An Efficient Overloaded Method for Computing Derivatives of Mathematical Functions in MATLAB

Michael A. Patterson, University of Florida
Matthew Weinstein, University of Florida
Anil V. Rao, University of Florida

An object-oriented method is presented that computes without truncation error derivatives of functions defined by MATLAB computer codes. The method implements forward mode automatic differentiation via operator overloading in a manner that produces a new MATLAB code which computes the derivatives of the outputs of the original function with respect to the variables of differentiation. Because the derivative code has the same input as the original function code, the method can be used recursively to generate derivatives of any order that are desired. In addition, the approach developed in this paper has the feature that the derivatives are generated simply by evaluating the function on an instance of the class, thus making the method straightforward to use while simultaneously enabling differentiation of highly complex functions. A detailed description of the method is presented and the approach is illustrated and is shown to be efficient on four examples.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Automatic Differentiation

General Terms: Automatic Differentiation, Numerical Methods, MATLAB.

Additional Key Words and Phrases: Scientific Computation, Applied Mathematics.

ACM Reference Format:
Patterson, M. A., Weinstein, M., and Rao, A. V. 2010. An efficient overloaded method for computing derivatives of mathematical functions in MATLAB. ACM Trans. Math. Soft. 39, 3, Article 1 (July 2013), 39 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Obtaining accurate approximations to derivatives of general functions is important in a variety of subjects including science, engineering, economics, and scientific computation. Derivative approximation techniques fall into three general categories: numerical approximation, symbolic differentiation, and automatic differentiation. Numerical approximation techniques include the classical methods of finite-differencing (e.g., forward and central differencing) and the complex-step derivative approximation [Martins et al. 2003]. In the case of finite-differencing, the derivative is replaced by a computation where the function is evaluated at two neighboring values of the ordinate

The authors gratefully acknowledge support for this research from the U.S. Office of Naval Research (ONR) under Grant N00014-11-1-0068 and from the U.S. Defense Advanced Research Projects Agency (DARPA) Under Contract HR0011-12-0011. The authors would also like to thank the Referees for improving the manuscript through their thoughtful and constructive reviews. **Disclaimer:** The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.
Author's addresses: M. A. Patterson, M. Weinstein, and A. V. Rao, Department of Mechanical and Aerospace Engineering, P.O. Box 116250, University of Florida, Gainesville, FL 32611-6250; e-mail: {mpatterson,mweinstein,anilvrao}@ufl.edu.

and the difference between these function values is divided by the difference in the ordinate values. Great care must be taken in finite-differencing because ordinate values that are too widely spaced lead to an inaccurate derivative while ordinate values that are too closely spaced can result in catastrophic cancellation in the numerator of the approximation. In the *complex-step derivative approximation* [Martins et al. 2003], the perturbation is taken in the imaginary direction in the complex plane. The complex-step derivative approximation has a useful feature that the derivative can be approximated without taking the difference between two values of the function. Instead, by perturbing the ordinate in the imaginary direction, it is possible to make the perturbation size extremely small and obtain a highly accurate derivative approximation. Unlike finite-differencing, the complex-step derivative approximation does not suffer from catastrophic cancellation when extremely small perturbation sizes are used. It is noted that a more general version of the complex-step derivative approximation, called the *multi-complex-step derivative approximation* [Lantoine et al. 2012] has been developed to obtain approximations to higher-order derivatives. Finally, the software *PMAD* [Shampine 2007] is a MATLAB implementation of the complex-step derivative approximation.

A second differentiation approach that is available as part of a more general computer algebra system, is *symbolic differentiation*. Symbolic differentiation is performed using symbolic representations of the actual quantities or variables being manipulated and is essentially a way of programming a computer to perform the steps that a human would perform by hand. Symbolic differentiation is most often implemented by accumulating the function into a single expression and differentiating the expression using defined rules of algebra and calculus. While symbolic differentiation is appealing, it can become computationally difficult for very large problems due to the possible explosion in expression size that can arise when computing the derivatives of a complicated function. Many software programs that implement symbolic differentiation are available today including *MAPLE* [Monagan et al. 2005], *Mathematica* [Wolfram 2008], and the *MATLAB Symbolic Toolbox* [Mathworks 2010].

In the past few decades, a great deal of research has focused on *automatic differentiation* (AD). AD is the process of determining accurate derivatives of a function defined by computer programs [Griewank 2008] using the rules of differential calculus. In other words, the goal of AD is to employ ordinary calculus algorithmically with the goal of obtaining machine precision accuracy derivatives in a computationally efficient manner. AD exploits the fact that a computer code that implements a general function $y = f(x)$ can be decomposed into a sequence of elementary function operations. The derivative is then obtained by applying the standard differentiation rules (e.g., product, quotient, and chain rules).

The most well known methods for automatic differentiation are *forward and reverse mode*. In either forward or reverse mode, each link in the calculus chain rule is implemented until the derivative of the input with respect to itself is encountered. The fundamental difference between forward and reverse modes is the direction in which the operations are performed. In forward mode, the operations are performed from the variable of differentiation to the final derivative of the function, while in reverse mode the operations are performed from the function back to the variable of differentiation.

Forward and reverse mode automatic differentiation are implemented using either operator overloading or source transformation. In an operator-overloaded approach, a custom class is constructed and all standard arithmetic operations and mathematical functions are defined to operate on objects of the class. Furthermore, any object of the custom class typically contains properties that include the value and first- or higher-order derivatives of the object at a particular numerical value of the input. The process of forward mode automatic differentiation begins with an instance of the class (for which the instance of the class represents the identity matrix). The derivatives are then propagated through each overloaded operation and function call, and the derivative of the output with respect to the input is obtained at the numerical value contained in the value property of the instantiated object. In a source transformation approach, the original source code is replaced by a new code such that the new code contains statements that compute the derivatives, and these new lines of code are interleaved with the statements in the original code. Well known implementations of forward and reverse mode that utilize operator overloading include *ADMIT-1* [Coleman and Verma 1998b], *ADMAT* [Coleman and Verma 1998a], *INTLAB* [Rump 1999], *ADOL-C* [Griewank et al. 1996], and *MAD* [Forth 2006], while well known implementations of source transformation include *ADIFOR* [Bischof et al. 1992; Bischof et al. 1996], *PCOMP* [Dobmann et al. 1995], *TAPENADE* [Hascoët and Pascual 2004], *AdiMat* [Bischof et al. 2002], and *MSAD* [Kharche and Forth 2006]. Amongst these previously developed source transformation tools, the programs *AdiMat* [Bischof et al. 2002] and *MSAD* [Kharche and Forth 2006] are designed to return MATLAB code that computes the derivative of the original function [Neidinger 2010]. *AdiMat* utilizes a hybrid source transformation and operator-overloading approach to implement the forward mode [Bischof et al. 2002]. *AdiMat* computes single directional derivatives, however, computing multiple directional derivatives require the use of a MATLAB mex interface [Kharche 2011]. *MSAD* [Kharche and Forth 2006] combines source transformation with with the efficient data structures in *MAD* [Forth 2006], resulting in a hybrid source transformation/operator overloading approach similar to that found in *AdiMat* [Bischof et al. 2002]. Ref. Kharche and Forth [2006] provides a description of *MSAD* and demonstrates the effectiveness of *MSAD* on several test cases.

In this paper we present a new approach for automatic differentiation of MATLAB code. The method of this paper combines features of operator-overloading and source transformation together with forward mode automatic differentiation in a manner that produces a new MATLAB code that contains statements to compute the derivative of the original MATLAB function in terms of the native MATLAB library. In order to realize our approach, a new MATLAB object class, called *CADA* (where *CADA* stands for "Computation of Analytic Derivatives Automatically"), is developed. Using instances of class *CADA*, the function is evaluated on the input *CADA* object. *CADA* then produces a file that contains the statements that compute the derivative of the original function with respect to the input variable. The sequence of mathematical operations in the file that computes the derivative file is identical to those obtained by applying the forward mode on a numeric value of the variable of differentiation.

As stated, the approach developed in this paper differs from previously developed automatic differentiation methods in that it is neither purely a traditional operator overloaded method nor purely a source transformation tool. Instead, the approach combines features of operator overloading and source transformation in that an object-oriented approach is used to generate a new code that contains the derivatives of the original code. The similarity with an operator overloading approach lies in the fact that the method also uses operator overloading. The method of this paper, however, does not use operator overloading in the same manner as is used by other operator overloaded automatic differentiation software. Unlike other operator overloaded automatic differentiation software where the derivative is obtained at a particular numerical value of the input argument, the method of this paper returns a function that contains the derivatives. The similarity with source transformation arises from the fact that the method of this paper returns a new source code that computes the derivative. The approach described in this paper, however, does not scan, interpret or rewrite the original source code prior to generating the derivative code. Instead, the method of this paper *evaluates* the function on an instance of the class and directly produces a derivative code in terms of the native MATLAB library. Finally, it is noted that, unlike a computer algebra system, the method of this paper does not manipulate symbols. Instead, the method of this paper applies the forward mode of automatic differentiation using the syntax of the MATLAB language to return a MATLAB code that contains the derivatives.

The motivation for this paper is as follows. First, in the approach developed in this paper, the overloaded operations need to be performed only once, when the derivative file is created, as opposed to each time a derivative is required at a numerical value of the input. As a result, the overhead associated with the overloaded process is only incurred once, thus improving the efficiency with which the derivatives can be evaluated. Second, derivatives of any order can be obtained by applying the method repeatedly on the previously generated derivative file. This recursive process differs from typical overloaded implementations where generating higher-order derivative requires that either overloaded functions that carry out the rules of calculus for the derivative of a particular order be written (as is the case with *INTLAB* [Rump 1999]) or that the overloaded process be performed in a layered manner to allow for higher-order derivatives (as is the case with *MAD* [Forth 2006]). With regard to using operating overloading to compute derivatives of a particular order, the rules of calculus become increasingly complex, while the process of layered overloading becomes inefficient due to the need to overload the overloading. The approach of this paper attempts to combine the simplicity of needing to compute only a single derivative (and repeating the process as necessary) with the efficiency gained by generating source code that can simply be evaluated. Third, because the method generates MATLAB source code, the derivative code can be ported to another machine without requiring that the *CADA* software exist on the other machine.

This paper is organized as follows. In Section 2 we provide the notation and conventions that will be used throughout the paper. In Section 3 we provide a brief overview of the forward mode of automatic differentiation. In Section 4 we explain the approach to using the *CADA* class to determine analytic derivatives of general functions. In Section 5 we demonstrate the application of *CADA* on four examples. The first example illustrates how *CADA* avoids expression explosion in the derivative on a function whose derivative expression is known to explode using symbolic differentiation. The second example demonstrates the ability of *CADA* to efficiently generate first derivatives along with the efficiency of *CADA* to efficiently evaluate the resulting derivative code when compared with other well developed automatic differentiation tools. In addition, the second example demonstrates how *CADA* generates sparse vectorized derivative code if the original function is vectorized and for which the Jacobian is known to be sparse. The third example demonstrates the ability of *CADA* to efficiently generate the Hessian of a scalar function of a vector, and again provides a comparison between *CADA* and other well known automatic differentiation tools. The fourth example shows the applicability of *CADA* to generate both the constraint Jacobian and Lagrangian Hessian required to solve a large sparse nonlinear programming problem. In Section 6 we provide a discussion of the results obtained in the examples of Section 5. In Section 7 we describe some limitations of our approach. Finally, in Section 8 we provide conclusions on our work.

2. NOTATION AND CONVENTIONS

In this paper we employ the following notation. First, all vector or matrix variables will be denoted by a lower-case bold letter. Thus, if $\mathbf{x} \in \mathbb{R}^{m\times n}$, then

$$\mathbf{x} = \begin{bmatrix} x_{1,1} & \cdots & x_{1,n} \\ x_{2,1} & \cdots & x_{2,n} \\ \vdots & \ddots & \vdots \\ x_{m,1} & \cdots & x_{m,n} \end{bmatrix} \in \mathbb{R}^{m\times n}, \tag{1}$$

where $x_{i,j}$, $(i = 1, \ldots, m)$, $(j = 1, \ldots, n)$ are the *elements* of the $m \times n$ matrix $\mathbf{x}$. Similarly, the output of any function will be denoted by a lower case bold letter. Consequently, if $\mathbf{f} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$ is a matrix function of the matrix variable $\mathbf{x}$, then $\mathbf{f}(\mathbf{x})$ has the form

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_{1,1}(\mathbf{x}) & \cdots & f_{1,q}(\mathbf{x}) \\ f_{2,1}(\mathbf{x}) & \cdots & f_{2,q}(\mathbf{x}) \\ \vdots & \ddots & \vdots \\ f_{p,1}(\mathbf{x}) & \cdots & f_{p,q}(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^{p\times q}, \tag{2}$$

where $f_{k,l}(\mathbf{x})$, $(k = 1, \ldots, p)$, $(l = 1, \ldots, q)$ are the *elements* of the $p \times q$ matrix function $\mathbf{f}(\mathbf{x})$. The *Jacobian* of the matrix function $\mathbf{f}(\mathbf{x})$, denoted $\mathbf{Jf}(\mathbf{x})$, is then a four-dimensional array of size $p \times q \times m \times n$ that consists of $pqmn$ elements. This multi-dimensional array will be referred to generically as the *rolled* representation of the derivative of $\mathbf{f}(\mathbf{x})$ with respect to $\mathbf{x}$ (where the term "rolled" is similar to the term "external" as used in [Forth 2006]). In order to provide a more tractable form for the Jacobian of $\mathbf{f}(\mathbf{x})$, the matrix variable $\mathbf{x}$ and matrix function $\mathbf{f}(\mathbf{x})$ are transformed into the following so called *unrolled* form (where, again, the term "unrolled" is similar to the term "internal" [Forth 2006]). First, $\mathbf{x} \in \mathbb{R}^{m\times n}$, is mapped isomorphically to a column vector $\mathbf{x}^\dagger \in \mathbb{R}^{mn}$,

$$\mathbf{x}^\dagger = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_n \end{bmatrix} = \begin{bmatrix} x_1^\dagger \\ \vdots \\ x_{mn}^\dagger \end{bmatrix}, \quad \mathbf{x}_i = \begin{bmatrix} x_{1,i} \\ \vdots \\ x_{m,i} \end{bmatrix}, \quad (i = 1, \ldots, n), \tag{3}$$

where the single index used for $x_a^\dagger$, $(a = 1, \ldots, mn)$, can be found by the relation $a = i + m(j-1)$. Similarly, let $\mathbf{f}^\dagger(\mathbf{x}^\dagger) \in \mathbb{R}^{pq}$ be the one-dimensional transformation of the function $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^{p\times q}$,

$$\mathbf{f}^\dagger(\mathbf{x}^\dagger) = \begin{bmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_q \end{bmatrix} = \begin{bmatrix} f_1^\dagger(\mathbf{x}^\dagger) \\ \vdots \\ f_{pq}^\dagger(\mathbf{x}^\dagger) \end{bmatrix}, \quad \mathbf{f}_i = \begin{bmatrix} f_{1,i}(\mathbf{x}) \\ \vdots \\ f_{p,i}(\mathbf{x}) \end{bmatrix}, \quad (i = 1, \ldots, q), \tag{4}$$

where the single index used for $f_b^\dagger(\mathbf{x}^\dagger)$, $(b = 1, \ldots, pq)$, can be found by the relation $b = k + p(l-1)$. Using the one-dimensional representations $\mathbf{x}^\dagger$ and $\mathbf{f}^\dagger(\mathbf{x}^\dagger)$, the four-dimensional Jacobian, $\mathbf{Jf}(\mathbf{x})$ can be represented in two-dimensional form as

$$\mathbf{Jf}^\dagger(\mathbf{x}^\dagger) = \begin{bmatrix} \frac{\partial f_1^\dagger}{\partial x_1^\dagger} & \cdots & \frac{\partial f_1^\dagger}{\partial x_{mn}^\dagger} \\ \frac{\partial f_2^\dagger}{\partial x_1^\dagger} & \cdots & \frac{\partial f_2^\dagger}{\partial x_{mn}^\dagger} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{pq}^\dagger}{\partial x_1^\dagger} & \cdots & \frac{\partial f_{pq}^\dagger}{\partial x_{mn}^\dagger} \end{bmatrix} \in \mathbb{R}^{pq \times mn}. \tag{5}$$

Equation (5) provides what is known as the *internal* [Forth 2006] two-dimensional representation of the derivative. It is noted that $\mathbf{Jf}^\dagger(\mathbf{x}^\dagger) \in \mathbb{R}^{pq\times mn}$ in Eq. (5) contains the same information as the four-dimensional array $\mathbf{Jf}(\mathbf{x}) \in \mathbb{R}^{p\times q\times m\times n}$. Once a Jacobian of a function has been determined in the aforementioned manner, higher-order derivatives can be computed by repeating the process on $\mathbf{Jf}^\dagger(\mathbf{x}^\dagger)$ [Padulo et al. 2008]. As an example, starting with $\mathbf{x}^\dagger$ and $\mathbf{Jf}^\dagger(\mathbf{x}^\dagger)$, the Jacobian of $\mathbf{Jf}^\dagger$, denoted $\mathbf{J}^2\mathbf{f}^\dagger(\mathbf{x}^\dagger)$, can be computed by transforming $\mathbf{Jf}^\dagger$ to a one-dimensional array in the manner given in Eq. (4) and computing $\mathbf{J}^2\mathbf{f}^\dagger(\mathbf{x}^\dagger)$ in a form similar to that given in Eq. (5). The resulting matrix $\mathbf{J}^2\mathbf{f}^\dagger(\mathbf{x}^\dagger)$ would then be of size $pqmn \times mn$. Thus, when starting from a matrix function of a matrix, any order derivative is represented as a two-dimensional array. With regard to implementation on a computer, it is much more efficient to isomorphically map all variables and functions from higher-dimensional quantities to one-dimensional quantities because it provides a more systematic way to reference and store the derivatives. Finally, it is noted that the method described in this paper utilizes MATLAB object-oriented programming with operator overloading. In order to refer to a MATLAB object that corresponds to a multi-dimensional array $\mathbf{y}$, we use a corresponding calligraphic character, $\mathcal{Y}$. In other words, if $\mathbf{y}$ is a vector or matrix, then $\mathcal{Y}$ is a MATLAB object that corresponds to $\mathbf{y}$.

