AIAA Guidance, Navigation, and Control Conference
2 - 5 August 2010, Toronto, Ontario Canada

AIAA 2010-7783

An Object-Oriented Method for Computation of Analytic Derivatives

Michael. A. Patterson*
Anil V. Rao†
University of Florida
Gainesville, FL 32611

A new object-oriented method is presented for generating analytic derivatives of functions defined by computer codes. The method developed in this paper combines operator overloading with forward mode automatic differentiation in a manner that enables the construction of string representations of the analytic derivatives of the function. The string representations follow the syntax of the language in which the original function was coded, thus making it possible to write these strings to a new function file that can be evaluated in the same manner as is used to evaluate the original function. Because the computer code for the derivative function created through this process has the same input as the computer code of the original function, the algorithm can be used recursively to generate as many derivatives of the original function as may be needed for a particular application. An important feature of the approach developed in this paper is that the derivatives are generated simply by evaluating the function on an input object, thus making it possible to generate potentially complex derivative expressions that would otherwise have been intractable or would have required tedious and error-prone hand calculations and coding. A MATLAB implementation of the algorithm, called SIMDIFF, is described and an example is performed to show the functionality of the approach.

I. Introduction

As defined in Ref. 1, *automatic differentiation*, also known as *algorithmic differentiation*, is the process of determining accurate derivatives of a function defined by computer programs. The goal of automatic differentiation is to employ ordinary calculus in an algorithmic manner with the goal of obtaining machine precision accuracy derivatives in a computationally efficient manner. Automatic differentiation exploits the fact that a computer code that implements a general function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ can be decomposed into a sequence of elementary operations, each of which can be differentiated using the well known calculus rules for differentiation. Automatic differentiation algorithms are categorized into either *forward and reverse mode* algorithms or *source transformation* algorithms. In either a forward or reverse mode algorithm, the derivative is obtained by implementing the chain rule until all required links in the chain have been implemented. The derivative is then obtained by multiplying the links together. In a source transformation algorithm, the original source code that defines the function is differentiated line by line, resulting in a new code that

*Ph.D. Candidate, Department of Mechanical and Aerospace Engineering. E-mail: mpatterson@ufl.edu.
†Assistant Professor, Department of Mechanical and Aerospace Engineering. E-mail: anilvrao@ufl.edu. Corresponding Author.

contains the derivative of the original function. Many automatic differentiation software programs have been developed over the years. Well known forward and reverse mode programs include *ADMIT-1*,[2] *ADMAT*,[3] *ADOL-C*,[4] and MAD[5] while source transformation tools include *ADIFOR*[6] and TAPENADE.[7]

In this paper we present a new MATLAB package, called SIMDIFF (*simulation of hand differentiation*), for the generation of analytic derivatives. In the approach developed here, an object of the newly developed class SIMDIFF is instantiated. The derivatives of a particular function of interest are then computed by *evaluating* the function on the input SIMDIFF object. In the process of evaluating the function on a SIMDIFF object, forward mode automatic differentiation is used to generate string representations of the derivatives of each link in the calculus chain rule, where each string contains an expression that can be evaluated in MATLAB. The strings corresponding to these chain rule links are then concatenated to produce individual strings that contain the derivative of a particular output with respect to a particular input. These strings can then be written to a new file that can be used to evaluate the derivative at a particular numerical value of the argument. The derivative file generated by this process has the same input as that of the original function.

Because the approach developed in this paper generates analytic derivative expressions by function evaluation, it simulates the computations that a user would perform when determining analytic derivatives by hand on paper. When determining a partial derivative of a function with respect to a particular input, a user assumes that all quantities other than the argument of differentiation are constant. Similarly, by instantiating a SIMDIFF object and evaluating the function on this object, the method of this paper simulates the behavior of computing derivatives by hand by differentiating only those lines of code that are functions of the SIMDIFF object. Using this approach, it is possible to efficiently compute the derivatives of each output with respect to each input, treating all other information in the function as if it were a constant. Since SIMDIFF stores the derivatives as strings, the derivatives array can be used to find the sparsity pattern of the outputs derivatives without the possibility of generating false zeros. The algorithm is described in detail and the approach is demonstrated on an example. The method developed in this paper is found to be a flexible and viable approach to computing derivatives of general MATLAB functions.

