathbf{z}},\mathbf{b}) - \mathbf{z} = \frac{dg(\mathbf{z},\mathbf{b})}{d\mathbf{z}}\,\hat{\mathbf{z}} \tag{12}$$

where $\hat{\mathbf{z}}$ is an infinitesimal perturbation, then all the eigenvalues of $\mathbf{J}$ must lie within the unit circle on the complex plane. In general, the eigenvalue spectrum of $\mathbf{J}$ determines the convergence of (3). In a slight abuse of notation we can conflate linear functions with matrices, so

$$\mathbf{J} = \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\} = \overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}^T \tag{13}$$

5.2 Convergence of $\overrightarrow{\mathcal{J}}\{\mathcal{F}\}$

Consider the convergence of (5). (This was discussed by Griewank et al. (1993) and Bartholomew-Biggs (1998), and also to some extent anticipated by Kedem (1980).) The fixedpoint operator is being applied to a linear function, as the argument to $\mathcal{F}$ is a linear function $\hat{\mathbf{a}}$, so our analysis is simple and not restricted to the asymptotic regime. The constant driving term is irrelevant; all that matters are the eigenvalues of the gains, $\overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}$. However this is precisely $\mathbf{J}$ from the last section, so if the primal converges asymptotically, the forward mode dual converges from any starting point. Moreover, the dual's convergence rate everywhere is the same as the asymptotic convergence rate of the primal.

5.3 Convergence of $\overleftarrow{\mathcal{J}}\{\mathcal{F}\}$

Consider the convergence of (11). Again the fixedpoint operator is being applied to a linear function, so our analysis is not restricted to the asymptotic regime. What matters are the eigenvalues of the gains, $\overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}$. Once more conflating matrices with linear functions, this is the transpose of $\mathbf{J}$ above, and $\mathbf{J}$ and $\mathbf{J}^T$ have the same eigenvalues. So if the primal converges, so does the reverse mode transformed code, and again at the same rate.

6 Exact dual solutions

Equations (4) and (8) are linear, and hence (as noted by Christianson (1998) for the reverse accumulation case) can be solved using more specialized methods than proposed above. In particular, they are each of the form

$$\mathbf{M}\mathbf{x} = \mathbf{c} \tag{14}$$

where in the case of (4)

$$\begin{aligned}\mathbf{M} &= \mathbf{I} - \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\} \\ \mathbf{c} &= \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z},\mathbf{b})\}\hat{\mathbf{b}} \\ \mathbf{x} &= \hat{\mathbf{z}}\end{aligned} \tag{15}$$

and in the case of (8)

$$\begin{aligned}\mathbf{M} &= \mathbf{I} - \overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\} \\ \mathbf{c} &= \tilde{\mathbf{z}} \\ \mathbf{x} &= \tilde{\mathbf{a}}\end{aligned} \tag{16}$$

where $\mathbf{I}$ is the identity matrix, we conflate linear functions with matrices so $\overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}$ represents the Jacobian matrix $\mathbf{J} = dg(\mathbf{z},\mathbf{b})/d\mathbf{z}$, similarly $\overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}$ represents

$\mathbf{J}^T$, and $\tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z},\mathbf{b})\}\mathbf{x}$ rather than $\mathbf{x}$ itself is the desired quantity in the reverse mode case.

This can be solved analytically as $\mathbf{x} = \mathbf{M}^{-1}\mathbf{c}$, or by using a standard solver for linear systems, but the matrix $\mathbf{M}$ may be impractically large to calculate and manipulate explicitly. However, $\mathbf{M}$ is a generalized sparse matrix because we can multiply by it efficiently. Moreover, $\mathbf{M}$ is positive definite since $\mathbf{J}$'s eigenvalues all lie within the unit circle. We can therefore use the method of conjugate gradients to solve for $\mathbf{x}$ exactly in $O(n)$ steps where n is the dimensionality of the system and each step involves one $\mathbf{M}$-multiplication.

7 Programming Languages

Although it may be possible to automatically detect iterate-to-fixedpoint loops in dusty deck code, our initial implementation instead adds a new looping construct. For expository purposes, one can consider a **dofix** looping keyword, which iterates until a listed set of variables (which may be arrays) stop changing (up to the given tolerance), or (for safety) until a given iteration count is reached. If the given iteration count is actually reached, a warning is printed. For example,

```
$x = a/2;$
**dofix**$(x)$ {
    $x = (x + a/x)/2;$
} until (tolerance $0.00001$, iteration $500$);
```

would result in $x = \sqrt{a}$, approximately. This construct simplifies the expression of many loops, and lends itself to AD using the methods described above.

Another possibility is more radical, namely using a purely functional programming language, in which the iterate-to-fixedpoint operator is a first class higher order function, like **map** and **compose**, and the $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators are likewise first-class functions. The full power of a language of this nature requires an augmentation of the notion of derivative to higher order functions, which will be the subject of a future publication.