3. FORWARD MODE AUTOMATIC DIFFERENTIATION

Using the conventions developed in Section 2, consider a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$. Assume that $\mathbf{f}(\mathbf{x})$ can be reduced to a sequence of elementary function operations (e.g., polynomials, exponential functions, trigonometric functions). Furthermore, suppose that a computer implementation of the $\mathbf{f}(\mathbf{x})$ has been written in the MATLAB programming language. The goal is to automatically determine $\mathbf{Jf}(\mathbf{x})$, that is, the goal is to determine the derivative of each output function $f_{ij},\ (i = 1,\ldots,p),\ (j = 1\ldots,q)$ with respect to each input element $x_{kl},\ (k = 1,\ldots,m),\ (l = 1,\ldots,n)$. Moreover, it is desired to obtain a new MATLAB code that computes these derivatives for future use. The method of this paper uses an object-oriented approach that implements forward mode automatic differentiation in such a way that each operation in a function is differentiated in terms of the results of previous operations, and the result of each operation is printed to a file. In forward mode differentiation, the chain rule is used repeatedly until the step in the chain is encountered where the derivative of the outermost function with respect to the input is obtained. In the case of a scalar function $y = f(g(x))$, the derivative $\partial f/\partial x$ is obtained via the chain rule as

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}. \tag{6}$$

Finally, as part of the method developed in this paper, only the nonzero elements of the Jacobian are stored, reducing memory requirements and taking advantage of Jacobian sparsity [Forth et al. 2004].

4. MAJOR COMPONENTS OF METHOD

In this section we describe many of the major components of the method developed in this paper. Consider again a function of the form $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$. Next, suppose that a MATLAB class called *CADA* has been developed, where any *CADA* instance, $\mathcal{X}$, corresponds to a two-dimensional real mathematical variable $\mathbf{x} \in \mathbb{R}^{n\times m}$. Furthermore, assume that overloaded versions of all standard unary mathematical functions (e.g., polynomial, trigonometric, exponential, etc.), binary mathematical functions (e.g., plus, minus, times, etc.) and organizational functions (e.g., reshape, transpose, matrix, replication, and tiling, etc.) have been written specifically to operate on *CADA* objects.

Assume now the user has written a MATLAB code that implements a function $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where $\mathbf{f} : \mathbb{R}^{m\times n} \longrightarrow \mathbb{R}^{p\times q}$. Assume further that this code can be decomposed into a sequence of the aforementioned available overloaded mathematical and organizational functions. *CADA* is designed to write a new MATLAB code that can compute the analytic derivative, $\partial f_{ij}/\partial x_{kl},\ (i = 1,\ldots,p),\ (j = 1,\ldots,q),\ (k = 1,\ldots,m),\ (l = 1,\ldots,n)$, of each function output with respect to each input. Using an object-oriented approach, this new function is obtained by *evaluating* the function on the input *CADA* instance $\mathcal{X}$. When any overloaded function operates on a *CADA* object, symbolic representations of the result of the overloaded function and the derivative are printed to a file in terms of the input *CADA* object properties. The output of the overloaded function is itself another *CADA* object with the function and derivative properties redefined to represent the result of the overloaded function. The evaluation of any function is performed in the usual mathematical manner and derivatives are computed using the well known rules of differential calculus for each overloaded function. When evaluating the resulting derivative code, the output derivative is obtained by applying the rules of differential calculus (e.g., sum, product, quotient, and chain rules) until all rules have been completely applied. In the sections that follow, the major components of *CADA* are described along with capabilities that maintain functionality with general MATLAB code and the ability to generate gradient, Jacobian, and Hessian files for use in other applications.

4.1. The *CADA* Environment

The method of this paper requires that an environment within MATLAB be created to achieve the goal of generating efficient derivative code. The *CADA* environment consist of three key components, a temporary file to which the derivative code will be written, a global variable that contains information that is required for each step in the function evaluation process and the overloaded *CADA* objects that are being evaluated. All global information is stored in a global structure named GLOBALCADA. The structure GLOBALCADA has the following fields:

(i) **OPERATIONCOUNT:** a positive integer that represents the current overloaded operation count. Each time an overloaded *CADA* operation is evaluated, this value is incremented by unity. The value of **OPERATIONCOUNT** is used to create a unique handle for the resulting *CADA* object for each overloaded operation.

(ii) **LINECOUNT:** a positive integer that represents the current number of function lines in the *CADA* temporary file. Whenever a line of code is written to the temporary file, the value of **LINECOUNT** is incremented by unity. The counter **LINECOUNT** is used to keep track of the location in the temporary file where lines of code are being written.

(iii) **FUNCTIONLOCATION:** an $N \times 2$ MATLAB array, where N is the number of overloaded operations that have been evaluated. Elements $(i, 1)$ and $(i, 2)$ of **FUNCTIONLOCATION** contains the starting and ending lines in the temporarily file that correspond to the i^{th} overloaded operation that has been executed. The information in **FUNCTIONLOCATION** is used during the post-processing on the temporary file in order to determine when a function handle can be re-used (see Section 4.8)

(iv) **NUMBERVARIABLES:** a positive integer that represents the number of *CADA* instances in the current environment. The property **NUMBERVARIABLES** is defined when the *CADA* environment is instantiated and never changes during a *CADA* session.

(v) **VARIABLE:** a MATLAB cell array of size $\mathbf{NUMBERVARIABLES} \times 3$. The $\{i, 1\}$, $(i = 1, \ldots, \mathbf{NUMBERVARIABLES})$ element in **VARIABLE** contains the string name of the i^{th} *CADA* instance, while the $\{i, 2\}$, $(i = 1, \ldots, \mathbf{NUMBERVARIABLES})$ element in **VARIABLE** contains a 1×2 numeric array that represents the size of the i^{th} *CADA* instance. The cell array element $\{i, 3\}$, $(i = 1, \ldots, \mathbf{NUMBERVARIABLES})$ contains either unity or zero, where a unity indicates that the derivatives with respect to the i^{th} *CADA* instance are being calculated, while a zero indicates that the derivatives with respect to the i^{th} *CADA* instance are not being calculated. The property **VARIABLE** is defined when the *CADA* environment is instantiated and never changes during a *CADA* session.

(vi) **FID:** An integer that identifies the temporary file to which the derivative code will be written during a *CADA* session. The property **FID** is defined when the *CADA* environment is instantiated and never changes during a *CADA* session.

4.2. *CADA* Class and *CADA* Object Properties

As mentioned previously, the method of this paper is developed via a new overloaded MATLAB object class called *CADA*. Every *CADA* object has the following properties:

(i) **function:** a MATLAB cell array of size 1×3, where element $\{1, 1\}$ contains a string representing the current function handle, element $\{1, 2\}$ contains a 1×2 numeric array that represents the row and column size of the current function. The cell array element $\{1, 3\}$ contains a integer that represents the current function count. If the *CADA* object is representing a fixed numerical array then element $\{1, 1\}$ contains the numeric array.

(ii) **derivative:** an $\mathbf{NUMBERVARIABLES} \times 2$ MATLAB cell array, where $\mathbf{NUMBERVARIABLES}$ is the number of *CADA* instances in the current *CADA* environment. The cell array element $\{i, 1\}$, $(i = 1, \ldots, \mathbf{NUMBERVARIABLES})$, of the **derivative** cell array contains a string representing the current derivative handle with respect to the i^{th} *CADA* instance. The cell array element $\{i, 2\}$, $(i = 1, \ldots, \mathbf{NUMBERVARIABLES})$, contains a numeric array of size $n_z \times 2$, where n_z is the number of nonzero derivatives. The indices in this numeric array contain the locations of the nonzero derivatives. More specifically, the first and second elements in each row of this numeric array contain the function element and variable element indices, respectively. If the *CADA* object is representing a fixed numerical array then all cells are empty.

Any *CADA* object that represents a fixed numeric array must result from overloaded operations whose output is numeric. Examples of such operations include size, ones, and zeros. The derivatives of such operations are zero. Such *CADA* objects are referred to as *CADA* numeric objects.

4.3. *CADA* Differentiation Method

The method used in *CADA* to differentiate a function by evaluating the function on overloaded *CADA* objects relies upon the interaction between the different *CADA* environment properties. When any mathematical operation is evaluated on a *CADA* object, the lines of code that represent the mathematical operation and the derivative of the mathematical operation are printed to a temporary file. As an example of how an overloaded operation is written to the temporary file, consider a mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and assume that this mathematical operation is applied to a *CADA* object $\mathcal{X}$, resulting in an output *CADA* object $\mathcal{Z}$ (that is, $\mathcal{Z} = \mathbf{f}(\mathcal{X})$). Fig. 1 shows the operational flow that produces the derivative code for the operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$, where it is seen that information from both the object properties of $\mathcal{X}$ and the GLOBALCADA properties are required to compute the derivative of the operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$. While the object properties of $\mathcal{X}$ contain information from previous overloaded evaluations up to that point in the overall function, the GLOBALCADA information is needed to define the function and derivative handles of the output $\mathcal{Z}$. First, a check is performed to determine if the input $\mathcal{X}$ is a *CADA* numeric object. In this case the **function** property of $\mathcal{Z}$ is calculated directly and no code is written to the temporary file. If $\mathcal{X}$ is not a *CADA* numeric object, new function and derivative handles are defined for $\mathcal{Z}$ and the code that represents $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and the derivative of $\mathbf{z} = \mathbf{f}(\mathbf{x})$ are printed to the temporary file. In this second case, only the code that corresponds to the *nonzero* derivatives of the operation with respect to each *CADA* instance is written to the temporary file. Because only the nonzero derivatives are written to the temporary file, the method produces a sparse representation of the derivative of the function. The details on how the *CADA* method provides a sparse representation of the derivative is given below.

In order to ensure that a sparse representation of the derivative is written to the temporary file, it is necessary that the appropriate elements of the **function** property of $\mathcal{X}$ are properly referenced such that the code resulting from the application of the calculus chain rule is applied to only the elements in the array whose derivatives are not zero. The GLOBALCADA properties are also updated during an overloaded operation in the following manner. For every line of code that is written to the temporary file during an overloaded operation, the property **LINECOUNT** is increased by unity. The locations of the first and last lines of code that represent the function $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and the corresponding derivative are added to the **FUNCTIONLOCATION** property. Furthermore, the location of the last line of code in terms of the function handle representing $\mathcal{X}$ is updated. It is noted that the subroutine **cadaindprint** is used to efficiently print the derivative indices to the temporary file.

4.4. *CADA* Setup and Environment Instantiation

The *CADA* global environment and all *CADA* instances that are used in generating derivative code are instantiated using the function **cadasetup**. The **cadasetup** function performs the following operations: (1) instantiates the global structure GLOBALCADA; (2) defines the initial property values for GLOBALCADA; (3) creates and opens a temporary file named "**cada.$#$#$temp@derivfile.m**" to which the derivative code will be written; (4) and creates the initial *CADA* instances that will be used during the *CADA* session. For each initial *CADA* instance that is desired, **cadasetup** requires the following four inputs: a string containing the name of the *CADA* instance (where each instance must have a unique name and the name cannot be a MATLAB reserved word); two integers that, respectively, define the row and column size of the *CADA* instance; and an integer flag that indicates whether or not *CADA* should compute derivatives with respect to the particular *CADA* instance. If the derivative option flag corresponding to a *CADA* instance is unity, then derivatives with respect to the

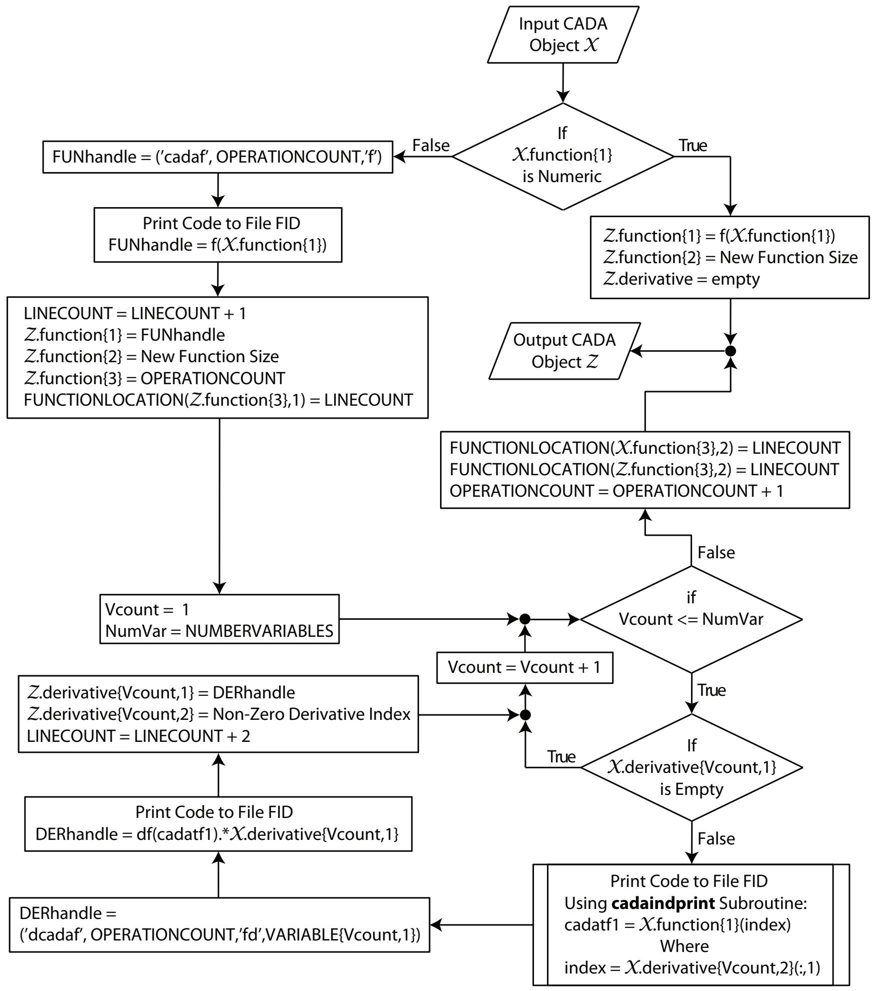

Input CADA Object $\mathcal{X}$
If $\mathcal{X}$.function{1} is Numeric
False
True
FUNhandle = ('cadaf', OPERATIONCOUNT,'f')
Print Code to File FID FUNhandle = f($\mathcal{X}$.function{1})
LINECOUNT = LINECOUNT + 1
$\mathcal{Z}$.function{1} = FUNhandle
$\mathcal{Z}$.function{2} = New Function Size
$\mathcal{Z}$.function{3} = OPERATIONCOUNT
FUNCTIONLOCATION($\mathcal{Z}$.function{3},1) = LINECOUNT
$\mathcal{Z}$.function{1} = f($\mathcal{X}$.function{1})
$\mathcal{Z}$.function{2} = New Function Size
$\mathcal{Z}$.derivative = empty
Output CADA Object $\mathcal{Z}$
FUNCTIONLOCATION($\mathcal{X}$.function{3},2) = LINECOUNT
FUNCTIONLOCATION($\mathcal{Z}$.function{3},2) = LINECOUNT
OPERATIONCOUNT = OPERATIONCOUNT + 1
False
Vcount = 1
NumVar = NUMBERVARIABLES
if Vcount <= NumVar
Vcount = Vcount + 1
True
$\mathcal{Z}$.derivative{Vcount,1} = DERhandle
$\mathcal{Z}$.derivative{Vcount,2} = Non-Zero Derivative Index
LINECOUNT = LINECOUNT + 2
True
If $\mathcal{X}$.derivative{Vcount,1} is Empty
Print Code to File FID DERhandle = df(cadatf1).*$\mathcal{X}$.derivative{Vcount,1}
False
DERhandle = ('dcadaf', OPERATIONCOUNT,'fd',VARIABLE{Vcount,1})
Print Code to File FID
Using **cadaindprint** Subroutine:
cadatf1 = $\mathcal{X}$.function{1}(index)
Where
index = $\mathcal{X}$.derivative{Vcount,2}(:,1)

Fig. 1: Flowchart Describing How an Overloaded Operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$ Produces Derivative Code.