II. Notation and Conventions

In this paper we will employ the following notation. First, all $n-$dimensional quantities will be denoted by lower case bold letters and will be treated as *row vectors*. That is, if $\mathbf{x} \in \Re^n$, then

$$\mathbf{x} = \left[\begin{array}{ccc} x_1 & \cdots & x_n \end{array} \right] \in \Re^n$$

Similarly, any function $\mathbf{f}(\mathbf{x})$ that operates on $n-$dimensional row vectors is assumed to produce an $m-$dimensional row vector as an output, that is

$$\mathbf{f}(\mathbf{x}) = \left[\begin{array}{ccc} f_1(\mathbf{x}) & \cdots & f_m(\mathbf{x}) \end{array} \right] \in \Re^m$$

Furthermore, the Jacobian of $\mathbf{f} : \Re^n \longrightarrow \Re^m$, denoted $\partial \mathbf{f}/\partial \mathbf{x}$, is assumed to be an $n \times m$ of the form

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \left[\begin{array}{ccc} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_1}{\partial x_n} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{array} \right] \in \Re^{n \times m}$$

With regard to implementation on a computer, the algorithm of this paper utilizes MATLAB object-oriented programming with operator overloading. Whenever we refer to an object, we will use a capital blackboard bold character (e.g., $\mathbb{X}$), and any property of an object will be denoted by a lower-case bold character (e.g., the notation $\mathbb{X}.\mathbf{a}$ denotes the property $\mathbf{a}$ of the object $\mathbb{X}$).

Finally, it is important to understand that the algorithm developed in this paper assumes that functions have been codes in a *vectorized* manner. In other words, it is assumed that a code for a general function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where $\mathbf{f} : \Re^n \longrightarrow \Re^m$, has been written so that all operations on scalars apply transparently to vectors, matrices, and higher dimensional arrays. Suppose further that an $N \times n$ matrix has been assembled from the N row vectors $\mathbf{x}_i \in \Re^n,\ (i = 1, \ldots, N)$, that is

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}.$$

The function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ can then be evaluated in MATLAB in a vectorized manner on the array $\mathbf{X}$ to return a matrix $\mathbf{Y}$, where each row of $\mathbf{Y}$ corresponds to a value $\mathbf{y}_i \in \Re^m$ (thus making $\mathbf{Y}$ a matrix of size $N \times m$). The vectorized implementation of $\mathbf{y} = \mathbf{f}(\mathbf{x})$ enables simultaneous evaluation of the function at the N values $(\mathbf{x}_1, \ldots, \mathbf{x}_N)$. In the algorithm of this paper, we assume that the functions have been coded in a vectorized manner, making it possible to evaluate the function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ simultaneously on a set of values of the input.

III. Forward Mode Automatic Differentiation

Without loss of generality, consider a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \Re^n \longrightarrow \Re^m$. Assume that $\mathbf{f}(\mathbf{x})$ can be reduced to a sequence of elementary function operations (that is, polynomial, trigonometric, and exponential functions) and has well-defined first and second derivatives. Finally, suppose that a computer implementation of the $\mathbf{f}(\mathbf{x})$ has been written in the MATLAB programming language. The goal is to generate algorithmically a new computer code that contains the Jacobian, $\mathbf{J}(\mathbf{x}) = \partial\mathbf{f}/\partial\mathbf{x}$, and the Hessian, $\mathbf{H}(\mathbf{x}) = \partial^2\mathbf{f}/\partial\mathbf{x}^2$, of the original MATLAB computer implementation of $\mathbf{f}(\mathbf{x})$ (we note that third-order and higher derivatives can be generated in the same recursive manner as is used to generate the first-order and second-order derivatives).

The algorithm developed in this paper uses a *forward mode* approach to compute the analytic derivatives of the function with respect to the input argument. Because the function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ will in general be comprised of intermediate steps that are composite functions of the inputs $\mathbf{x}$, determining the derivative will require using the calculus chain rule to differentiate a composite function of the form $\mathbf{z} = \mathbf{g}(\mathbf{h}(\mathbf{x}))$. Using our convention that all inputs and outputs are row vectors, we have from the calculus chain rule that

$$\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \frac{\partial \mathbf{g}}{\partial \mathbf{h}}, \tag{1}$$

where, reading from right to left, Eq. (1) is a mathematical representation of what is commonly known as the *forward mode* of automatic differentiation. In the forward mode, the chain rule is repeatedly used until the last step in the chain is encountered where the derivative of the innermost function with respect to the input is obtained. The chains are then multiplied together in an appropriate manner (that is, taking into account the proper dimensioning of the derivatives encountered in each step of the chain). Thus, for a multiple-layer composite function of the form

$$\mathbf{z} = \mathbf{f}_1(\mathbf{f}_2(\cdots \mathbf{f}_K(\mathbf{x})),$$

The forward mode of automatic differentiation would be a computer implementation of the derivative

$$\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}} = \frac{\partial \mathbf{f}_K}{\partial \mathbf{x}} \frac{\partial \mathbf{f}_{K-1}}{\partial \mathbf{f}_K} \cdots \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_3} \frac{\partial \mathbf{f}_1}{\partial \mathbf{f}_2} \tag{2}$$

where each link in the chain is of an appropriate dimension so that the products in Eq. (2) make mathematical sense.

IV. SIMDIFF Class and SIMDIFF Constructor Function

The algorithm of this paper is developed via a new overloaded MATLAB object class called *SIMDIFF*, where any SIMDIFF object has the following properties:

(i) **argument:** a MATLAB cell array containing strings with the names of the arguments upon which the function depends.