8 Conclusion

Although the basics of AD through iterate-to-fixedpoint has been known in the AD community for some time, it has not been possible to automatically process codes containing such constructs. By giving a unified treatment along with a convenient notation and simplified derivations, we have built the conceptual infrastructure to enable convenient and even automatic AD of codes containing iterate-to-fixedpoint loops. (The construction of such a system is, naturally, our next endeavor.)

This has many practical applications. To take a particular example, we can regard most p.d.e. solvers as starting with a guess for the solution to the p.d.e. and then iterating an error-reduction function until a sufficiently accurate solution is achieved. Consider a routine for calculating the performance of a given wing shape

$$\begin{aligned}&\text{wing} = \text{spline-to-surface}(\text{spline-control-points});\\&\text{airflow} = \text{pde-solver}(\text{wing}, \text{Navier-Stokes});\\&\text{drag}, \text{lift} = \text{surface-integral}(\text{wing}, \text{airflow}, \text{force});\\&\text{performance} = f(\text{drag}, \text{lift}, \text{weight}(\text{wing}));\end{aligned}$$

which we would like to optimize using a gradient method. The primary difficulty is performing reverse accumulation through the p.d.e. solver at the heart of the routine, which traditional AD systems cannot handle efficiently. But if this solver is expressed using the $\mathcal{F}$ operator, an AD system incorporating knowledge of this operator could automatically and efficiently calculate $\nabla_{\text{spline-control-points}}\text{performance} = \overleftarrow{\mathcal{J}}_{\text{spline-control-points}}\{\text{performance}\}(1)$.

For the AD practitioner, or the scientist wishing to manually apply these methods, the result to remember is:

$$\begin{aligned}&\overrightarrow{\mathcal{J}}\{\ \mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a},\mathbf{b})\}\ \}\\&\qquad\Rightarrow\qquad \hat{\mathbf{z}} = \mathcal{F}_{\hat{\mathbf{a}}}\{\ \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}\hat{\mathbf{a}} + \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z},\mathbf{b})\}\hat{\mathbf{b}}\ \}\\&\overleftarrow{\mathcal{J}}\{\ \mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a},\mathbf{b})\}\ \}\\&\qquad\Rightarrow\qquad \tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z},\mathbf{b})\}\cdot\mathcal{F}_{\tilde{\mathbf{a}}}\{\ \overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z},\mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}}\ \}\end{aligned}$$

These extensions open the door to AD implementations with automatic efficient transformation of a broadened class of codes, including functional programs, optimization routines, p.d.e. solvers, and sophisticated methods used in machine learning.

Acknowledgements

Supported by NSF CAREER award 97-02-311, the National Foundation for Functional Brain Imaging, an equipment grant from Intel corporation, and a gift from the NEC Research Institute. We are grateful for rapid helpful feedback from George F. Corliss, Bruce Christianson, and Cynthia D. Rudin.

References

Almeida, L. B. (1987). A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In Caudill, M. and Butler, C., editors, *IEEE First International Conference on Neural Networks*, volume 2, pages 609–618, San Diego, CA.

Bartholomew-Biggs, M. C. (1998). Using forward accumulation for automatic differentiation of implicitly-defined functions. *Computational Optimization and Applications*, 9:65–84.

Cappelaere, B., Elizondo, D., and Faure, C. (2001). Odyssée versus hand differentiation of a terrain modelling application. In Corliss et al. (2001), chapter 7, pages 75–82.

Carle, A. and Fagan, M. (1996). Improving derivative performance for CFD by using simplified recurrences. In Berz, M., Bischof, C., Corliss, G., and Griewank, A., editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 343–351. SIAM, Philadelphia, PA.

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

Christianson, B. (1998). Reverse accumulation and implicit functions. *Optimization Methods and Software*, 9(4):307–322.

Corliss, G., Faure, C., Griewank, A., Hascoët, L., and Naumann, U., editors (2001). *Automatic Differentiation: From Simulation to Optimization*. Computer and Information Science. Springer, New York, NY.

Feynman, R. (1939). Forces in molecules. *Physical Review*, 56(340).

Gilbert, J.-C. (1992). Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21.

Giles, M. B. (2001). On the iterative solution of adjoint equations. In Corliss et al. (2001), chapter 16, pages 145–151.

Gockenbach, M. S., Reynolds, D. R., and Symes, W. W. (2001). Automatic differentiation and the adjoint state method. In Corliss et al. (2001), chapter 18, pages 161–166.

Griewank, A., Bischof, C., Corliss, G. F., Carle, A., and Williamson, K. (1993). Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355.

Hoefkens, J., Berz, M., and Makino, K. (2001). Efficient high-order methods for ODEs and DAEs. In Corliss et al. (2001), chapter 41, pages 343–348.

Kedem, G. (1980). Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150–165.

Moore, R. E. (1966). *Interval Analysis*. Prentice Hall, Englewood Cliffs, NJ.

Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160.

Pineda, F. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, 19(59):2229–2232.

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

Simard, P. Y., Rayzs, J. P., and Victorri, B. (1991). Shaping the state space landscape in recurrent networks. In *Advances in Neural Information Processing Systems 3*, pages 105–112. Morgan Kaufmann.

Speelpenning, B. (1980). *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL.

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464.