CADA instance are calculated. On the other hand, if the derivative option flag corresponding to a *CADA* instance is zero, derivatives with respect to the *CADA* instance are not calculated. The call to the function **cadasetup** is shown in Fig. 2.

Example of cadasetup Usage. Consider instantiating a *CADA* session with the three *CADA* instances $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$, where $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ represent, respectively, the variables $\mathbf{x} \in \mathbb{R}^{3\times1}$, $\mathbf{y} \in \mathbb{R}^{3\times1}$, and $\mathbf{z} \in \mathbb{R}$. Furthermore, suppose that it is desired to obtain derivatives with respect to **X** and **Y**, but it is *not* desired to compute derivatives with respect to **z**. Then a MATLAB code that instantiates the *CADA* session is given as follows:

```
[x y z] = cadasetup('x',3,1,1,'y',3,1,1,'z',1,1,0);
```

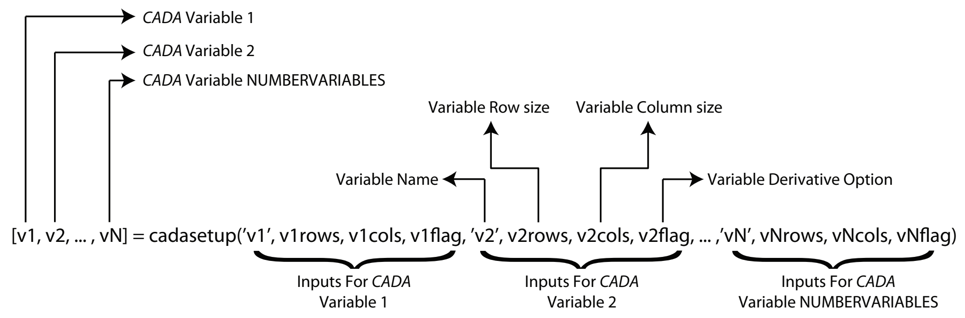

CADA Variable 1
CADA Variable 2
CADA Variable NUMBERVARIABLES
Variable Row size
Variable Column size
Variable Name
Variable Derivative Option
[v1, v2, ... , vN] = cadasetup('v1', v1rows, v1cols, v1flag, 'v2', v2rows, v2cols, v2flag, ... ,'vN', vNrows, vNcols, vNflag)
Inputs For *CADA* Variable 1
Inputs For *CADA* Variable 2
Inputs For *CADA* Variable NUMBERVARIABLES

Fig. 2: Input/Output Scheme of the *CADA* Environment Constructor Function.

The properties of GLOBALCADA after instantiation of the *CADA* environment are given as

```
GLOBALCADA =

      OPERATIONCOUNT: 1
           LINECOUNT: 0
    FUNCTIONLOCATION: [0 0]
     NUMBERVARIABLES: 3
            VARIABLE: {3x3 cell}
                 FID: 3
```

More details of the **VARIABLE** property of GLOBALCADA are given as

```
GLOBALCADA.VARIABLE =

    'x'    [3 1]    [1]
    'y'    [3 1]    [1]
    'z'    [1 1]    [0]
```

It is noted that the GLOBALCADA properties **NUMBERVARIABLES**, **VARIABLE**, and **FID** never change from the instantiated values during the *CADA* session. Furthermore, the object properties of the *CADA* instance will be shown in detail, the object properties of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ are given, respectively, as

```
x.function =

    'x'    [3 1]    []

x.derivative =

    'cadadxdx'    [3x2 double]
            []              []
            []              []

x.derivative{1,2} =

     1     1
     2     2
     3     3

y.function =

    'y'    [3 1]    []

y.derivative =

            []              []
    'cadadydy'    [3x2 double]
            []              []

y.derivative{2,2} =

     1     1
     2     2
     3     3

z.function =

    'z'    [1 1]    []

z.derivative =

    []    []
    []    []
    []    []
```

Because in this example it was not desired to compute derivatives with respect to $\mathcal{Z}$, the **derivative** property of $\mathcal{Z}$ is instantiated as empty (thus implying that derivatives with respect to $\mathcal{Z}$ will *not* be computed). For ease of use, an overloaded **display** function has been implemented for *CADA* objects, the **display** shows pertinent information from the *CADA* object properties in a compact manner. From this point forward in the paper, all *CADA* object details will be shown using the overloaded **display** function. Details of the *CADA* instances $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ are given as follows:

```
>> x
    Current CADA handle:
       'x'     size: 3 x 1
    Current CADA derivative handles with respect to each variable:
       'cadadxdx' has 3 nonzero derivatives with respect to 'x'

>> y
    Current CADA handle:
       'y'     size: 3 x 1
    Current CADA derivative handles with respect to each variable:
       'cadadydy' has 3 nonzero derivatives with respect to 'y'

>> z
    Current CADA handle:
       'z'     size: 1 x 1
```

Because $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ represent *CADA* instances, the indices of the nonzero derivative locations correspond to those of the identity matrix. In other words, the derivative of any of these *CADA* instances with respect to itself is the identity function. Again, it is re-emphasized that these indices correspond to the locations of only the *nonzero* derivatives. Moreover, as the derivative is propagated using the forward mode, at each step only the nonzero derivatives and their locations are computed. Consequently, the derivative is represented sparsely at each step in the evaluation of the function on the *CADA* object.

4.5. Differentiation of Unary Mathematical Functions

The overloaded versions of all standard unary mathematical functions (e.g., polynomial, trigonometric, exponential) operate on only a single *CADA* input object, resulting in an output that is a new *CADA* object. Consider an arbitrary unary mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x})$ and assume that this mathematical operation is applied to a *CADA* object $\mathcal{X}$, resulting in an output *CADA* object $\mathcal{Z}$ (that is, $\mathcal{Z} = \mathbf{f}(\mathcal{X})$). Fig. 3 shows the operational flow that produces the derivative code for the unary operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$. If the input *is not* a *CADA* numeric object, new code that represents the function and its derivative with respect to each *CADA* instance is written to the temporary file. The nonzero derivative indices of the output will be the same as the nonzero derivative indices of the input. If the input to the overloaded unary operation is a *CADA* numeric object, the output of the overloaded unary operation is also a *CADA* numeric object. In this case the numeric array in the *CADA* numeric object is evaluated and neither are derivatives calculated nor is code written to the temporary file.

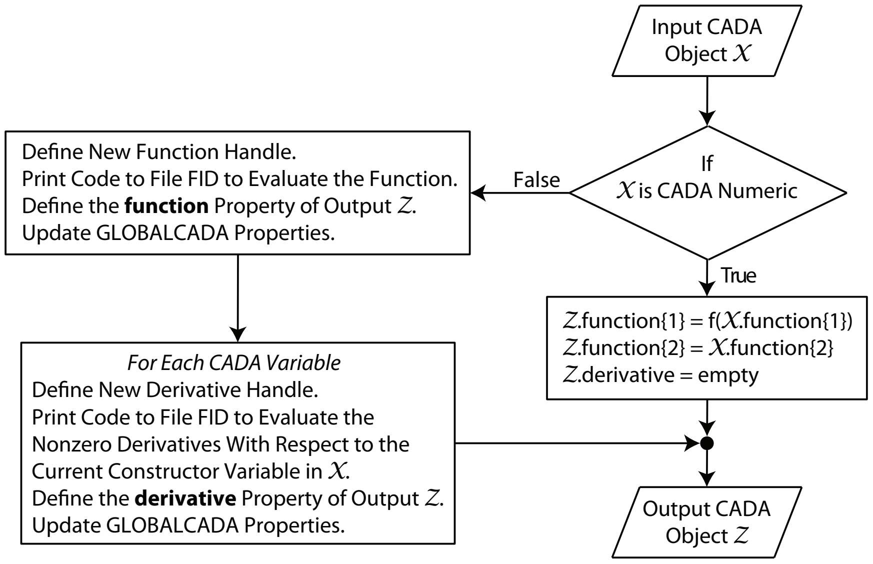

Input CADA Object $\mathcal{X}$
If $\mathcal{X}$ is CADA Numeric
False
True
Define New Function Handle.
Print Code to File FID to Evaluate the Function.
Define the **function** Property of Output $\mathcal{Z}$.
Update GLOBALCADA Properties.
$\mathcal{Z}$.function{1} = f($\mathcal{X}$.function{1})
$\mathcal{Z}$.function{2} = $\mathcal{X}$.function{2}
$\mathcal{Z}$.derivative = empty
For Each CADA Variable
Define New Derivative Handle.
Print Code to File FID to Evaluate the Nonzero Derivatives With Respect to the Current Constructor Variable in $\mathcal{X}$.
Define the **derivative** Property of Output $\mathcal{Z}$.
Update GLOBALCADA Properties.
Output CADA Object $\mathcal{Z}$

Fig. 3: Flowchart Describing How an Overloaded Unary Operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$ Produces Derivative Code.

Example of Unary Mathematical Function. Consider evaluating the unary mathematical function $\mathbf{a} = \sin(\mathbf{x})$ on the *CADA* object $\mathcal{X}$ from the example in Section 4.4. The MATLAB code to evaluate $\mathcal{A} = \sin(\mathcal{X})$ is as follows.

```
>> a = sin(x)
    Current CADA handle:
       'cadaf1f'     size: 3 x 1
    Current CADA derivative handles with respect to each variable:
       'dcadaf1fdx' has 3 nonzero derivatives with respect to 'x'
```

The temporary file then contains the code that calculates both the function $\mathbf{a} = \sin(\mathbf{x})$ and the nonzero derivatives of $\mathbf{a} = \sin(\mathbf{x})$ as seen in the following lines of code:

```
cadaf1f = sin(x);
cadatf1 = x(1:3);
dcadaf1fdx = cos(cadatf1(:)).*cadadxdx;
```

In the above code segment the quantity 'cadadxdx' is the derivative of the *CADA* instance $\mathcal{X}$ and will be defined in a post-processing step that manipulates the temporary file into a executable MATLAB function (see Section 4.8 below). In order to ensure that only the nonzero derivatives are computed, the proper elements of x need to be referenced using the nonzero derivative indices in $\mathcal{X}$ in the line 'cadatf1 = x(1:3)'. The chain rule for the nonzero elements of a is applied using array multiplication in the line 'dcadaf1fdx = cos(cadatf1(:)).*cadadxdx;'. The properties of GLOBALCADA after the evaluation of $\mathcal{A} = \sin(\mathcal{X})$ are as follows:

```
GLOBALCADA =

      OPERATIONCOUNT: 2
           LINECOUNT: 3
    FUNCTIONLOCATION: [1 3]
     NUMBERVARIABLES: 3
```

```
          VARIABLE: {3x3 cell}
               FID: 3
```

In the above code segment, the property **OPERATIONCOUNT** has been increased by one, the property **LINECOUNT** is equal to the number of lines of code in the temporary file, and the property **FUNCTIONLOCATION** now shows that the *CADA* function handle 'cadaf1f' is used in lines 1 through 3 of the temporary file.

4.6. Differentiation of Binary Mathematical Functions

The overloaded binary mathematical functions (e.g., plus, minus, times, array multiplication) have two inputs, where at least one of the inputs must be a *CADA* object and the output is a *CADA* object. Consider a binary mathematical operation $\mathbf{z} = \mathbf{f}(\mathbf{x}, \mathbf{y})$. If $\mathbf{x}$ is a *CADA* object but $\mathbf{y}$ is not a *CADA* object, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathcal{X}, \mathbf{y})$. Similarly, if $\mathbf{y}$ is a *CADA* object but $\mathbf{x}$ is not a *CADA* object, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathbf{x}, \mathcal{Y})$. Finally, if both $\mathbf{x}$ and $\mathbf{y}$ are *CADA* objects, then the function is evaluated as $\mathcal{Z} = \mathbf{f}(\mathcal{X}, \mathcal{Y})$. Fig. 4 shows the operational flow that produces the derivative code for the binary operation $\mathcal{Z} = \mathbf{f}(\mathbf{x}, \mathbf{y})$. First, it is necessary to determine the class of each input. If only one of the inputs is a *CADA* object, then the operation follows in a manner similar to that of a unary function. In the case where both inputs are *CADA* objects, it is necessary to determine if any or both inputs are *CADA* numeric objects. If both inputs are *CADA* numeric objects, the function is evaluated on the numeric values of $\mathcal{X}$ and $\mathcal{Y}$, resulting in the *CADA* numeric object $\mathcal{Z}$. Furthermore, as is the case with all *CADA* numeric objects, no derivatives are calculated and no code is written to the temporary file. If exactly one of the two inputs is a *CADA* numeric object, the numeric value of the *CADA* numeric object is found and the overloaded operation is performed as if only one of the inputs is a *CADA* object. Lastly, if both *CADA* objects contain derivatives with respect to the same *CADA* instance, the binary chain rule (e.g., product rule, quotient rule, etc.) is applied to calculate the nonzero derivatives. The nonzero derivative indices of the output are determined by taking the union of the nonzero derivative indices of $\mathcal{X}$ and $\mathcal{Y}$.

Example of Binary Mathematical Functions. Let $\mathbf{b} = \mathbf{a} \otimes \mathbf{y}$ represent element-by-element multiplication of $\mathbf{a}$ and $\mathbf{y}$. Furthermore, let $\mathcal{A}$ be the *CADA* object from the example of Section 4.5 and let $\mathcal{Y}$ be the *CADA* object from the example of Section 4.4. The MATLAB code that evaluates $\mathcal{B} = \mathcal{A} \otimes \mathcal{Y}$ is as follows:

```
>> b = a.*y
    Current CADA handle:
       'cadaf2f'     size: 3 x 1
    Current CADA derivative handles with respect to each variable:
       'dcadaf2fdx' has 3 nonzero derivatives with respect to 'x'
       'dcadaf2fdy' has 3 nonzero derivatives with respect to 'y'
```

The temporary file now contains the code that calculates the function and the nonzero derivatives of $\mathbf{b} = \mathbf{a} \otimes \mathbf{y}$ and is given as

```
cadaf1f = sin(x);
cadatf1 = x(1:3);
dcadaf1fdx = cos(cadatf1(:)).*cadadxdx;
cadaf2f = cadaf1f .* y;
cadatf1 = y(1:3);
dcadaf2fdx = cadatf1(:).*dcadaf1fdx;
cadatf1 = cadaf1f(1:3);
dcadaf2fdy = cadatf1(:).*cadadydy;
```

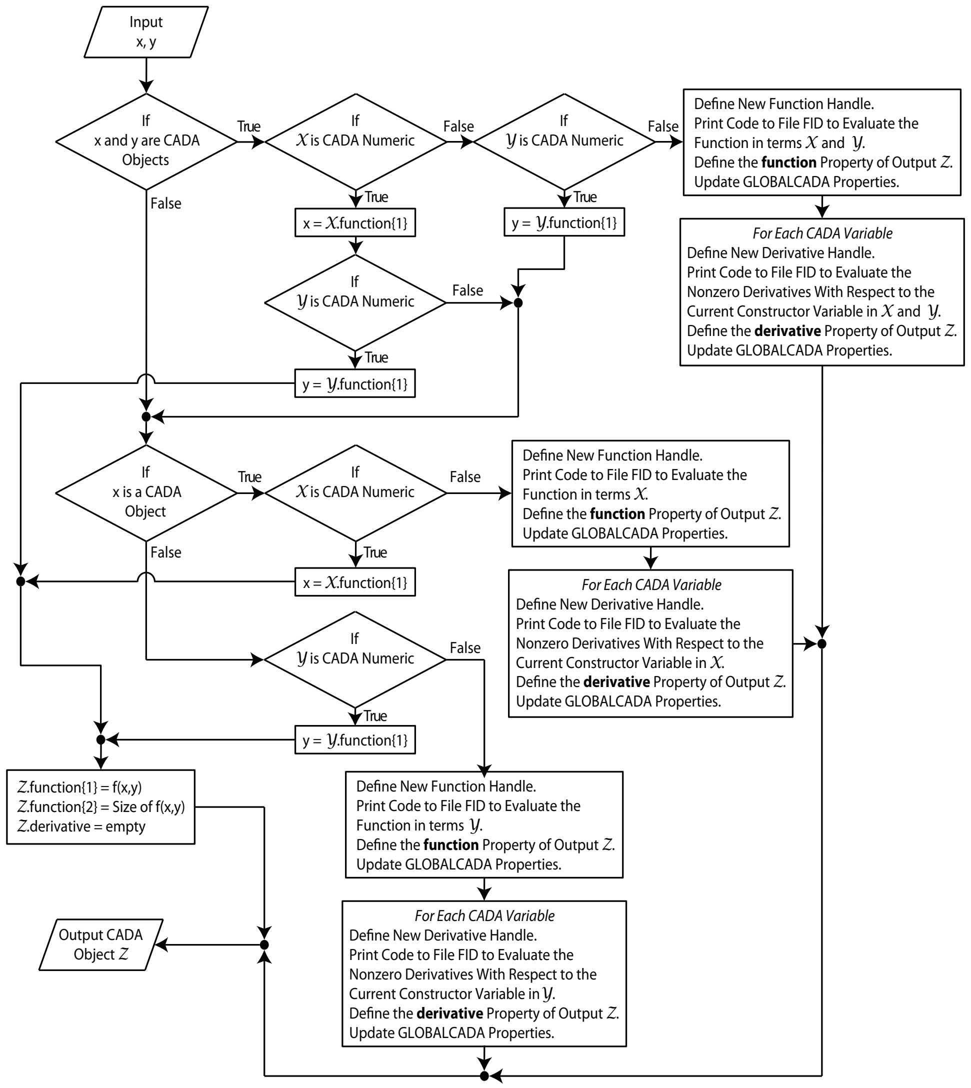

Input x, y
If x and y are CADA Objects
True
False
If $\mathcal{X}$ is CADA Numeric
If $\mathcal{Y}$ is CADA Numeric
Define New Function Handle. Print Code to File FID to Evaluate the Function in terms $\mathcal{X}$ and $\mathcal{Y}$. Define the **function** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
For Each CADA Variable Define New Derivative Handle. Print Code to File FID to Evaluate the Nonzero Derivatives With Respect to the Current Constructor Variable in $\mathcal{X}$ and $\mathcal{Y}$. Define the **derivative** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
x = $\mathcal{X}$.function{1}
y = $\mathcal{Y}$.function{1}
If x is a CADA Object
Define New Function Handle. Print Code to File FID to Evaluate the Function in terms $\mathcal{X}$. Define the **function** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
For Each CADA Variable Define New Derivative Handle. Print Code to File FID to Evaluate the Nonzero Derivatives With Respect to the Current Constructor Variable in $\mathcal{X}$. Define the **derivative** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
Define New Function Handle. Print Code to File FID to Evaluate the Function in terms $\mathcal{Y}$. Define the **function** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
For Each CADA Variable Define New Derivative Handle. Print Code to File FID to Evaluate the Nonzero Derivatives With Respect to the Current Constructor Variable in $\mathcal{Y}$. Define the **derivative** Property of Output $\mathcal{Z}$. Update GLOBALCADA Properties.
$\mathcal{Z}$.function{1} = f(x,y)
$\mathcal{Z}$.function{2} = Size of f(x,y)
$\mathcal{Z}$.derivative = empty
Output CADA Object $\mathcal{Z}$

Fig. 4: Flowchart Describing How an Overloaded binary Operation $\mathcal{Z} = \mathbf{f}(\mathcal{X})$ Produces Derivative Code.