(ii) **function:** a MATLAB cell array that contains string representations of the original functions of the input arguments.

(iii) **derivative:** a multiple-level MATLAB cell array where the the i^{th} element of the first level of the cell array is itself a cell array that contains string representations of the derivatives of the output with respect to the i^{th} input SIMDIFF argument, and the second level of the cell array contains string representations of the derivatives of the output with respect to each component of the variable represented by the i^{th} input SIMDIFF argument.

A SIMDIFF object $\mathbb{X}$ is instantiated via the SIMDIFF constructor function, where the newly instantiated SIMDIFF object is a representation of the identity function $\mathbf{y} = \mathbf{x}$. The following three inputs are required in order to instantiate a constructor SIMDIFF object: (1) the argument number of the SIMDIFF argument (this is used to separate information from multiple SIMDIFF objects in a non-overlapping manner); (2) a string containing the name of the SIMDIFF argument (any name can be used as long it is not a reserved MATLAB function); and (3) the dimension of the user defined argument. To show how the SIMDIFF constructor function instantiates a new SIMDIFF object, the SIMDIFF constructor function code is shown in Fig. 1.

```
function y = simdiff(argnumber, argname, argsize)
% SIMDIFF object creation function
% created by Michael Patterson

c = argsize;  % the size of the argument
n = argnumber; % the argument number
y.argument{n,1} = argname; % argname is the argument name
for ccount = 1:c;
    y.function{1,ccount} = [argname,'(:,',num2str(ccount),')'];
    for rcount = 1:c;
        if rcount == ccount;
            derivative{rcount,ccount} = '1';
        else
            derivative{rcount,ccount} = '0';
        end
    end
end
y.derivative{n,1} = derivative;
y = class(y,'simdiff');
```

Figure 1: SIMDIFF Constructor Function.

As a starting point, consider the instantiation of a SIMDIFF object $\mathbb{X}$ corresponding to a vector $\mathbf{x} \in \mathbb{R}^3$, such that $\mathbb{X}$ is the only input argument to a function of the form $\mathbf{y} = \mathbf{f}(\mathbf{x})$. Then $\mathbb{X}$ is instantiated in MATLAB as follows:

```
argument_number = 1;
argument_name   = 'x';
argument_size   = 3;
x = simdiff(argument_number, argument_name, argument_size);
```

The object properties **argument**, **function**, and **derivative** that result from the instantiation of the SIMDIFF object $\mathbb{X}$ can then be displayed in MATLAB as

```
Argument(s) of SIMDIFF object:
    'x'

Function(s) of SIMDIFF Object:
    'x(:,1)'    'x(:,2)'    'x(:,3)'

Derivative(s) of SIMDIFF Object:
    {3x3 cell}
```

where the contents of the cell array $\mathbb{X}$.**derivative** is given as

```
>> x.derivative{1}

    '1'    '0'    '0'
    '0'    '1'    '0'
    '0'    '0'    '1'
```

V. Functions of Multiple SIMDIFF Arguments

Consider multiple SIMDIFF constructor objects that have been assigned unique argument numbers. Then SIMDIFF object that is the result of evaluating a function using these constructor SIMDIFF objects will depend upon all of the input SIMDIFF objects such that the derivatives corresponding to each input SIMDIFF argument are stored in a non-overlapping manner. Specifically, the syntax $\mathbb{X}.\mathbf{derivative}\{i\}\{j,k\}$ is used to denote the partial derivative of the k^{th} component of the output with respect to the j^{th} component of the i^{th} argument.

In order to explain how functions with multiple input arguments can be differentiated using SIMDIFF, consider the following two *equivalent* functions:

$$\begin{aligned} \mathbf{y} &= \mathbf{f}(\mathbf{x}) \\ \mathbf{z} &= \mathbf{g}(x_1, x_2) \end{aligned} \tag{3}$$

In Eq. (3), the vector $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \in \Re^2$ and the functions $\mathbf{f}(\mathbf{x})$ and $\mathbf{g}(x_1, x_2)$ perform the same operations to obtain the outputs $\mathbf{y}$ and $\mathbf{z}$. As a result, we have

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \dfrac{\partial \mathbf{g}}{\partial x_1} \\ \dfrac{\partial \mathbf{g}}{\partial x_2} \end{bmatrix}$$