In the aforementioned code segment the quantity ’cadadydy’ is the $\mathcal{Y}$ with respect to the *CADA* instance and will be defined in a post-processing step that transforms the temporary file into a executable MATLAB function (see Section 4.8 below). When computing the nonzero derivatives of b with respect to x, the proper elements of y are referenced using the nonzero derivative indices in $\mathcal{A}$ as shown in the line ’cadatf1 = y(1:3)’. The chain rule to obtain the nonzero derivatives with respect to x is applied via array multiplication in the line ’dcadaf2fdx = cadatf1(:).*dcadaf1fdx’. When computing the nonzero derivatives of b with respect to y, the proper elements of **A** are referenced using the nonzero derivative index in $\mathcal{Y}$ on line ’cadatf1 = cadaf1f(1:3)’. The chain rule to obtain the nonzero derivatives respect to y is then applied via array multiplication in the line 'cadatf1 = cadaf1f(1:3)'. The properties of GLOBALCADA after the evaluation of $\mathcal{B} = \mathcal{A} \otimes \mathcal{Y}$ are as follows:

```
GLOBALCADA =

      OPERATIONCOUNT: 3
           LINECOUNT: 8
    FUNCTIONLOCATION: [2x2 double]
     NUMBERVARIABLES: 3
            VARIABLE: {3x3 cell}
                 FID: 3
```

In the aforementioned code segment, the property **OPERATIONCOUNT** has been increased by one, and the property **LINECOUNT** is equal to the number of lines of code in the temporary file. The contents of the **FUNCTIONLOCATION** is shown as

```
GLOBALCADA.FUNCTIONLOCATION =

     1     8
     4     8
```

The quantity **FUNCTIONLOCATION** now shows that the *CADA* function handle 'cadaf1f' is used in lines 1 through 8, and the *CADA* function handle 'cadaf2f' is used in lines 4 through 8 of the temporary file.

4.7. Array Referencing, Assignment, and Concatenation on *CADA* Objects

The operations of referencing, assignment, and concatenation have the unifying feature that none of them performs any mathematical calculations on array elements. Instead, these operations only alter the locations of these elements in an array. Referencing the elements of an array creates an output that consist of a subset of the original array. Similarly, assigning elements into an array results in a new array that includes elements in the assigned locations. Finally, concatenation merges multiple arrays into a new array. The overloaded referencing, assignment, and concatenation operations for *CADA* generate code that performs in the same standard way as the standard non-overloaded versions of these functions. The overloaded *CADA* versions of these operations, however, keeps track of the derivatives of the elements that have been referenced, assigned, or concatenated, thus, preserving the nonzero derivatives of every element with respect to the *CADA* instance.

Example of Concatenation. Consider the operation of vertical concatenation,

$$\mathbf{c} = \begin{bmatrix} \mathbf{z} \\ \mathbf{b} \end{bmatrix}. \tag{7}$$

In this example the vertical concatenation operation is applied to the *CADA* object $\mathcal{Z}$ from the example in Section 4.4 and the *CADA* object $\mathcal{B}$ from the example in Section 4.6. The MATLAB code that performs the vertical concatenation given in Eq. (7) on $\mathcal{Z}$ and $\mathcal{B}$ is given as

```
>> c = [z; b]
    Current CADA handle:
       'cadaf3f'     size: 4 x 1
    Current CADA derivative handles with respect to each variable:
       'dcadaf3fdx' has 3 nonzero derivatives with respect to 'x'
       'dcadaf3fdy' has 3 nonzero derivatives with respect to 'y'
```

The result of the aforementioned code segment is that the temporary file contains the code that performs the vertical concatenation shown in Eq. (7). The nonzero derivatives of c with respect to x and y are then found by defining new indices for the nonzero derivatives of b as follows:

```
cadaf1f = sin(x);
cadatf1 = x(1:3);
dcadaf1fdx = cos(cadatf1(:)).*cadadxdx;
cadaf2f = cadaf1f .* y;
cadatf1 = y(1:3);
dcadaf2fdx = cadatf1(:).*dcadaf1fdx;
cadatf1 = cadaf1f((1:3));
dcadaf2fdy = cadatf1(:).*cadadydy;
cadaf3f = vertcat(z,cadaf2f);
dcadaf3fdx = zeros(3,cadaunit);
dcadaf3fdx((1:3)) = dcadaf2fdx;
dcadaf3fdy = zeros(3,cadaunit);
dcadaf3fdy((1:3)) = dcadaf2fdy;
```

The derivative indices corresponding to the nonzero derivatives in $\mathcal{C}$ are found in the **derivative** property of $\mathcal{C}$ as

```
c.derivative{1,1} =

dcadaf3fdx

c.derivative{1,2} =

     2     1
     3     2
     4     3

c.derivative{2,1} =

dcadaf3fdy

c.derivative{2,2} =

     2     1
     3     2
     4     3
```

It is seen the this last code segment that the derivative indices of $\mathcal{C}$ have been altered based on the vertically concatenated objects $\mathcal{Z}$ and $\mathcal{B}$.

4.8. Post-Processing of Temporary File to Create Derivative Function

Once the desired function evaluations have been completed within the *CADA* environment, the resulting temporary file is post-processed in order to obtain the derivative function file for use with MATLAB. The post-processing of the temporary file is divided into the following two parts: (1) reassignment of function and derivative function handle names that were created when the temporary file was written and (2) formatting the nonzero derivative values and their corresponding indices into the desired output. Each of these steps is now described, along with a rationale for why it is important to reassign function handles as alluded to in step (1).

4.8.1. Re-Numbering of Function and Derivative Handles in Final Derivative Code. In the process of performing this research, we encountered an issue that MATLAB has a limit on the number of variables that can be created in its workspace (we note that Forth et al. [2004] and Tadjouddine et al. [2002] encountered similar issues with regard to running out of registers). If the number of variables exceeds this limit, MATLAB will terminate a function call with an error stating that this limit has been exceeded. Because the derivative code created by *CADA* produces a function and derivative handle for every mathematical operation that is differentiated, it is realistic that this limit may be encountered. Reducing the number of variables in the workspace also reduces the amount of memory needed to execute the function, and improves performance. Thus, it is important to reduce the number of unique function handles that exist in the derivative file.

The following approach was used to greatly reduce the number of unique function handles in the final derivative code. From the **FUNCTIONLOCATION** property of the **GLOBALCADA** variable, the lines corresponding to the first and last occurrence of each intermediate function and its derivatives are known. Using this information, a sorting algorithm is used to determine those function and derivative handles that can be replaced by previously used handles, where the previously used handles do not occur later in the derivative file. Then, when the final derivative file is written, the old function number for each intermediate function and derivative is replaced by a previously used function number. If there is no previously used function number available, the number does not change. For example, suppose that n is a particular overloaded operation and cadaf$n$f and dcadaf$n$fdx are the function handle strings associated with this overloaded operation. If a function handle number m has been used earlier in the temporary file and this lower-numbered function handle is no longer needed at the point in the file where function handle n appears, then every occurrence of cadaf$n$f and dcadaf$n$fdx can be replaced with cadaf$m$f and dcadaf$m$fdx. Using this approach, the final derivative file will contain the fewest number of unique function handles as possible, thus reducing the number of unique variables that are created in the workspace when evaluating the derivative file on a numeric input.

4.8.2. Printing File with Nonzero Derivatives and Corresponding Indices. The final derivative file can be created in one of three ways. First, suppose that the function being differentiated is either a scalar function of a vector or a vector function of a vector, that is, $\mathbf{y} = \mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$. In this case the file that evaluates the the function and Jacobian of $\mathbf{f}(\mathbf{x})$ can be printed using the *CADA* function **cadajacobian**. Second, suppose we restrict attention to a scalar function of a vector, that is, $y = f(\mathbf{x})$ where $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. In this case a file that outputs the function, gradient, and Hessian can be created using function **cadahessian**. In both of these cases the derivative (Jacobian or Hessian) is written as a sparse matrix in MATLAB. If desired, **cadajacobian** and **cadahessian** can also produce an additional executable function that outputs only the sparsity pattern of the corresponding Jacobian or Hessian function. The third and final case of printing derivatives corresponds to a function that is *neither* a scalar or vector function of a vector. In this case it is not possible to use either **cadajacobian** or **cadahessian**. Instead, in this more general case a MATLAB function called **cadagenderiv** must be used. When using the function **cadagenderiv**, the nonzero derivatives, along with their *unrolled* indices [see Eq. (5)]. When using the function **cadagenderiv** it is necessary for the user to have knowledge of the relationship between the indices corresponding to the unrolled representation of the derivative and the *rolled* representation (that is, the indices in the actual multi-dimensional derivative).

Example of Post-Processing. Consider again the example in Section 4.7. Suppose that it is desired to compute the derivatives of $\mathbf{c}$ with respect to $\mathbf{x}$ and $\mathbf{y}$, where $\mathbf{x}$ and $\mathbf{y}$ were defined in the example in Section 4.4. These derivatives can be printed to a derivative file by applying the function **cadagenderiv** to the *CADA* object $\mathcal{C}$ as follows:

```
>> cadagenderiv(c,'ExampleDerivative')
```

In the process of applying **cadagenderiv** to the *CADA* object $\mathcal{C}$, the temporary file has been removed and the derivative file, 'ExampleDerivative.m', has been created. The file 'ExampleDerivative.m" is an executable MATLAB and given as

```
function output = ExampleDerivative(input)
x = input.x;
y = input.y;
z = input.z;
cadadxdx = ones(3,1);
cadadydy = ones(3,1);
cadaunit = size(x(1),1);
cadaf1f = sin(x);
cadatf1 = x((1:3));
dcadaf1fdx = cos(cadatf1(:)).*cadadxdx;
cadaf2f = cadaf1f .* y;
cadatf1 = y((1:3));
dcadaf2fdx = cadatf1(:).*dcadaf1fdx;
cadatf1 = cadaf1f((1:3));
dcadaf2fdy = cadatf1(:).*cadadydy;
cadaf3f = vertcat(z,cadaf2f);
dcadaf3fdx = zeros(3,cadaunit);
dcadaf3fdx((1:3)) = dcadaf2fdx;
dcadaf3fdy = zeros(3,cadaunit);
dcadaf3fdy((1:3)) = dcadaf2fdy;
output.f1.value = cadaf3f;
output.f1.dx.value = dcadaf3fdx;
cadadind1 = [2 7 12];
[cadadind1,cadadind2] = ind2sub([4,3],cadadind1);
output.f1.dx.location = [cadadind1',cadadind2'];
output.f1.dy.value = dcadaf3fdy;
cadadind1 = [2 7 12];
[cadadind1,cadadind2] = ind2sub([4,3],cadadind1);
output.f1.dy.location = [cadadind1',cadadind2'];
```

Because the function **cadagenderiv** is designed for any number of input *CADA* instances and any number of output functions, the input and output of the executable file are structured in order to better organize the resulting evaluations. In order to see how the input and output are organized, consider the following numerical values of x, y, and z:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \quad \mathbf{z} = 7. \tag{8}$$

The function *ExampleDerivative.m* is then evaluated using MATLAB on these numerical inputs as follows:

```
>> input.x = [1; 2; 3];
>> input.y = [4; 5; 6];
>> input.z = 7;

>> output = ExampleDerivative(input);
```

The value of c, the derivatives of c with respect to x and y, and the locations of the derivatives of c with respect to x and y, when evaluated on the numeric inputs given in Eq. (8), are then given as

```
output.f1.value =

    7.0000
    3.3659
    4.5465
    0.8467

output.f1.dx.value =

    2.1612
   -2.0807
   -5.9400

output.f1.dx.location =

     2     1
     3     2
     4     3

output.f1.dy.value =
    0.8415
    0.9093
    0.1411

output.f1.dy.location =

     2     1
     3     2
     4     3
```

It is noted that only the nonzero derivatives with respect to x and y are computed (that is, derivatives corresponding to the nonzero locations), and derivatives with respect to z are *not* calculated because z was declared during the setup to be a variable for which derivatives were *not* desired.

5. EXAMPLES

In this section we apply *CADA* to four examples. The examples provide coverage of the significant capabilities of *CADA* and compare *CADA* against other leading automatic differentiation software. The first example is the classic Speelpenning problem where the goal is to demonstrate that *CADA* avoids expression explosion when computing the derivative. The analysis of the Speelpenning problem also contains a comparison of *CADA* with the MATLAB Symbolic Math Toolbox where it is well known that symbolic differentiation does suffer from expression explosion. The second example is the Arrowhead function where the goal is to compare the efficiency of *CADA* against the leading operator overloaded software programs *INTLAB* [Rump 1999], *ADMAT* [Coleman and Verma 1998a; CAYUGA RESEARCH 2009], and *MAD* [Forth 2006]. The results of this second example show that *CADA* compares favorably with the computational efficiency of *INTLAB* and outperforms both *ADMAT* and *MAD*. In addition, the second example demonstrates the ability of *CADA* to generate not only sparse code, but also code that is vectorized if the original function code is vectorized. The third example is the Brown function from the MATLAB optimization toolbox. The goal of the third example is to demonstrate the efficiency with which *CADA* generates the Hessian of a scalar function of a vector and to compare the efficiency of *CADA* against *INTLAB*, *ADMAT*, and *MAD* when evaluating the Hessian. The fourth example is a large sparse nonlinear programming problem (NLP). The objective of the fourth example is to show the ability of *CADA* to generate the constraint Jacobian and Lagrangian Hessian of a complex NLP. Similar to the first three examples, in the fourth example the efficiency of *CADA* is compared against *INTLAB*, *ADMAT*, and *MAD*. Finally, it is noted that all computations shown in this section were performed using an Apple MacPro 2.26 GHz Quad-Core Intel Xeon computer with 16 GB of RAM running Mac OS-X Snow Leopard version 10.6.8 and MATLAB R2010b [Mathworks 2010].

Example 1: Speelpenning Problem

Consider the following function [Speelpenning 1980]:

$$f(\mathbf{x}) = \prod_{i=1}^{n} x_i, \quad \mathbf{x} \in \mathbb{R}^n, \quad f : \mathbb{R}^n \longrightarrow \mathbb{R}. \tag{9}$$

It is well known that the derivatives of the function in Eq. (9) are prone to expression explosion as n gets large when Eq. (9) is implemented in the form considered in Speelpenning [1980] (that is, the function is implemented in the form of a *loop* as opposed to using the built-in product function in MATLAB). A code that utilizes a loop to evaluate the function in Eq. (9) is shown in Fig. 5. Because Eq. (9) requires the computation of a product of length n, the gradient requires the computation of $n-1$ products between the n terms. The goal of this example is to demonstrate the effectiveness with which *CADA* avoids function explosion when writing the file that is capable of computing the gradient of Eq. (9), and the corresponding efficiency with which the resulting derivative code executes on a numeric argument. In this analysis the results obtained using *CADA* are compared against the MATLAB Symbolic Toolbox (see Mathworks [2010]).

```
function y = speelpenning4sym(x, n)

y = x(1);
for Icount = 2:n;
   y = y.*x(Icount);
end
```

Fig. 5: File Containing MATLAB Implementation of the Function in Eq. (9).

First consider the manner in which the gradient file is written using *CADA*. For the purpose of this first part of this example, let $n = 6$. The code corresponding to the *temporary* file created by *CADA prior* to post-processing is shown in Fig. 6. Specifically, it is seen in Fig. 6 that eleven individual operations are performed, thus leading to the creation of eleven function handles (and corresponding derivative function handles) in the temporary derivative file, and that the sequence of operations written to the temporary derivative file are the same as those that would have been obtained had the forward mode been applied on a numeric value of the vector x. Furthermore, a closer examination of Fig. 6 shows that in the early part of the temporary file the first three function handles (defined in the lines containing 'cadaf1f', 'cadaf2f', 'cadaf3f', 'dcadaf1fdx', 'dcadaf2fdx', and 'dcadaf3fdx') can be re-used after the sixth line without creating a variable conflict. As a result, higher-numbered function handles after the sixth line in the temporary file can be replaced with these lower-numbered function handles from before the sixth line. As a result the number of function handles in the final derivative file can be reduced significantly when post-processing the derivative file using the post-processing step as described in Section 4.8. Figure. 7 shows how the post-processing of the temporary derivative file reduces the number of unique function handles in the final file. It is seen in the final derivative file that the first three function handles, created in lines one through six in the temporary file, are *re-used* later in the file, thus reducing the number of newly created variables in the MATLAB workspace when evaluating the derivative file on a numeric input.

Next, we compare the efficiency with which *CADA* and the MATLAB Symbolic Math Toolbox both to generate and evaluate the derivatives code. Figures 8a–8c show, respectively, the derivative code generate time required by *CADA* and the MATLAB Symbolic Math Toolbox, the computation time required to evaluate the resulting derivative functions using each generated file on an input vector of all ones, and the gradient-to-function computation time ratio. It is seen that the derivative file is created much more quickly using *CADA* as compared with the MATLAB Symbolic Toolbox, and this time difference grows quickly as n increases. As an example, for $n = 250$ the MATLAB Symbolic Math Toolbox creates the derivative file in approximately 200 s. On the other hand, it takes *CADA* approximately 45 s to create the derivative file for $n = 10000$. Furthermore, it can be seen in Fig. 8a that the time required to produce the derivative file using the MATLAB Symbolic Toolbox appears to asymptotically approaches a vertical at $n \approx 316$. The rapid growth in derivative code generation times for the MATLAB Symbolic Toolbox is a result of expression explosion, where the symbolic expression grows in size and complexity such that the effort required to differentiate the expression becomes so large that the process becomes intractable. However, the *CADA*

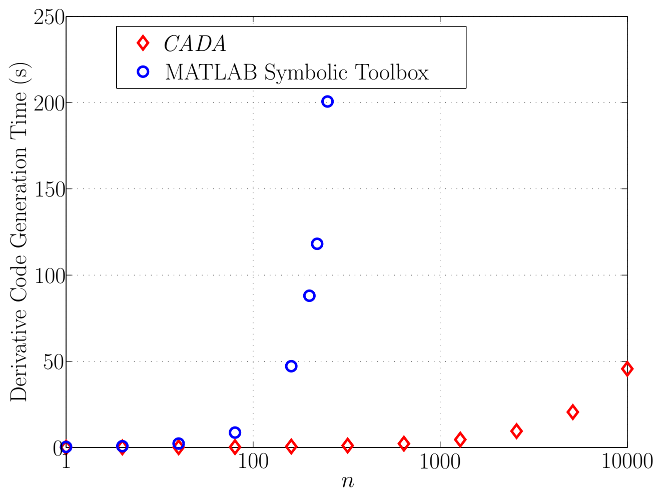
```
cadaf1f = x(1);
dcadaf1fdx = cadadxdx(1);
cadaf2f = x(2);
dcadaf2fdx = cadadxdx(2);
cadaf3f = cadaf1f .* cadaf2f;
dcadaf3fdx = zeros(2,cadaunit);
cadadind1 = 1;
cadadind2 = 2;
cadatf1 = cadaf1f(1);
cadatf2 = cadaf2f(1);
dcadaf3fdx(cadadind1) = cadatf2(:).*dcadaf1fdx;
dcadaf3fdx(cadadind2) = dcadaf3fdx(cadadind2) + cadatf1(:).*dcadaf2fdx;
cadaf4f = x(3);
dcadaf4fdx = cadadxdx(3);
cadaf5f = cadaf3f .* cadaf4f;
dcadaf5fdx = zeros(3,cadaunit);
cadadind1 = (1:2);
cadadind2 = 3;
cadatf1 = cadaf3f(1);
cadatf2 = cadaf4f(1*ones(2,1));
dcadaf5fdx(cadadind1) = cadatf2(:).*dcadaf3fdx;
dcadaf5fdx(cadadind2) = dcadaf5fdx(cadadind2) + cadatf1(:).*dcadaf4fdx;
cadaf6f = x(4);
dcadaf6fdx = cadadxdx(4);
cadaf7f = cadaf5f .* cadaf6f;
dcadaf7fdx = zeros(4,cadaunit);
cadadind1 = (1:3);
cadadind2 = 4;
cadatf1 = cadaf5f(1);
cadatf2 = cadaf6f(1*ones(3,1));
dcadaf7fdx(cadadind1) = cadatf2(:).*dcadaf5fdx;
dcadaf7fdx(cadadind2) = dcadaf7fdx(cadadind2) + cadatf1(:).*dcadaf6fdx;
cadaf8f = x(5);
dcadaf8fdx = cadadxdx(5);
cadaf9f = cadaf7f .* cadaf8f;
dcadaf9fdx = zeros(5,cadaunit);
cadadind1 = (1:4);
cadadind2 = 5;
cadatf1 = cadaf7f(1);
cadatf2 = cadaf8f(1*ones(4,1));
dcadaf9fdx(cadadind1) = cadatf2(:).*dcadaf7fdx;
dcadaf9fdx(cadadind2) = dcadaf9fdx(cadadind2) + cadatf1(:).*dcadaf8fdx;
cadaf10f = x(6);
dcadaf10fdx = cadadxdx(6);
cadaf11f = cadaf9f .* cadaf10f;
dcadaf11fdx = zeros(6,cadaunit);
cadadind1 = (1:5);
cadadind2 = 6;
cadatf1 = cadaf9f(1);
cadatf2 = cadaf10f(1*ones(5,1));
dcadaf11fdx(cadadind1) = cadatf2(:).*dcadaf9fdx;
dcadaf11fdx(cadadind2) = dcadaf11fdx(cadadind2) + cadatf1(:).*dcadaf10fdx;
```

Fig. 6: Temporary Derivative File Created by *CADA* for Speelpenning Problem with $n = 6$.

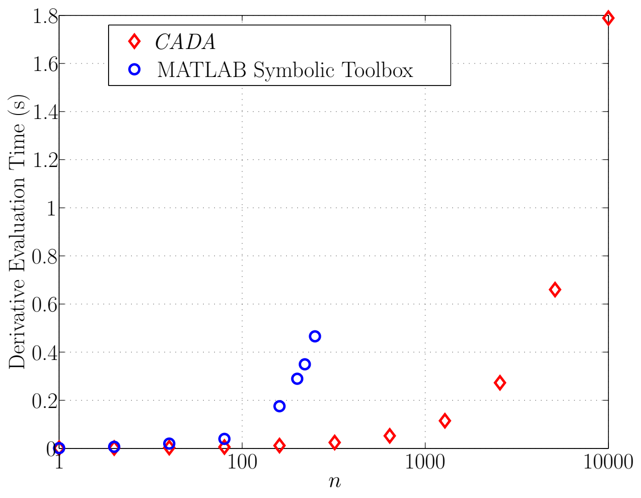
derivative code generation times grow at a significantly slower rate as a function of n. Next, examining Fig. 8b it is seen that the time required to compute the derivative using the code generated by *CADA* is reduced when compared to the code generated by the MATLAB Symbolic Toolbox. Moreover, examining Fig. 8c, it is seen that the ratio $\text{CPU}(\mathbf{J}f)/\text{CPU}(f)$ grows rapidly using the MATLAB symbolic toolbox because of expression explosion, while $\text{CPU}(\mathbf{J}f)/\text{CPU}(f)$ using *CADA* grows at a slightly faster than logarithmic rate because *CADA* does not suffer from the expression explosion seen in the MATLAB symbolic toolbox.

```
function [Jac,Fun] = speelpenning_Jac(x)
cadadxdx = ones(6,1);
cadaunit = size(x(1),1);
cadaf1f = x(1);
dcadaf1fdx = cadadxdx(1);
cadaf2f = x(2);
dcadaf2fdx = cadadxdx(2);
cadaf3f = cadaf1f .* cadaf2f;
dcadaf3fdx = zeros(2,cadaunit);
cadadind1 = 1;
cadadind2 = 2;
cadatf1 = cadaf1f(1);
cadatf2 = cadaf2f(1);
dcadaf3fdx(cadadind1) = cadatf2(:).*dcadaf1fdx;
dcadaf3fdx(cadadind2) = dcadaf3fdx(cadadind2) + cadatf1(:).*dcadaf2fdx;
cadaf1f = x(3);
dcadaf1fdx = cadadxdx(3);
cadaf2f = cadaf3f .* cadaf1f;
dcadaf2fdx = zeros(3,cadaunit);
cadadind1 = (1:2);
cadadind2 = 3;
cadatf1 = cadaf3f(1);
cadatf2 = cadaf1f(1*ones(2,1));
dcadaf2fdx(cadadind1) = cadatf2(:).*dcadaf3fdx;
dcadaf2fdx(cadadind2) = dcadaf2fdx(cadadind2) + cadatf1(:).*dcadaf1fdx;
cadaf3f = x(4);
dcadaf3fdx = cadadxdx(4);
cadaf1f = cadaf2f .* cadaf3f;
dcadaf1fdx = zeros(4,cadaunit);
cadadind1 = (1:3);
cadadind2 = 4;
cadatf1 = cadaf2f(1);
cadatf2 = cadaf3f(1*ones(3,1));
dcadaf1fdx(cadadind1) = cadatf2(:).*dcadaf2fdx;
dcadaf1fdx(cadadind2) = dcadaf1fdx(cadadind2) + cadatf1(:).*dcadaf3fdx;
cadaf2f = x(5);
dcadaf2fdx = cadadxdx(5);
cadaf3f = cadaf1f .* cadaf2f;
dcadaf3fdx = zeros(5,cadaunit);
cadadind1 = (1:4);
cadadind2 = 5;
cadatf1 = cadaf1f(1);
cadatf2 = cadaf2f(1*ones(4,1));
dcadaf3fdx(cadadind1) = cadatf2(:).*dcadaf1fdx;
dcadaf3fdx(cadadind2) = dcadaf3fdx(cadadind2) + cadatf1(:).*dcadaf2fdx;
cadaf1f = x(6);
dcadaf1fdx = cadadxdx(6);
cadaf11f = cadaf3f .* cadaf1f;
dcadaf11fdx = zeros(6,cadaunit);
cadadind1 = (1:5);
cadadind2 = 6;
cadatf1 = cadaf3f(1);
cadatf2 = cadaf1f(1*ones(5,1));
dcadaf11fdx(cadadind1) = cadatf2(:).*dcadaf3fdx;
dcadaf11fdx(cadadind2) = dcadaf11fdx(cadadind2) + cadatf1(:).*dcadaf1fdx;
cadadind1 = (1:6);
[cadadind1,cadadind2] = ind2sub([1,6],cadadind1);
Jac = sparse(cadadind1,cadadind2,dcadaf11fdx,1,6);
Fun = cadaf11f;
```

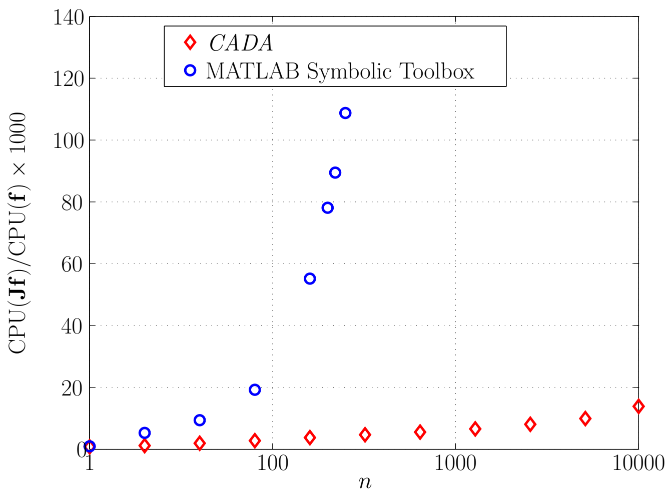
Fig. 7: Final Derivative File Created by *CADA* for Speelpenning Problem with $n = 6$.



(a) Derivative Code Generation Time vs. n.



(b) Derivative Evaluation Time vs. n.



(c) Ratio CPU(**Jf**)/CPU(**f**) vs. n.

Fig. 8: Derivative Code Generation Time and Jacobian-to-Function Computation Time Ratio, CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$), vs n for Example 1. Derivative Code Generation Time Are Averaged Over Over 10 Evaluations, While CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) Is Obtained By Averaging the Values Obtained Over 1000 Jacobian and Function Evaluations.

Example 2: Arrowhead Function

Consider the Arrowhead function $\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ taken from Section 7.4 of [Griewank 2008]:

$$f_1(\mathbf{x}) \;=\; 2x_1^2 + \textstyle\sum_{i=1}^{n} x_i^2 \;\;,\;\; f_j(\mathbf{x}) \;=\; x_1^2 + x_j^2, \quad (j = 2, \ldots, n). \tag{10}$$

It is well known that the function in Eq. (10) has a sparse Jacobian, and this sparsity is shown in Fig. 9 for $n = 7$. It is seen that the Jacobian of Eq. (10) has nonzero entries in only the first column, first row, and main diagonal. It is also interesting to note that the function in Eq. (10) has the property that function elements $2, \ldots, n$ can be computed in a *vectorized* manner, thus taking advantage of computational efficiency in MATLAB. In this example we demonstrate the fact that *CADA* generates derivatives sparsely and only writes the nonzero elements (along with the indices) to the final derivative file. In addition, we demonstrate how *CADA* takes advantage of vectorization of the original function by producing a derivative code that is itself vectorized. In this example we compare the efficiency of *CADA* against the automatic differentiation tools *INTLAB* [Rump 1999], *ADMAT* [Coleman and Verma 1998a], and *MAD* [Forth 2006]. In performing this comparison, *INTLAB* was used in its only available sparse forward mode, *ADMAT* was used in the manner described on pages 25–27 of the *ADMAT* User's Guide [Cayuga Research 2009], and *MAD* was used in sparse forward mode as described in Section 5.3 of Forth [2006]. Finally, it is noted in this example that the first function depends upon all of the components of x which functions $f_2(\mathbf{x}), \ldots, f_n(\mathbf{x})$ depend upon only two components of x. As a result, *ADMAT* computes the derivative of $f_1(\mathbf{x})$ using *reverse mode* and computes the derivatives of the remaining $n-1$ functions using *forward mode*.

$$\mathbf{struct}(\mathbf{Jf}) = \begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & & & & \\ \bullet & & \bullet & & & & \\ \bullet & & & \bullet & & & \\ \bullet & & & & \bullet & & \\ \bullet & & & & & \bullet & \\ \bullet & & & & & & \bullet \end{bmatrix}$$

Fig. 9: Sparsity Pattern of Example 2 [Eq. (10)].

Table. I shows the time required for *CADA* to generate the Jacobian code of the function in Fig. 10 alongside the Jacobian-to-function computation time ratio as a function of n using *CADA*, *INTLAB*, *ADMAT*, and *MAD*. It is seen that CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) using *CADA* is smaller than any of the other tools over the entire range from $n = 10$ to $n = 5120$. Next, CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) using any of the tools is approximately constant for small to moderate values of n. As n increases, however, *INTLAB* becomes more efficient, *ADMAT* remains approximately constant in efficiency, and *MAD* becomes increasingly inefficient. Moreover, it is interesting to see that for the largest values of n in Table. I the ratio CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) is approximately the same using either *CADA* and *INTLAB*, but this ratio is approximately six times higher using *ADMAT* and approximately 70 times greater using *MAD*. Using *CADA* the ratio CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) increases with n because *CADA* needs to read the nonzero indices each time the Jacobian is evaluated and this index reading consumes more computation time as n increases. Using *INTLAB*, the ratio CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) decreases with n because *INTLAB* performs purely sparse arithmetic and this sparse arithmetic becomes more efficient as n increases. Using *ADMAT* the ratio CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$) is nearly constant because, as stated above, *ADMAT* computes the derivative of the first function $f_1(\mathbf{x})$ in Eq. (10)

using *reverse mode*, but uses sparse *forward mode* to differentiate the remaining $n-1$ functions. Using *MAD* the ratio $\text{CPU}(\mathbf{Jf})/\text{CPU}(\mathbf{f})$ is relatively low for small to moderate values of n, but climbs to very high values for large n because *MAD* has the added complexity of allowing a user to compute higher-order derivatives by overloading an instance of the **fmad** class, and this ability to repeatedly overload objects increases the overhead for large values of n.