Suppose now that the function $\mathbf{z} = \mathbf{f}(\mathbf{x})$ is evaluated using a two-dimensional SIMDIFF object $\mathbb{X}$ that represent the variable $\mathbf{x}$, and that $\mathbf{z} = \mathbf{g}(x_1, x_2)$ is evaluated using the SIMDIFF objects $\mathbb{X}_1$ and $\mathbb{X}_2$ that represent the scalar variables x_1 and x_2, respectively. Furthermore, suppose the output SIMDIFF objects from these two function evaluations are $\mathbb{Y}$ and $\mathbb{Z}$, respectively. Then the information contained in the **function** and **derivative** properties of the SIMDIFF object $\mathbb{Y}$ would be the same as the information stored in the **function** and **derivative** properties of the SIMDIFF object $\mathbb{Z}$. The only difference between the SIMDIFF objects $\mathbb{Y}$ and $\mathbb{Z}$ is the manner in which the derivative information in each SIMDIFF object is stored. Specifically, $\mathbb{Y}.\mathbf{derivative}\{1\}\{1,1\} = \mathbb{Z}.\mathbf{derivative}\{1\}\{1,1\}$ and $\mathbb{Y}.\mathbf{derivative}\{1\}\{2,1\} = \mathbb{Z}.\mathbf{derivative}\{2\}\{1,1\}$. Because the two main differences between functions evaluated using multiple SIMDIFF arguments and functions evaluated using a single SIMDIFF argument are (1) how the function is written and (2) the manner in which the corresponding derivatives are stored, without loss of generality from this point forth we can restrict our attention to using SIMDIFF to represent a single variable.

VI. Using SIMDIFF to Find Derivatives

Consider again a single-input function of the form $\mathbf{y} = \mathbf{f}(\mathbf{x})$. Suppose that a SIMDIFF object $\mathbb{X}$ corresponding to the variable $\mathbf{x}$ has been instantiated. Assuming that overloaded versions have been developed for every function contained in the MATLAB implementation of the function $\mathbf{f}(\mathbf{x})$, the analytic derivatives $\partial\mathbf{f}/\partial\mathbf{x}$ are computed simply by *evaluating* the function $\mathbf{f}(\mathbf{x})$ on the SIMDIFF object $\mathbb{X}$. The set of overloaded functions written for SIMDIFF objects include all of the commonly used smooth unary operators (i.e., polynomial, trigonometric, hyperbolic trigonometric, exponential, and logarithmic functions) and the binary arithmetic operators (i.e., $+$ and $-$, $.*$ (array multiplication) and $./$ (array division)). When a function operates on a SIMDIFF object, the output of the function is another SIMDIFF object that contains the original function plus the analytic derivative of the function with respect to the SIMDIFF input. For composite functions, the analytic functions and derivatives are automatically propagated at each step, being assembled from the stored analytic functions and derivatives of the input objects using the calculus chain rule until the constructor SIMDIFF object is reached (at which point the propagation process stops because the derivative of a constructor SIMDIFF object is an identity matrix). When sub referencing specific elements of a SIMDIFF object, those elements of the **function** property are returned and the **derivative** property will contain all the derivatives of the referenced functions with respect to every variable in the argument.

A. Differentiation of Unary Functions

To demonstrate how unary operations are applied to SIMDIFF objects the overloaded function for $\sin(\mathbf{x})$ is shown. The unary operator logic, shown in Fig. 2, operates on the constructor SIMDIFF object $\mathbf{x}$ as follows. First, the **function** property of the output is a string representation of the function (in this case $\sin(\cdot)$) of the input SIMDIFF object (in this case, $\mathbf{x}$). Next, the **derivative** property of the output is a string representation of the derivative of the function with respect to the input SIMDIFF object. Note that the **derivative** property of the output SIMDIFF object is the derivative of the function (in this case, $\cos(\cdot)$) of the function with respect to the variable defined by the constructor SIMDIFF object. A similar process to that shown for $\sin(\mathbf{x})$ is applied to all other unary functions, the only difference being the specific calculus rule for computing the derivative of another function (e.g., the derivative with respect to $\mathbf{x}$ of $\sin(\mathbf{x})$ is $\cos(\mathbf{x})$, the derivative of $\log(\mathbf{x})$ is $1/\mathbf{x}$, etc.). In all cases of unary functions, the derivative of the output with respect to the constructor SIMDIFF object are stored as strings that can be evaluated or written to a function file.

B. Differentiation of Binary Functions

Now consider a binary function of the form $\mathbf{z} = \mathbf{f}(\mathbf{g}(\mathbf{x}), \mathbf{h}(\mathbf{x}))$, where $\mathbf{g}(\mathbf{x})$ and $\mathbf{h}(\mathbf{x})$ are functions of the variable $\mathbf{x}$. Assume that $\mathbf{g}(\mathbf{x})$ and $\mathbf{h}(\mathbf{x})$ have been evaluated on the SIMDIFF object $\mathbb{X}$ that corresponds to the variable $\mathbf{x}$. Then the evaluations of $\mathbf{g}$ and $\mathbf{h}$ on $\mathbb{X}$ leads to SIMDIFF objects $\mathbb{G}$ and $\mathbb{H}$, respectively. If the derivatives of the function $\mathbf{f}$ are to be computed with respect to the variable $\mathbf{x}$, the SIMDIFF objects $\mathbb{G}$ and $\mathbb{H}$ must either be the same size, or one of the arguments must of dimension one (that is, $\mathbf{g}(\mathbf{x}) \in \Re$ or $\mathbf{h}(\mathbf{x}) \in \Re$). The following two cases for $\mathbb{G}$ and $\mathbb{H}$ must then be considered:

Case 1: If the SIMDIFF objects $\mathbb{G}$ and $\mathbb{H}$ share no common components, the **function** property of $\mathbb{Z}$ is determined using the **function** property of both $\mathbb{G}$ and $\mathbb{H}$. The **derivative** property of $\mathbb{Z}$ is then determined by differentiating $\mathbf{f}(\mathbf{g}(\mathbf{x}), \mathbf{h}(\mathbf{x}))$ independently with respect to each SIMDIFF object $\mathbb{G}$ and $\mathbb{H}$.

```
function y = sin(x)
% SIMDIFF sin function
% created by Michael Patterson

y.argument        = x.argument;
[r, c]       = size(x.function); % c represents the number of functions
[varr, varc] = size(x.argument);
for ccount = 1:c;  % F sim x
    y.function{1,ccount} = ['sin(',x.function{1,ccount},')'];
end
for varcount = 1:varr;
    if isempty(x.argument{varcount,1})
        y.derivative{varcount,1} = [];
    else
        [dr, dc] = size(x.derivative{varcount,1});
        %dr represents the number of variables
        %dc represents the number of functions
        for ccount = 1:dc;
            for rcount = 1:dr;
                if strcmp(x.derivative{varcount,1}{rcount,ccount},'0')
                    derivative{rcount,ccount} = '0';
                else  % dF sim x, rep x
                    if strcmp(x.derivative{varcount,1}{rcount,ccount},'1')
                        derivative{rcount,ccount} = ['cos(',x.function{1,ccount},')'];
                    elseif strcmp(x.derivative{varcount,1}{rcount,ccount},'-1')
                        derivative{rcount,ccount} = ['-cos(',x.function{1,ccount},')'];
                    else
                        derivative{rcount,ccount} = ['cos(',x.function{1,ccount},').*...
                        (',x.derivative{varcount,1}{rcount,ccount},')'];
                    end
                end
            end
        end
        y.derivative{varcount,1} = derivative;
    end
end
y = class(y,'simdiff');
```

Figure 2: SIMDIFF Overloaded Unary Sine Function.

Case 2: If the SIMDIFF objects $\mathbb{G}$ and $\mathbb{H}$ share common components, then the **function** property of $\mathbb{Z}$ is determined using the **function** property of both $\mathbb{G}$ and $\mathbb{H}$. The **derivative** property of $\mathbb{Z}$ is determined by implementing necessary binary rule for $\mathbf{f}(\mathbf{g}(\mathbf{x}), \mathbf{h}(\mathbf{x}))$ (that is, the product rule, the quotient rule, etc.) with respect to the overlapping variables between the SIMDIFF objects $\mathbb{G}$ and $\mathbb{H}$.

If it turns out that either input $\mathbf{g}(\mathbf{x})$ or $\mathbf{h}(\mathbf{x})$ is not a function of $\mathbf{x}$, then only one of the input arguments to the function will be a SIMDIFF object (i.e., either the first or second argument, but not both, is a SIMDIFF object). In this situation, the following third case must be considered:

Case 3: The **function** property of output SIMDIFF object $\mathbb{Z}$ is determined using the **function** property of the input SIMDIFF object together with a string representation of the floating point input. The **derivative** property of $\mathbb{Z}$ is determined using the chain rule taking derivatives of with respect to the variables of the input SIMDIFF object.

The binary operations contain several logic operations that are used to determine the particular case to implement. First a logic operation is performed to determine the class of each input argument. If both of the input arguments are SIMDIFF objects either **Case 1** or **Case 2** will be implemented to define the **derivative** property of $\mathbb{Z}$. If only one of the input arguments is a SIMDIFF object, then **Case 3** is used to define the **derivative** property. The use of **Case 1** or **Case 2** is determined by analyzing the **derivative** property of both input arguments, then inspecting the elements of each column in **derivative** cell array of the inputs and performing a check of the elements of the same row to determine if the stored derivative function of each argument is nonzero. If only one of the input arguments for a function has a nonzero derivative with respect to a variable, then that element of the **derivative** property cell array of $\mathbb{Z}$ corresponding to the function and variable will be determined via **Case 1**. If both inputs contain nonzero derivatives for a function with respect to

a variable, then **Case 2** is implemented to determine the corresponding element of the **derivative** property cell array of $\mathbb{Z}$.