We now consider the manner in which the Jacobian is created when the original function is vectorized. Figure 10 shows the MATLAB code for Eq. (10), where it is seen that the function code is vectorized. Next, Figs. 11 and 12 show the Jacobian files created by *CADA* using the function in Fig. 10 (and creating the Jacobian using the *CADA* function **cadajacobian**) for $n = 5$ and $n = 15$, respectively. It is seen that the two Jacobian files are the same length (56 lines of code to be exact). The reason that the number of lines in the Jacobian file is independent of n is that *CADA* vectorizes the derivative computations, leading to a derivative file that contains the exact same sequence of vectorized operations regardless of the size of n. Again, we note that this sequence of mathematical operations is the same as applying the forward mode on a numeric value of the input. Because the amount of text written to the Jacobian file changes with n only by the number of indices corresponding to the nonzero derivatives (that is, the larger the value of n, the more nonzero indices are written to the file), the time required to generate the derivative is a function of the time required for *CADA* to compute the nonzero derivatives and print the corresponding nonzero derivative indices. Because the number of scalar mathematical operations grows as n increases, the time required for *CADA* to evaluate the resulting Jacobian file also grows, albeit slowly because the derivatives are being computed sparsely.

Table I: *CADA* Derivative Code Generation Time and Ratio of Jacobian-to-Function Computation Time, $\text{CPU}(\mathbf{Jf})/\text{CPU}(\mathbf{f})$, for Example 2 Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. Derivative Code Generation Times Were Averaged Over 100 Evaluations on a *CADA* Instance, While $\text{CPU}(\mathbf{Jf})/\text{CPU}(\mathbf{f})$ Was Obtained by Averaging the Values Obtained Over 1000 Jacobian and Function Evaluations.

n	Derivative Code Generation Time Using *CADA* (s)	CPU(**Jf**)/CPU(**f**) Ratio Using *CADA*	CPU(**Jf**)/CPU(**f**) Ratio Using *INTLAB*	CPU(**Jf**)/CPU(**f**) Ratio Using *ADMAT*	CPU(**Jf**)/CPU(**f**) Ratio Using *MAD*
10	0.0308	19.9	196.7	589.8	316.9
20	0.0312	27.6	198.7	595.2	322.1
40	0.0334	27.9	196.2	596.1	313.1
80	0.0408	28.5	188.4	595.2	313.3
160	0.0541	29.9	181.6	591.3	313.9
320	0.0819	43.7	170.6	581.7	346.1
640	0.135	53.1	149.4	585.3	546.1
1280	0.246	76.8	122.3	574.9	1235.1
2560	0.472	79.6	82.5	573.7	2832.1
5120	0.857	86.4	89.6	581.2	6206.7

```
function y = arrowhead4cada(x)

y = zeros(size(x));
y(1) = 2*x(1).^2+sum(x.^2);
y(2:end) = x(1).^2+x(2:end).^2;
```

Fig. 10: MATLAB Code for Example 2 [Eq. (10)].

```
function [Jac,Fun] = arrowhead_der(x)
cadadxdx = ones(5,1);
cadaunit = size(x(1),1);
cadaf1f = x(1);
dcadaf1fdx = cadadxdx(1);
cadaf2f = cadaf1f.^2;
cadatf1 = cadaf1f(1);
dcadaf2fdx = 2.*cadatf1(:).^1.*dcadaf1fdx;
cadaf3f = cadaf2f .* 2;
dcadaf3fdx = 2 .* dcadaf2fdx;
cadaf1f = x.^2;
cadatf1 = x((1:5));
dcadaf1fdx = 2.*cadatf1(:).^1.*cadadxdx;
cadaf2f = sum(cadaf1f);
cadadind1 = [1 7 13 19 25];
cadatd1 = zeros(5,5,cadaunit);
cadatd1(cadadind1) = dcadaf1fdx;
cadatd1 = sum(cadatd1);
dcadaf2fdx = zeros(5,cadaunit);
dcadaf2fdx(:) = cadatd1((1:5));
cadaf1f = cadaf3f + cadaf2f;
dcadaf1fdx = zeros(5,cadaunit);
cadadind1 = 1;
cadadind2 = (1:5);
dcadaf1fdx(cadadind2) = dcadaf2fdx;
dcadaf1fdx(cadadind1) = dcadaf1fdx(cadadind1) + dcadaf3fdx;
cadaf7f = zeros(5,1,cadaunit);
cadaf7f(1) = cadaf1f;
dcadaf7dx = dcadaf1fdx;
cadaf3f = x(1);
dcadaf3fdx = cadadxdx(1);
cadaf1f = cadaf3f.^2;
cadatf1 = cadaf3f(1);
dcadaf1fdx = 2.*cadatf1(:).^1.*dcadaf3fdx;
cadaf2f = x((2:5));
dcadaf2fdx = cadadxdx((2:5));
cadaf3f = cadaf2f.^2;
cadatf1 = cadaf2f((1:4));
dcadaf3fdx = 2.*cadatf1(:).^1.*dcadaf2fdx;
cadaf2f = repmat(cadaf1f,4,1);
dcadaf2fdx = dcadaf1fdx(1*ones(4,1));
cadaf1f = cadaf2f + cadaf3f;
dcadaf1fdx = zeros(8,cadaunit);
cadadind1 = (1:4);
cadadind2 = (5:8);
dcadaf1fdx(cadadind1) = dcadaf2fdx;
dcadaf1fdx(cadadind2) = dcadaf1fdx(cadadind2) + dcadaf3fdx;
cadaf14f = cadaf7f;
cadaf14f((2:5)) = cadaf1f;
dcadaf14fdx = zeros(13,cadaunit);
dcadaf14fdx([1 6 8 10 12]) = dcadaf7dx((1:5));
dcadaf14fdx([2:5 7 9 11 13]) = dcadaf1fdx;
cadadind1 = [1:7 11 13 16 19 21 25];
[cadadind1,cadadind2] = ind2sub([5,5],cadadind1);
Jac = sparse(cadadind1,cadadind2,dcadaf14fdx,5,5);
Fun = cadaf14f;
```

Fig. 11: Jacobian of Function in Fig. 10 Using *CADA* for $n = 5$.

```
function [Jac,Fun] = arrowhead_der(x)
cadadxdx = ones(15,1);
cadaunit = size(x(1),1);
cadaf1f = x(1);
dcadaf1fdx = cadadxdx(1);
cadaf2f = cadaf1f.^2;
cadatf1 = cadaf1f(1);
dcadaf2fdx = 2.*cadatf1(:).^1.*dcadaf1fdx;
cadaf3f = cadaf2f .* 2;
dcadaf3fdx = 2 .* dcadaf2fdx;
cadaf1f = x.^2;
cadatf1 = x((1:15));
dcadaf1fdx = 2.*cadatf1(:).^1.*cadadxdx;
cadaf2f = sum(cadaf1f);
cadadind1 = [1 17 33 49 65 81 97 113 129 145 161 177 193 209 225];
cadatd1 = zeros(15,15,cadaunit);
cadatd1(cadadind1) = dcadaf1fdx;
cadatd1 = sum(cadatd1);
dcadaf2fdx = zeros(15,cadaunit);
dcadaf2fdx(:) = cadatd1((1:15));
cadaf1f = cadaf3f + cadaf2f;
dcadaf1fdx = zeros(15,cadaunit);
cadadind1 = 1;
cadadind2 = (1:15);
dcadaf1fdx(cadadind2) = dcadaf2fdx;
dcadaf1fdx(cadadind1) = dcadaf1fdx(cadadind1) + dcadaf3fdx;
cadaf7f = zeros(15,1,cadaunit);
cadaf7f(1) = cadaf1f;
dcadaf7dx = dcadaf1fdx;
cadaf3f = x(1);
dcadaf3fdx = cadadxdx(1);
cadaf1f = cadaf3f.^2;
cadatf1 = cadaf3f(1);
dcadaf1fdx = 2.*cadatf1(:).^1.*dcadaf3fdx;
cadaf2f = x((2:15));
dcadaf2fdx = cadadxdx((2:15));
cadaf3f = cadaf2f.^2;
cadatf1 = cadaf2f((1:14));
dcadaf3fdx = 2.*cadatf1(:).^1.*dcadaf2fdx;
cadaf2f = repmat(cadaf1f,14,1);
dcadaf2fdx = dcadaf1fdx(1*ones(14,1));
cadaf1f = cadaf2f + cadaf3f;
dcadaf1fdx = zeros(28,cadaunit);
cadadind1 = (1:14);
cadadind2 = (15:28);
dcadaf1fdx(cadadind1) = dcadaf2fdx;
dcadaf1fdx(cadadind2) = dcadaf1fdx(cadadind2) + dcadaf3fdx;
cadaf14f = cadaf7f;
cadaf14f((2:15)) = cadaf1f;
dcadaf14fdx = zeros(43,cadaunit);
dcadaf14fdx([1 16 18 20 22 24 26 28 30 32 34 36 38 40 42]) = dcadaf7dx((1:15));
dcadaf14fdx([2:15 17 19 21 23 25 27 29 31 33 35 37 39 41 43]) = dcadaf1fdx;
cadadind1 = [1:17 31 33 46 49 61 65 76 81 91 97 106 113 121 129 136 145 151 161 166 177 181 193 196 209 211 225];
[cadadind1,cadadind2] = ind2sub([15,15],cadadind1);
Jac = sparse(cadadind1,cadadind2,dcadaf14fdx,15,15);
Fun = cadaf14f;
```

Fig. 12: Jacobian of Function in Fig. 10 Using *CADA* for $n = 15$.

Example 3: Brown Function

Consider the following function taken from the MATLAB optimization toolbox [Mathworks 2010]:

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[(x_i^2)^{x_{i+1}^2+1} + (x_{i+1}^2)^{x_i^2+1} \right], \tag{11}$$

where $\mathbf{x} = (x_1, \ldots, n_n) \in \mathbb{R}^n$ and $f : \mathbb{R}^n \longrightarrow \mathbb{R}$. It is known that the Hessian of the function in Eq. (11), $\mathbf{H}f \equiv \mathbf{H}f(\mathbf{x}) = \partial^2 f/\partial \mathbf{x}^2$, is sparse. The sparsity pattern of $\mathbf{H}f$ is shown in Fig. 13 for $n = 8$ where it is seen that the nonzero elements of the Hessian lie on the main diagonal, the sub-diagonal, and super-diagonal. The objectives of this example are (a) to analyze the efficiency with with *CADA* generates the Hessian of Eq. (11); (b) compare the efficiency which which *CADA* evaluates the resulting Hessian file against *INTLAB* [Rump 1999], *ADMAT* [Coleman and Verma 1998a], and *MAD* [Forth 2006]; and (c) to compare the efficiency of generating second derivatives against the first derivative results obtained in Example 1.

$$\textbf{struct}(\mathbf{H}f) = \begin{bmatrix} \bullet & \bullet & & & & & & \\ \bullet & \bullet & \bullet & & & & & \\ & \bullet & \bullet & \bullet & & & & \\ & & \bullet & \bullet & \bullet & & & \\ & & & \bullet & \bullet & \bullet & & \\ & & & & \bullet & \bullet & \bullet & \\ & & & & & \bullet & \bullet & \bullet \\ & & & & & & \bullet & \bullet \end{bmatrix}$$

Fig. 13: Hessian Sparsity Pattern, $\textbf{struct}(\mathbf{H}f)$, of Example 3 [Eq. (11)] for $n = 8$.

Table II shows the derivative code generation time using *CADA* alongside the Hessian-to-function computation time ratio, CPU($\mathbf{H}f$)/CPU(f), using *CADA*, *INTLAB*, *ADMAT*, and *MAD*. Unlike Example 1, where the behavior in the first derivative Jacobian-to-function ratio was different for each tool, in this example it is seen that CPU($\mathbf{H}f$)/CPU(f) increases as a function of n using any of the automatic differentiation tools. Despite the fact that CPU($\mathbf{H}f$)/CPU(f) increases with n using any of the software programs, it is seen that CPU($\mathbf{H}f$)/CPU(f) is significantly larger using either *ADMAT* or *MAD* than it is using either *CADA* or *INTLAB*. Different from the result of Example 1, where the efficiency of *INTLAB* increased with n, in this case the efficiency of *INTLAB* decreases with n because the increased computational complexity associated with direct implementation of the rules for computing a second derivative. In the case of either *ADMAT* or *MAD*, the computational cost of computing the Hessian of $f(\mathbf{x})$ is much higher than the cost of computing a comparable sparse Jacobian because both *ADMAT* and *MAD* incur additional overhead due to operator overloading at the second derivative level. As a result, while *INTLAB* is less efficient in computing a sparse Hessian than it is computing a sparse Jacobian, *INTLAB* is still more efficient than either *ADMAT* or *MAD* because it is performing purely sparse arithmetic to compute the Hessian. Different from any of the other tools, *CADA* generates two separate derivative files [the first file generated computes the gradient of $f(\mathbf{x})$ while the second file generated computes the Hessian of $f(\mathbf{x})$]. As a result, the computational complexity of the Hessian derivative code is only as complex as the number of operations required to compute sparsely the first derivative of the gradient (that is, the Hessian of the original function).

Table II: *CADA* Derivative Generation Time and Hessian-to-Function Computation Time Ratio, CPU($\mathbf{H}f$)/CPU(f), for Example 3 Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. The Derivative Generation Time Using *CADA* Was Obtained by Averaging Over 100 Evaluations on a *CADA* Instance, While CPU($\mathbf{H}f$)/CPU(f) Was Obtained By Averaging the Values Obtained Over 1000 Jacobian and Function Evaluations.

n	Derivative Code Generation Time Using *CADA* (s)	CPU($\mathbf{H}f$)/CPU(f) Ratio Using *CADA*	CPU($\mathbf{H}f$)/CPU(f) Ratio Using *INTLAB*	CPU($\mathbf{H}f$)/CPU(f) Ratio Using *ADMAT*	CPU($\mathbf{H}f$)/CPU(f) Ratio Using *MAD*
10	0.184	18.8	128.7	509.8	366.5
20	0.226	24.3	127.8	522.4	359.5
40	0.277	58.0	127.7	534.3	383.6
80	0.396	89.9	128.3	599.7	389.2
160	0.655	129.1	143.1	683.9	529.2
320	1.142	189.2	230.3	846.2	1101.6
640	2.146	310.9	336.4	1236.3	3159.6
1280	4.331	547.1	715.9	2150.0	8143.5

Example 4: Sparse Nonlinear Programming

Consider the following sparse nonlinear programming problem (NLP) that arises from the discretization of the optimal control problem given on pages 247–253 of Betts [2009] using an hp–adaptive [Darby et al. 2011a; 2011b] version of the Radau pseudospectral method [Benson et al. 2006; Garg et al. 2010; Garg et al. 2011a; Garg et al. 2011b; Patterson and Rao 2012; Rao et al. 2010; Rao et al. 2011]. The NLP decision vector $\mathbf{x} \in \mathbb{R}^{8N+7}$ is given as

$$\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_6, \mathbf{w}_1, \mathbf{w}_2, \kappa), \tag{12}$$

where N is a parameter that defines the total number of *Legendre-Gauss-Radau* (LGR) points [Abramowitz and Stegun 1965], $\mathbf{x}_i = (x_{1,i}, \ldots, x_{N+1,i}),\ (i = 1, \ldots, 6)$, and $\mathbf{w}_i = (w_{1,i}, \ldots, w_{N,i}),\ (i = 1, 2)$. The objective of the NLP is to minimize the cost function

$$J = -x_{N+1,3} \tag{13}$$

subject to the nonlinear algebraic constraints*

$$\begin{aligned}
&\sum_{k=1}^{N+1} D_{i,k} x_{k,1} - \frac{\kappa}{2} x_{i,4} \sin x_{i,5} = 0, \\
&\sum_{k=1}^{N+1} D_{i,k} x_{k,2} - \frac{\kappa}{2} \frac{x_{i,4} \cos x_{i,5} \sin x_{i,6}}{x_{i,1} \cos x_{i,3}} = 0, \\
&\sum_{k=1}^{N+1} D_{i,k} x_{k,3} - \frac{\kappa}{2} \frac{x_{4,i} \cos x_{i,5} \cos x_{i,6}}{x_{i,1}} = 0, \\
&\sum_{k=1}^{N+1} D_{i,k} x_{k,4} - \frac{\kappa}{2} \left[-\alpha P_i \exp[-\beta(x_{i,1} - 1)] x_{i,4}^2 - \frac{\sin x_{i,5}}{x_{i,1}^2} \right] = 0, \\
&\sum_{k=1}^{N+1} D_{i,k} x_{k,5} - \frac{\kappa}{2} \left[\alpha Q_i \exp[-\beta(x_{i,1} - 1)] x_{i,4} \cos w_{i,1} + \left(x_{i,4} - \frac{1}{x_{i,1} x_{i,4}} \right) \frac{\cos x_{i,5}}{x_{1,i}} \right] = 0, \\
&\sum_{k=1}^{N+1} D_{i,k} x_{k,6} - \frac{\kappa}{2} \left[\frac{\alpha Q_i \exp[-\beta(x_{i,1} - 1)] x_{i,4} \sin w_{i,1}}{\cos x_{i,5}} + \frac{x_{i,4} \cos x_{i,5} \sin x_{i,6} \tan x_{i,3}}{x_{i,1}} \right] = 0,
\end{aligned} \tag{14}$$

*In order to maximize computational efficiency in MATLAB, the algebraic constraints given in Eq. (14) are implemented in a *vectorized* manner. As a result, the derivative files produced by *CADA* are the same number of lines regardless of the value of N used to solve this problem.

and the equality constraints

$$\begin{aligned} & x_{1,1}-u_1=0\,,\; x_{1,2}-u_2=0\,,\; x_{1,3}-u_3=0\,, \\ & x_{1,4}-u_4=0\,,\; x_{1,5}-u_5=0\,, \\ & x_{1,6}-u_6=0\,,\; x_{N+1,1}-u_7=0\,, \\ & x_{N+1,4}-u_8=0\,,\; x_{N+1,5}-u_9=0, \end{aligned} \tag{15}$$

where $u_1 = 1.01248$, $u_2 = 0$, $u_3 = 0$, $u_4 = 0.986496$, $u_5 = -0.0174533$, $u_6 = 1.57080$, $u_7 = 1.00383$, $u_8 = 0.0963375$, $u_9 = -0.0872664$, $\alpha = 10596.2$, $\beta = 878.273$, $\mu_0 = -0.20704$, $\mu_1 = 1.67556$, $\gamma_0 = 0.07854$, $\gamma_1 = -0.352896$, $\gamma_2 = 2.03996$, and $D_{i,k},\ (i = 1,\dots,N,\ k = 1,\dots,N+1)$ is the Radau pseudospectral differentiation matrix (see Appendix A or Patterson and Rao [2012]),

$$P_i \;=\; \gamma_0+\gamma_1 w_{i,2}+\gamma_2 w_{i,2}^2\ \ ,\ Q_i \;=\; \mu_0+\mu_1 w_{i,2}. \tag{16}$$

and $(i = 1,\dots,N)$ in Eqs. (14) and (16). The total number of LGR points, $N = N_k K$, is obtained by dividing the problem into K mesh intervals using N_k LGR points in each mesh interval (for details see Patterson and Rao 2012). The details of the matrix $D_{i,k},\ (i = 1,\dots,N,\ k = 1,\dots,N+1)$ are given in Appendix A.

The NLP defined in Eqs. (13) and (14) can be written in the following more generic mathematical form. Determine the decision vector $\mathbf{z} \in \mathbb{R}^n$ that minimizes the objective function

$$f(\mathbf{z}) \tag{17}$$

subject to the nonlinear equality constraints

$$\mathbf{g}(\mathbf{z}) = \mathbf{0} \tag{18}$$

and the simple bounds on the decision variables

$$\mathbf{z}_{\min} \le \mathbf{z} \le \mathbf{z}_{\max}. \tag{19}$$

We divide the analysis in this example into two parts. First, we study the efficiency with which *CADA* generates and computes the constraint Jacobian, $\partial \mathbf{g}/\partial \mathbf{z}$, and the Lagrangian Hessian, $\partial^2 L/\partial \mathbf{z}^2$, where $L = \sigma f(\mathbf{z}) + \boldsymbol{\lambda}^{\mathsf{T}}\mathbf{g}(\mathbf{z})$ and the variables $\sigma \in \mathbb{R}$ and $\boldsymbol{\lambda} \in \mathbb{R}^{6N}$ are the Lagrange multipliers for the cost and the constraints, respectively [where we note that $\boldsymbol{\lambda} \in \mathbb{R}^{6N}$ because the NLP contains $6N$ nonlinear constraints as shown in Eq. (14)]. Second, we demonstrate the efficiency of *CADA* in solving the NLP for various values of N. In order to incorporate both first and second derivatives, in this research the NLP was solved was solved using the second-derivative open-source NLP solver ***IPOPT*** [Biegler and Zavala 2008; Waechter and Biegler 2006] with the indefinite sparse symmetric linear solver MA57 [Duff 2004]. As a second derivative NLP solver, ***IPOPT*** requires the user to supply the objective function gradient, $\partial f/\partial \mathbf{z}$, the constraint Jacobian, $\partial \mathbf{g}/\partial \mathbf{z}$, the Lagrangian Hessian, $\partial^2 \mathcal{L}/\partial \mathbf{z}^2$, the sparsity pattern of the constraint Jacobian, and the sparsity pattern of the lower-triangular portion of the Lagrangian Hessian.

We analyze the efficiency of *CADA* in a manner similar to that which was used in Examples 1–3. In order to analyze efficiency as the NLP grows in size, in this example we fix the number of LGR points in each mesh interval to four (that is, $N_k = 4$) and we vary the number of mesh intervals K. Thus, in this example the total number of LGR points is $N = 4K$. Table III shows the Jacobian-to-function ratio, CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$), for the NLP constraint function of Eq. (14 as a function of N. It is seen that the ratio CPU($\mathbf{Jg}$)/CPU($\mathbf{g}$) using *CADA* is smaller than any of the other methods for all values of N except $N = 128$. In the case $N = 128$, ***INTLAB*** is slightly more efficient than *CADA* in generating the constraint Jacobian, but *CADA* is more efficient than either

ADMAT or *MAD*. Furthermore, consistent with the results obtained in the Arrowhead function of Example 1, it is seen that CPU$(\mathbf{J}\mathrm{g})/\mathrm{CPU}(\mathrm{g})$ increases with N using *CADA*, decreases using *INTLAB*, and remains essentially constant using either *ADMAT* or *MAD*. Table IV shows the Hessian-to-function ratio, CPU$(\mathbf{H}L)/\mathrm{CPU}(L)$, for the Lagrangian Hessian $L = \sigma f(\mathbf{z}) + \boldsymbol{\lambda}^{\mathsf{T}}\mathrm{g}(\mathbf{z})$. In the case of the Hessian Lagrangian it is seen that *CADA* is more efficient than *INTLAB* and is *significantly* more efficient than either *ADMAT* or *MAD* for all values of N. It is interesting to see for $N = 128$ that the ratios CPU$(\mathbf{H}L)/\mathrm{CPU}(L)$ using *ADMAT* and *MAD* are approximately 15 and 50 times higher, respectively, when compared against *CADA*.

We now turn our attention to the solution of the aforementioned NLP using *CADA*, *INTLAB*, *ADMAT*, and *MAD* with the NLP solver *IPOPT*. While *INTLAB* and *MAD* are designed to utilize forward mode automatic differentiation, *ADMAT* can be applied using either forward or reverse mode. Because the NLP of Eqs. (13) and (14) consists of a large number of functions (that is, a large number of outputs), we choose the forward mode option when using *ADMAT*. Next, because *IPOPT* requires that the user provide the constraint Jacobian and Lagrangian Hessian sparsity patterns, it was necessary to construct these patterns prior to solving the NLP. While it is possible to determine the Jacobian and Lagrangian Hessian sparsity patterns with *ADMAT*, we found the process for determining the Hessian sparsity pattern to be unacceptably slow. In addition, *INTLAB* does not have a built-in capability of determining either Jacobian or Hessian sparsity patterns. As a result, the approach used in this research was to use the constraint Jacobian and Lagrangian Hessian sparsity patterns that *CADA* generates simultaneous with its generation of the Jacobian and Hessian derivative functions. These sparsity patterns were then supplied to *IPOPT* for use with all three automatic differentiation programs. Finally, it is noted that the constraint Jacobian was determined by differentiating Eq. (14). On the other hand, it is known that the terms Eq. (14) involving the coefficients $D_{i,k},\ (i = 1,\ldots,N,\ j = 1,\ldots,N+1)$ are *linear* in the decision vector. Consequently, the second derivatives of these terms are zero and, thus, will not appear in the Lagrangian Hessian. Thus, the terms involving the coefficients $D_{i,k},\ (i = 1,\ldots,N,\ j = 1,\ldots,N+1)$ are not included in the computation of the NLP Lagrangian.

The NLP given in Eqs. (13)–(15) was solved using *IPOPT* for $K = (8, 16, 32, 64, 128)$ with $N_k = 4,\ (k = 1,\ldots,K)$ (that is, the same number of Legendre-Gauss-Radau points [Abramowitz and Stegun 1965] was used in every mesh interval) with the following initial guess:

$$\begin{aligned}
\{x_{1,1},\cdots,x_{N+1,1}\} &= \mathbf{linspace}(u_1,u_7,N+1)\\
\{x_{1,2},\ldots,x_{N+1,2}\} &= \mathbf{linspace}(u_1,u_2,N+1)\\
\{x_{1,3},\ldots,x_{N+1,3}\} &= \mathbf{linspace}(u_3,u_3,N+1)\\
\{x_{1,4},\cdots,x_{N+1,4}\} &= \mathbf{linspace}(u_4,u_8,N+1)\\
\{x_{1,5},\cdots,x_{N+1,5}\} &= \mathbf{linspace}(u_5,u_9,N+1)\\
\{x_{1,6},\ldots,x_{N+1,6}\} &= \mathbf{linspace}(u_6,u_6,N+1)\\
\{w_{1,1},\ldots,x_{N,1}\} &= \mathbf{linspace}(0,0,N)\\
\{w_{1,2},\ldots,x_{N,2}\} &= \mathbf{linspace}(0,0,N)\\
\kappa &= 1.241475432154217,
\end{aligned}$$

where the function $\mathbf{linspace}(a,b,M)$ provides a set of M linearly equally-spaced points between a to b. The solution obtained for this problem (which matches the solution in Betts [2009]) is shown in Fig. 14 for $K = 16$, where the NLP decision vector, x, is decomposed as given by Eq. (12). Using this decomposition of x, the vectors $(x_{1,j},\ldots,x_{N+1,j}),\ (j = 1,\ldots,6)$ are plotted as a function of $(s_1,\ldots,s_{N+1})$, where $(s_1,\ldots,s_N)$ are the N LGR points on the interval $[-1,+1)$ and $s_{N+1} = +1$, while the vectors $(w_{1,j},\ldots,x_{N,j}),\ (j = 1,2)$ are plotted against the LGR points $(s_1,\ldots,s_N)$. The computational results obtained for this example are shown in Tables V and VI (where the derivative code generation time includes the post-processing step that reduces the number of MATLAB variables). As might be expected, the derivative code generation time using *CADA* increases with K. In addition, it is seen that the time required to solve the NLP (excluding the time required to create the derivative files) using *CADA* is smaller than the corresponding computation time required by either *INTLAB* or *ADMAT*. Next, it is seen for $K = (8, 16)$ that the *total* computation time (that is, the derivative code generation time plus the time to solve the NLP) using derivatives obtained by *CADA* is slightly larger than the solution time required by either of the other automatic differentiators. On the other hand, for the larger values of K (that is, $K = (32, 64, 128)$), the total computation time using *CADA* is significantly *less* than either *INTLAB* or *ADMAT*. Thus, *CADA* becomes more computationally attractive as the size of the NLP increases.

Another interesting advantage of *CADA* over either *INTLAB* or *ADMAT* pertains to the determination of the constraint Jacobian and Lagrangian Hessian sparsity patterns required by *IPOPT*. It is known that *INTLAB* does not have a built-in ability to compute sparsity patterns, while the sparsity pattern generator using *ADMAT* is quite inefficient (particularly when determining the sparsity pattern of the Lagrangian Hessian). Thus, neither *INTLAB* nor *ADMAT* is a practical option for determining sparsity patterns. As a result, this example demonstrates that *CADA* is not only efficient for generating derivatives, but serves the dual purpose that it is capable of efficiently generating the required sparsity patterns.

Table III: *CADA* Derivative Generation Time and Jacobian-to-Constraint Computation Time Ratio, CPU($\mathbf{J}$g)/CPU(g), for Example 4 Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. Derivative Generation Times Using *CADA* Were Averaged Over 100 File Generations, While CPU($\mathbf{J}$g)/CPU(g) Was Obtained by Averaging the Values Obtained Over 1000 Jacobian and Function Evaluations.

K	$N = 4K$	Derivative Code Generation Time Using *CADA* (s)	CPU(**Jg**)/CPU(**g**) Ratio Using *CADA*	CPU(**Jg**)/CPU(**g**) Ratio Using *INTLAB*	CPU(**Jg**)/CPU(**g**) Ratio Using *ADMAT*	CPU(**Jg**)/CPU(**g**) Ratio Using *MAD*
8	32	0.493	10.0	56.7	62.9	80.5
16	64	0.798	14.1	56.3	63.7	78.2
32	128	1.397	18.8	53.2	67.2	97.1
64	256	2.611	27.4	49.3	67.3	86.3
128	512	5.090	49.3	45.2	68.8	72.8

Table IV: *CADA* Derivative Generation Time and Lagrangian Hessian to Lagrangian Computation Time Ratio, CPU($\mathbf{H}L$)/CPU(L) for Example 4 Using *CADA*, *INTLAB*, *ADMAT* and *MAD*. Derivative Generation Times Using *CADA* Were Averaged Over 100 File Generations, While CPU($\mathbf{H}L$)/CPU(L) Was Obtained by Averaging the Values Obtained Over 1000 Lagrangian Hessian and Lagrangian Function Evaluations.

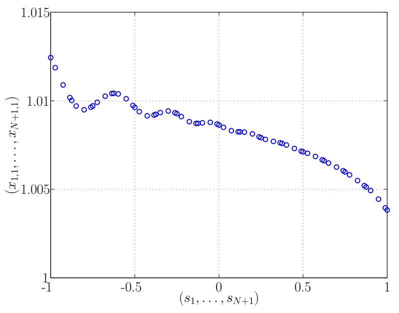
K	$N = 4K$	Hessian Code Generation Time Using *CADA* (s)	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *CADA*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *INTLAB*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *ADMAT*	CPU($\mathbf{H}L$)/CPU(L) Ratio Using *MAD*
8	32	2.749	84.5	253.7	1862	1965
16	64	4.557	154.9	344.8	2245	3120
32	128	8.189	272.8	904.4	3070	8227
64	256	15.49	512.4	2220	5604	21452
128	512	30.81	1145	5191	16565	57968

Table V: *IPOPT* Solution Times for Example 4 Using Derivatives Supplied by *CADA*, *INTLAB*, *ADMAT*, and *MAD*.

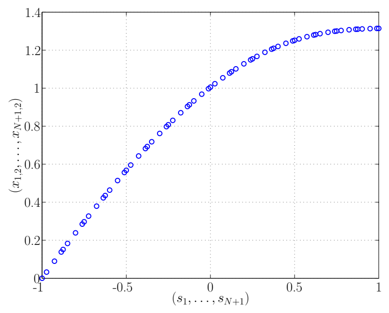
K	$N = 4K$	Derivative Code Generation Time Using *CADA* (s)	IPOPT Run Time Using *CADA* (s)	IPOPT Run Time Using *INTLAB* (s)	IPOPT Run Time Using *ADMAT* (s)	IPOPT Run Time Using *MAD* (s)
8	32	3.63	2.81	5.89	20.91	21.14
16	64	5.67	4.53	7.38	29.13	35.67
32	128	9.92	10.85	22.31	54.43	142.59
64	256	18.19	25.08	69.78	163.08	499.12
128	512	35.15	70.65	276.10	1088.24	25101.00

Table VI: Problem Size and Densities of Constraint Jacobian and Lagrangian Hessian Matrices for Example 4.

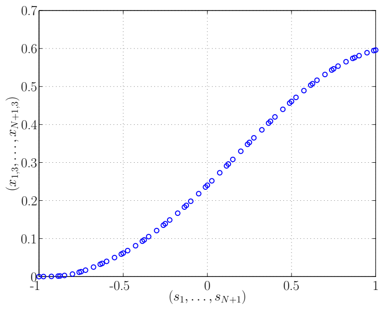
K	$N = 4K$	NLP Variables	NLP Constraints	Jacobian Non-Zeros	Jacobian Density (%)	Hessian Non-Zeros	Hessian Density (%)
8	32	263	192	1920	1.92	992	1.434
16	64	519	384	3840	1.92	1984	0.737
32	128	1031	768	7680	0.970	3968	0.373
64	256	2055	1536	15360	0.487	7936	0.188
128	512	4103	3072	30720	0.244	15872	0.094



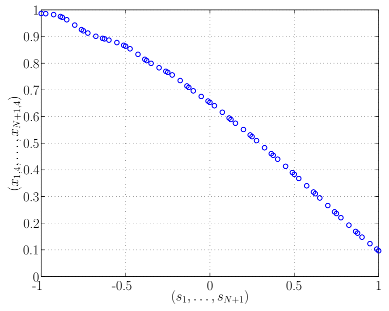
(a) $(x_{1,1},\dots,x_{N+1,1})$ vs. $(s_1,\dots,s_{N+1})$.



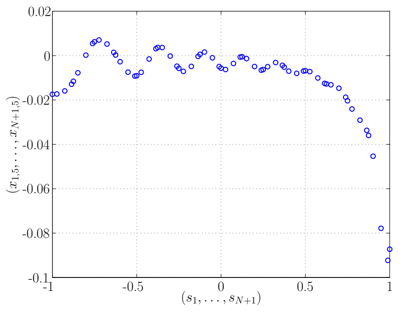
(b) $(x_{1,2},\dots,x_{N+1,2})$ vs. $(s_1,\dots,s_{N+1})$.