To demonstrate how binary operations are applied to SIMDIFF objects, suppose it is desired to perform an element-by-element multiplication on two $n-$dimensional vectors $\mathbf{x}$ and $\mathbf{y}$. In MATLAB, this element-by-element would be accomplished via the command $\mathbf{z} = \mathbf{x}.*\mathbf{y}$. Assume that we have instantiated SIMDIFF objects $\mathbb{X}$ and $\mathbb{Y}$ corresponding to the $n-$dimensional vectors $\mathbf{x}$ and $\mathbf{y}$. Fig. 3 and Fig. 7 show MATLAB code for how the **function** and **derivative** properties are determined for $\mathbf{z}$ using **Case 3** when $\mathbf{x}$ is a SIMDIFF object and $\mathbf{y}$ is of class double. The functions for $\mathbf{z}$ are created using the information from the **function** property of $\mathbf{x}$ and string representations of the numeric values of $\mathbf{y}$. The **derivative** property of $\mathbb{Z}$ for the previously mentioned case is defined using information from the **derivative** property of $\mathbb{X}$ and string representations of the numeric values of $\mathbb{Y}$.

```
.
.
.
if isa(x,'simdiff') && isa(y,'double')
    z.argument        = x.argument;
    [nill, yc]   = size(y);
    [xr, xc]     = size(x.function); % xc is number of functions in x
    [varr, varc] = size(x.argument); % varr = number of arguments in x
    if nill > 1
        error('incorrect size')
    end
    if yc == 1 || yc == xc
        if yc == 1
            y = y.*ones(1,xc);
        end
        for ccount = 1:xc;
        % determine function for x = simdiff, y = double
            if y(1,ccount) == 1
                z.function{1,ccount} = x.function{1,ccount};
            else
                z.function{1,ccount} = ['(',x.function{1,ccount},').*...
                (',num2str(y(1,ccount)),')'];
            end
        end
    else
        error('incorrect sizes')
    end
.
.
.
```

Figure 3: SIMDIFF Overloaded Binary Times Function for Case 1.

```
.
.
.
elseif isa(x,'simdiff') && isa(y,'simdiff')
    [xvarr, xvarc] = size(x.argument); % xvarr = number of arguments in x
    [yvarr, yvarc] = size(y.argument); % yvarr = number of arguments in y
    z.argument{1,1} = [];
    % make arguments x and y have same size
    if xvarr > yvarr
        y.argument{xvarr,1}   = [];
        y.derivative{xvarr,1} = [];
        varr                  = xvarr;
    elseif xvarr < yvarr
        x.argument{yvarr,1}   = [];
        x.derivative{yvarr,1} = [];
        varr                  = yvarr;
    else
        varr = xvarr;
    end
    % get function
    [xr, xc] = size(x.function); % xc is number of functions in x
    [yr, yc] = size(y.function); % yc is number of functions in y
    if xc == yc
        for ccount = 1:xc;  % F sim x sim y
            z.function{1,ccount} = ['(',x.function{1,ccount},').*...
            (',y.function{1,ccount},')'];
        end
    else
        error('incorrect sizes')
    end
.
.
.
```

Figure 4: SIMDIFF Overloaded Binary Times Function, Define **function** property for Cases 1 and 2.

```
.
.
.
% varcount = 1: number of arguments
% rcount = 1: number of variable in argument
% ccount = 1: number of functions
% take derivatives with respect to y, not function of x
if strcmp(x.derivative{varcount,1}{rcount,ccount},'0')
   % take derivatives with respect to y, not function of x
   if strcmp(y.derivative{varcount,1}{rcount,ccount},'1')
      derivative{rcount,ccount} = ['(',x.function{1,ccount},')'];
   elseif strcmp(y.derivative{varcount,1}{rcount,ccount},'-1')
      derivative{rcount,ccount} = ['-(',x.function{1,ccount},')'];
   else
      derivative{rcount,ccount} = ['(',x.function{1,ccount},').*...
      (',y.derivative{varcount,1}{rcount,ccount},')'];
   end
else
   % take derivatives with respect to x, not function of y
   if strcmp(x.derivative{varcount,1}{rcount,ccount},'1')
      derivative{rcount,ccount} = ['(',y.function{1,ccount},')'];
   elseif strcmp(x.derivative{varcount,1}{rcount,ccount},'-1')
      derivative{rcount,ccount} = ['-(',y.function{1,ccount},')'];
   else
      derivative{rcount,ccount} =...
      ['(',x.derivative{varcount,1}{rcount,ccount},').*...
      (',y.function{1,ccount},')'];
   end
end
.
.
.
z.derivative = derivative;
.
.
.
```