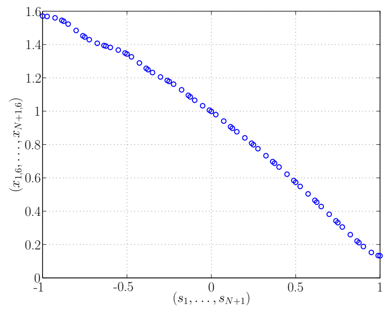
(c) $(x_{1,3},\dots,x_{N+1,3})$ vs. $(s_1,\dots,s_{N+1})$.



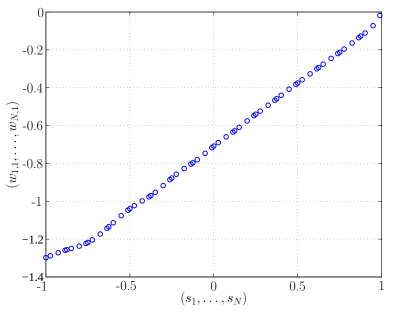
(d) $(x_{1,4},\dots,x_{N+1,4})$ vs. $(s_1,\dots,s_{N+1})$.



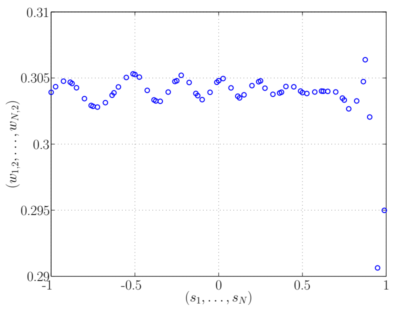
(e) $(x_{1,5},\dots,x_{N+1,5})$ vs. $(s_1,\dots,s_{N+1})$.



(f) $(x_{1,6},\dots,x_{N+1,6})$ vs. $(s_1,\dots,s_{N+1})$.



(g) $(w_{1,1},\dots,w_{N,1})$ vs. $(s_1,\dots,s_N)$.



(h) $(w_{1,2},\dots,w_{N,2})$ vs. $(s_1,\dots,s_N)$.

Fig. 14: Components of NLP Decision Vector for Example 4 with $K = 16$.

6. DISCUSSION

The four examples given in Sections 5 demonstrate three different aspects of *CADA*. Example 1 demonstrated the ability for *CADA* to avoid expression explosion that could possibly result in an exhaustion of memory by greatly reducing the number of variables created in the workspace when executing the final derivative file. In addition, it was seen in Example 1 that, despite the large number of function evaluations, the code produced by *CADA* was obtained and executed efficiently when compared against the MATLAB Symbolic Math Toolbox. Example 2 demonstrated the efficiency with which *CADA* generates the first derivative of a function whose Jacobian is sparse and compares the efficiency of the *CADA*-generated derivative code against the leading automatic differentiation software *INTLAB*, *ADMAT*, and *MAD*. Example 2 also demonstrated how *CADA* takes advantage of vectorization in the original function code by creating a vectorized derivative code that contains the same number of lines of code regardless of the value of n used to compute the original function. Finally, Example 2 demonstrated that *CADA* generates a sparse representation of the derivative, computing only the nonzero derivatives and storing only the indices corresponding to these non-zeros. Lastly, it was found in this second example that the *CADA* derivative code generation time grew quite slowly due to the fact that the derivative computations took advantage of vectorization and sparsity. Example 3 demonstrated the efficiency with which *CADA* generates the Hessian of a scalar function of a vector. This third example highlighted the fact that *CADA* is more efficient than *INTLAB* and is significantly more efficient than either *ADMAT* or *MAD*. This increased second derivative efficiency in *CADA* over the other automatic differentiation software is particularly noticeable as the function size gets large. Example 4 showed how *CADA* could be used on a potentially more practical problem of solving a large sparse nonlinear programming problem (NLP). This fourth example demonstrated not only the efficiency with which *CADA* can create and evaluate the first and second derivatives required by the NLP, but also highlighted the feature that *CADA* can generate the exact sparsity pattern for these required derivatives. It was also found in Example 4 that *CADA* compared favorably against two other well known and commonly used automatic differentiation tools.

7. LIMITATIONS OF *CADA*

While the examples show several key advantages of the current approach, we now discuss some limitations of the method. First, it is not possible in the present form to differentiate functions that contain conditional statements of *CADA* objects (for example, **if** and **while** statements) because the overloaded approach does not operate on numerical values of the input. Second, similar to the limitation pertaining to conditional statements, **for** loops with statements that include *CADA* objects can not be differentiated because the overloaded object does not contain numerical values. It is noted, however, that functions where a **for** loop does not contain a *CADA* object can be differentiated. In this latter case, the derivative function will not contain any **for** loops, but instead will produce a sequence of statements that executes each iteration of the loop. Finally, the derivative code generated using *CADA* depends upon the sizes of the variables specified when differentiating the function. Moreover, if it is desired to find derivatives of that same function using different sizes of the input variables, the function must be differentiated again using the new variable sizes, generating a new derivative code.

8. CONCLUSIONS

An object-oriented method has been described for computing derivatives of MATLAB functions. A new class called *CADA* has been developed. The derivatives of a general function can be obtained by evaluating a function of interest on a *CADA* object and writing the derivatives to a file that itself can be executed. The derivative file contains the same sequence of operations as the forward mode of automatic differentiation. Moreover, because the derivative function has the same input as the original function computer code, it is possible to apply the method recursively to generate higher-order derivatives. As a result, the method provides the ease of use associated with forward mode automatic differentiation while simultaneously providing the same functionality as would be obtained had the analytic derivatives been coded by hand. The key components of the method have been described in detail and the usability of the method was demonstrated on four examples, where each example exercised different aspects of the method. The method presented in this paper provides an efficient and reliable approach for computing analytic derivatives of general MATLAB functions.

REFERENCES

ABRAMOWITZ, M. AND STEGUN, I. 1965. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, New York.

BENSON, D. A., HUNTINGTON, G. T., THORVALDSEN, T. P., AND RAO, A. V. 2006. Direct trajectory optimization and costate estimation via an orthogonal collocation method. *Journal of Guidance, Control, and Dynamics 29,* 6, 1435–1440.

BETTS, J. T. 2009. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. SIAM Press, Philadelphia.

BIEGLER, L. T. AND ZAVALA, V. M. 2008. Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide optimization. *Computers and Chemical Engineering 33,* 3, 575–582.

BISCHOF, C. H., BÜCKER, H. M., LANG, B., RASCH, A., AND VEHRESCHILD, A. 2002. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*. IEEE Computer Society, Los Alamitos, CA, USA, 65–72.

BISCHOF, C. H., CARLE, A., CORLISS, G. F., GRIEWANK, A., AND HOVLAND, P. D. 1992. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming 1,* 1, 11–29.

BISCHOF, C. H., CARLE, A., KHADEMI, P., AND MAUER, A. 1996. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering 3,* 3, 18–32.

CAYUGA RESEARCH 2008–2009. *ADMAT: Automatic Differentiation Toolbox User's Guide, Version 2.0*. CAYUGA RESEARCH.

COLEMAN, T. F. AND VERMA, A. 1998a. *ADMAT: An Automatic Differentiation Toolbox for MATLAB. Technical Report*. Computer Science Department, Cornell University.

COLEMAN, T. F. AND VERMA, A. 1998b. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software 26,* 1, 150–175.

DARBY, C. L., HAGER, W. W., AND RAO, A. V. 2011a. Direct trajectory optimization using a variable low-order adaptive pseudospectral method. *Journal of Spacecraft and Rockets 48,* 3, 433–445.

DARBY, C. L., HAGER, W. W., AND RAO, A. V. 2011b. An hp–adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods 32,* 4, 476–502.

DOBMANN, M., LIEPELT, M., AND SCHITTKOWSKI, K. 1995. Algorithm 746: PCOMP: A Fortran code for automatic differentiation. *ACM Transactions on Mathematical Software 21,* 3, 233–266.

DUFF, I. S. 2004. MA57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Transactions on Mathematical Software 30,* 2, 118–144.

FORTH, S. A. 2006. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software 32,* 2, 195–222.

FORTH, S. A., TADJOUDDINE, M., PRYCE, J. D., AND REID, J. K. 2004. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand–coding. *ACM Transactions on Mathematical Software 30,* 4, 266–299.

GARG, D., HAGER, W. W., AND RAO, A. V. 2011a. Pseudospectral methods for solving infinite-horizon optimal control problems. *Automatica 47,* 4, 829–837.

GARG, D., PATTERSON, M. A., DARBY, C. L., FRANCOLIN, C., HUNTINGTON, G. T., HAGER, W. W., AND RAO, A. V. 2011b. Direct trajectory optimization and costate estimation of finite-horizon and infinite-horizon optimal control problems via a Radau pseudospectral method. *Computational Optimization and Applications 49,* 2, 335–358.

GARG, D., PATTERSON, M. A., HAGER, W. W., RAO, A. V., BENSON, D. A., AND HUNTINGTON, G. T. 2010. A unified framework for the numerical solution of optimal control problems using pseudospectral methods. *Automatica 46,* 11, 1843–1851.

GRIEWANK, A. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Frontiers in Appl. Mathematics*. SIAM Press, Philadelphia, Pennsylvania.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. Algorithm 755: ADOL-C, a package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software 22,* 2, 131–167.

HASCOËT, L. AND PASCUAL, V. 2004. TAPENADE 2.1 user's guide. Rapport Technique 300, INRIA, Sophia Antipolis.

KHARCHE, R. V. 2011. Matlab automatic differentiation using source transformation. Ph.D. thesis, Department of Informatics, Systems Engineering, Applied Mathematics, and Scientific Computing, Cranfield University.

KHARCHE, R. V. AND FORTH, S. A. 2006. Source transformation for MATLAB automatic differentiation. In *Computational Science – ICCS, Lecture Notes in Computer Science*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Vol. 3994. Springer, Heidelberg, Germany, 558–565.

LANTOINE, G., RUSSELL, R. P., AND DARGENT, T. 2012. Using multicomplex variables for automatic computation of high-order derivatives. *ACM Transactions on Mathematical Software 38,* 3, 16:1–16:21.

MARTINS, J. R. R. A., STURDZA, P., AND ALONSO, J. J. 2003. The complex-step derivative approximation. *ACM Transactions on Mathematical Software 29,* 3, 245–262.

MATHWORKS. 2010. *Version R2010b*. The MathWorks Inc., Natick, Massachusetts.

MONAGAN, M. B., GEDDES, K. O., HEAL, K. M., LABAHN, G., VORKOETTER, S. M., MCCARRON, J., AND DEMARCO, P. 2005. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada.

NEIDINGER, R. D. 2010. Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming. *SIAM REVIEW 52,* 3, 545–563.

PADULO, M., FORTH, S. A., , AND GUENOV, M. D. 2008. Robust aircraft conceptual design using automatic differentiation in matlab. In *Advances in Automatic Differentiation*, C. H. Bischof, H. M. BÄijcker, P. Hovland, U. Naumann, J. Utke, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, and T. Schlick, Eds. Lecture Notes in Computational Science and Engineering Series, vol. 64. Springer, Berlin, Heidelberg, 271–280.

PATTERSON, M. A. AND RAO, A. V. 2012. Exploiting sparsity in direct collocation pseudospectral methods for solving continuous-time optimal control problems. *Journal of Spacecraft and Rockets, 49,* 2, 364–377.

RAO, A. V., BENSON, D. A., CHRISTOPHER DARBY, M. A. P., FRANCOLIN, C., SANDERS, I., AND HUNTINGTON, G. T. 2010. Algorithm 902: GPOPS, A MATLAB software for solving multiple-phase optimal control problems using the Gauss pseudospectral method. *ACM Transactions on Mathematical Software 37,* 2, 22:1–22:39.

RAO, A. V., BENSON, D. A., DARBY, C. L., MAHON, B., FRANCOLIN, C., PATTERSON, M. A., SANDERS, I., AND HUNTINGTON, G. T. 2011. *User's Manual for GPOPS Version 4.x: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp–Adaptive Pseudospectral Methods*. http://www.gpops.org.

RUMP, S. M. 1999. Intlab – INTerval LABoratory. In *Developments in Reliable Computing*, T. Csendes, Ed. Kluwer Academic Publishers, Dordrecht, Germany, 77–104.

SHAMPINE, L. F. 2007. Accurate numerical derivatives in Matlab. *ACM Transactions on Mathematical Software 33,* 4, 26:1–26:17.

SPEELPENNING, B. 1980. Compiling fast partial derivatives of functions given by algorithms. Ph.D. thesis, University of Illinois at Urbana-Champaign.

TADJOUDDINE, M., FORTH, S. A., PRYCE, J. D., AND REID, J. K. 2002. Performance issues for vertex elimination methods in computing jacobians using automatic differentiation. In *Proceedings of the International Conference on Computational Science-Part II*. ICCS '02. Springer-Verlag, London, UK, 1077–1086.

WAECHTER, A. AND BIEGLER, L. T. 2006. On the implementation of a primal-dual interior-point filter line search algorithm for large-scale nonlinear programming. *Mathematical Programming 106,* 1, 575–582.

WOLFRAM. 2008. *Mathematica Edition: Version 7.0*. Wolfram Research, Inc., Champaign, Illinois.

A. COMPUTATION OF DIFFERENTIATION MATRIX IN EXAMPLE 4

The coefficients $D_{ik},\ (i = 1,\ldots,N,\ j = 1,\ldots,N+1)$ are defined as follows [Patterson and Rao 2012]. First, let s be defined on the domain $[-1,+1]$. Suppose further that $s \in [-1,+1]$ is partitioned into K mesh intervals $(S_1,\ldots,S_K)$, where $S_k = [s_{k-1}, s_k],\ (k = 1,\ldots,K)$ such that $s_0 = -1$ and $s_K = +1$. Within each mesh intervals $S_k,\ k = (1,\ldots,K)$, define the following basis of N_k+1 Lagrange polynomials [Patterson and Rao 2012]:

$$\ell_j^{(k)}(s) = \prod_{\substack{l=1 \\ l\neq j}}^{N_k+1} \frac{s - s_l^{(k)}}{s_j^{(k)} - s_l^{(k)}}, \quad k = 1,\ldots,K, \tag{20}$$

where $\left(s_1^{(k)},\ldots,s_{N_k}^{(k)}\right)$ are the N_k Legendre-Gauss-Radau [Abramowitz and Stegun 1965] (LGR) collocation points in mesh interval $S_k,\ (k = 1,\ldots,K)$. Next, within each mesh intervals k, let

$$D_{ij}^{(k)} = \left[\frac{d\ell_j^{(k)}(s)}{ds}\right]_{s_i^{(k)}}, \quad (i = 1,\ldots,N_k, \quad j = 1,\ldots,N_k+1, \quad k = 1,\ldots,K), \tag{21}$$

The matrix $D_{ik},\ \ (i = 1,\ldots,N,\ \ k = 1,\ldots,N+1)$ then has the following block structure that is comprised of the matrices defined in Eq. (21) such that the nonzero row-column indices in the k^{th} block of D_{ik} are given as $(\sum_{l=1}^{k-1} N_l + 1,\ldots,\sum_{l=1}^{k} N_l, \sum_{l=1}^{k-1} N_l + 1,\ldots,\sum_{l=1}^{k} N_l + 1)$, where for every mesh interval $k \in [1,\ldots,K]$ the nonzero elements are defined by the matrix given in Eq. (21). It is noted that the complete set of LGR points is given by $(s_1,\ldots,s_N)$, where $(s_1,\ldots,s_N) = \left((s_1^{(1)},\ldots,s_{N_1}^{(1)}),(s_1^{(2)},\ldots,s_{N_2}^{(2)}),\ldots,(s_1^{(K)},\ldots,s_{N_K}^{(K)})\right)$, and the parameter N is defined as

$$N = \sum_{k=1}^{K} N_k. \tag{22}$$

A schematic of the matrix $D_{ik},\ (i = 1,\ldots,N,\ k = 1,\ldots,N+1)$ is shown in Fig. 15.

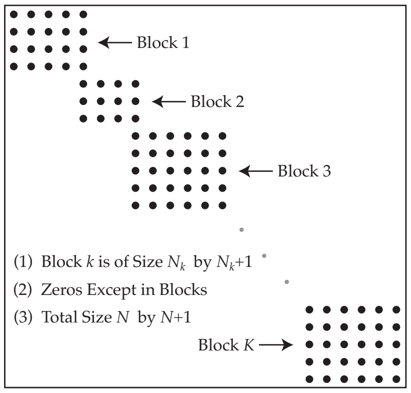

Block 1
Block 2
Block 3
(1) Block k is of Size N_k by N_k+1
(2) Zeros Except in Blocks
(3) Total Size N by N+1
Block K

Fig. 15: Structure of Matrix $D_{i,k},\ (i = 1,\ldots,N,\ k = 1,\ldots,N+1)$ for Example 3.

Received December 2012; revised May 2013; accepted August 2013