Figure 5: SIMDIFF Overloaded Binary Times Function, Define **derivative** property for Case 1.

```
.
.
.
% varcount = 1: number of arguments
% rcount = 1: number of variables
% ccount = 1: number of functions
if ~strcmp(x.derivative{varcount,1}{rcount,ccount},'0') &&...
   ~strcmp(y.derivative{varcount,1}{rcount,ccount},'0')
   % take derivative respect to x
   if strcmp(x.derivative{varcount,1}{rcount,ccount},'1')
      LS = ['(',y.function{1,ccount},')'];
   elseif strcmp(x.derivative{varcount,1}{rcount,ccount},'-1')
      LS = ['-(',y.function{1,ccount},')'];
   else
      LS = ['(',x.derivative{varcount,1}{rcount,ccount},').*...
      (',y.function{1,ccount},')'];
   end
   % take derivative respect to y
   if strcmp(y.derivative{varcount,1}{rcount,ccount},'1') ||...
      strcmp(y.derivative{varcount,1}{rcount,ccount},'-1')
      RS = ['(',x.function{1,ccount},')'];
   else
      RS = ['(',x.function{1,ccount},').*...
      (',y.derivative{varcount,1}{rcount,ccount},')'];
   end
   % apply product rule
   if strcmp(y.derivative{varcount,1}{rcount,ccount},'-1')
      derivative{rcount,ccount} = ['(',LS,' - ',RS,')'];
   else
      derivative{rcount,ccount} = ['(',LS,' + ',RS,')'];
   end
else
.
.
.
z.derivative = derivative;
.
.
.
```

Figure 6: SIMDIFF Overloaded Binary Times Function, Define **derivative** property for Case 2.

```
.
.
.
% varr = number of arguments in x
for varcount = 1:varr;
   if isempty(x.argument{varcount,1})
      z.derivative{varcount,1} = [];
   else
      [dxr, dxc] = size(x.derivative{varcount,1});
      %dxr represents the number of variables
      %dxc represents the number of functions
      % find derivatives of all functions with respect to all variables
      for ccount = 1:dxc;
         for rcount = 1:dxr;
            if strcmp(x.derivative{varcount,1}{rcount,ccount},'0')
               derivative{rcount,ccount} = '0';
            else  % dF sim x double y
               if strcmp(x.derivative{varcount,1}{rcount,ccount},'1')
                  derivative{rcount,ccount} = ['(',num2str(y(1,ccount)),')'];
               elseif strcmp(x.derivative{varcount,1}{rcount,ccount},'-1')
                  derivative{rcount,ccount} = ['-(',num2str(y(1,ccount)),')'];
               else
                  derivative{rcount,ccount} =...
                  ['(',x.derivative{varcount,1}{rcount,ccount},').*...
                  (',num2str(y(1,ccount)),')'];
               end
            end
         end
      end
      z.derivative{varcount,1} = derivative;
   end
end
.
.
.
```

Figure 7: SIMDIFF Overloaded Binary Times Function for Case 3.

VII. Example: System of Equations

Consider the following system of equations

$$y = \sin(x_1 x_2)u, \tag{4}$$

where $\mathbf{x} = [x_1\ x_2] \in \mathbb{R}^2$, and $u \in \mathbb{R}$. A MATLAB function code for this system of equations can be written as follows:

```
function y = func(x,u)
y = sin(x(:,1).*x(:,2)).*u;
```

The SIMDIFF object variables for $\mathbf{x}$ and u would then be instantiated as

```
>> x = simdiff(1, 'x', 2)
Argument(s) of SIMDIFF object:
    'x'
Function(s) of SIMDIFF Object:
    'x(:,1)'    'x(:,2)'
Derivative(s) of SIMDIFF Object:
    {2x2 cell}

>> u = simdiff(2, 'u', 1)
Argument(s) of SIMDIFF object:
     []
    'u'
Function(s) of SIMDIFF Object:
    'u(:,1)'
Derivative(s) of SIMDIFF Object:
           []
    {1x1 cell}
```

The argument numbers for the input SIMDIFF objects $\mathbb{X}$ and $\mathbb{U}$ corresponding the first and second input variables $\mathbf{x}$ and u are set to one and two, respectively. Furthermore, it is seen that the size of the SIMDIFF object $\mathbb{X}$ is two (because the variable $\mathbf{x}$ is two-dimensional), while the size of the SIMDIFF object $\mathbb{U}$ is one (because the variable u is a scalar). Finally, the variable names in the SIMDIFF objects $\mathbb{X}$ and $\mathbb{U}$ are "x" and "u", respectively. Evaluating the function on the SIMDIFF objects $\mathbb{X}$ and $\mathbb{U}$, we obtain

```
>> y = func(x,u)
Argument(s) of SIMDIFF object:
    'x'
    'u'
Function(s) of SIMDIFF Object:
    '(sin((x(:,1)).*(x(:,2)))).*(u(:,1))'
Derivative(s) of SIMDIFF Object:
    {2x1 cell}
    {1x1 cell}
```

The output SIMDIFF object $\mathbb{Y}$ contains all of the information pertaining to the input SIMDIFF objects, the output function, and the derivatives of the output with respect to each element of each input. Analyzing the output it can seen that $\mathbb{Y}$ is a one-dimensional SIMDIFF object that is a function of both $\mathbb{X}$ and $\mathbb{U}$. Furthermore, $\mathbb{Y}$ has derivatives with respect to both $\mathbb{X}$ and $\mathbb{U}$. As shown below, the **derivative** property of the SIMDIFF object $\mathbb{Y}$ contains string representations of the derivatives $\partial y/\partial x_1$, $\partial y/\partial x_2$, and $\partial y/\partial u$.

```
>> y.derivative{1}{1}
% dy_dx1
(cos((x(:,1)).*(x(:,2))).*((x(:,2)))).*(u(:,1))
>> y.derivative{1}{2}
% dy_dx2
(cos((x(:,1)).*(x(:,2))).*((x(:,1)))).*(u(:,1))
>> y.derivative{2}{1}
% dy_du
(sin((x(:,1)).*(x(:,2))))
```

The derivative functions contained in the string $\mathbb{Y}$.**derivative** can then be evaluated again using the SIMDIFF objects $\mathbb{X}$ and $\mathbb{U}$ to return the second derivative values of $y = f(\mathbf{x}, u)$,

```
>> dy_dx1 = (cos((x(:,1)).*(x(:,2))).*((x(:,2)))).*(u(:,1));

>> dy_dx2 = (cos((x(:,1)).*(x(:,2))).*((x(:,1)))).*(u(:,1));

>> dy_du = (sin((x(:,1)).*(x(:,2))));

>> dy = [dy_dx1, dy_dx2, dy_du];
```

The **derivative** property of the SIMDIFF object "**dy**" contains the second derivatives of $y = f(\mathbf{x}, u)$. The string representations of all first and second derivatives of y with respect to x_1, x_2, and u are given as

```
>> dy.derivative{1}{1,1}
% dy_dx1_dx1
((-sin((x(:,1)).*(x(:,2))).*((x(:,2)))).*(x(:,2))).*(u(:,1))

>> dy.derivative{1}{2,1}
% dy_dx1_dx2
(((-sin((x(:,1)).*(x(:,2))).*((x(:,1)))).*(x(:,2)) +...
(cos((x(:,1)).*(x(:,2)))))).*(u(:,1))

>> dy.derivative{1}{1,2}
% dy_dx2_dx1
(((-sin((x(:,1)).*(x(:,2))).*((x(:,2)))).*(x(:,1)) +...
(cos((x(:,1)).*(x(:,2)))))).*(u(:,1))

>> dy.derivative{1}{2,2}
% dy_dx2_dx2
((-sin((x(:,1)).*(x(:,2))).*((x(:,1)))).*(x(:,1))).*(u(:,1))

>> dy.derivative{1}{1,3}
% dy_du_dx1
cos((x(:,1)).*(x(:,2))).*((x(:,2)))

>> dy.derivative{1}{2,3}
% dy_du_dx2
cos((x(:,1)).*(x(:,2))).*((x(:,1)))

>> dy.derivative{2}{1,1}
% dy_dx1_du
((cos((x(:,1)).*(x(:,2)))).*(x(:,2)))

>> dy.derivative{2}{1,2}
% dy_dx2_du
((cos((x(:,1)).*(x(:,2)))).*(x(:,1)))

>> dy.derivative{2}{1,3}
% dy_du_du
0
```

Thus, this examples demonstrates the ability of SIMDIFF to compute first and second derivatives of multiple-input functions.

VIII. Conclusions

A method has been described for computing analytic derivatives of MATLAB functions. The derivatives of a general function, stored as strings, can be obtained by evaluating a function of interest. Because the computer code for the derivative function created through this process has the same input as the computer code of the original function, the algorithm can be used recursively to generate as many derivatives of the original function as may be needed for a particular application. As a result, the algorithm provides the ease of use associated with forward mode automatic differentiation while simultaneously providing the same functionality as would be obtained had a human derived and coded the analytic derivatives. The method was demonstrated on an example to demonstrate its utility.

References

[1]Griewank, A., *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Frontiers in Appl. Mathematics*, No. 19. SIAM, Philadelphia, Pennsylvania, 2000.

[2]Coleman, T. F. and Verma, A., "ADMIT-1: Automatic differentiation and MATLAB interface toolbox," *ACM Transactions on Mathematical Software*, Vol. 26, No. 1, March 1998b, pp. 150–175.

[3]Coleman, T. F. and Verma, A., *ADMAT: An automatic differentiation toolbox for MATLAB. Tech. rep*, Computer Science Department, Cornell University, 1998a.

[4]Griewank, A., Juedes, D., and Utke, J., "Algorithm 755: ADOL-C, a package for the automatic differentiation of algorithms written in C/C++," *ACM Transactions on Mathematical Software*, Vol. 22, No. 2, 1996, pp. 131–167.

[5]Forth, S. A., "An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB," *ACM Transactions on Mathematical Software*, Vol. 32, No. 2, 2006, pp. 195–222.

[6]Bischof, C. H., Carle, A., Corliss, G. F., Griewank, A., and Hovland, P. D., "ADIFOR: Generating Derivative Codes from Fortran Programs," *Scientific Programming*, Vol. 1, No. 1, 1992, pp. 11–29.

[7]Hascoët, L. and Pascual, V., "TAPENADE 2.1 User's Guide," Rapport technique 300, INRIA, Sophia Antipolis, 2004.