A Source Transformation via Operator Overloading Method for the Automatic Differentiation of Mathematical Functions in MATLAB

Matthew J. Weinstein[1]
Anil V. Rao[2]

University of Florida
Gainesville, FL 32611

A source transformation via operator overloading method is presented for computing derivatives of mathematical functions defined by MATLAB computer programs. The transformed derivative code that results from the method of this paper computes a sparse representation of the derivative of the function defined in the original code. As in all source transformation automatic differentiation techniques, an important feature of the method is that any flow control in the original function code is preserved in the derivative code. Furthermore, the resulting derivative code relies solely upon the native MATLAB library. The method is useful in applications where it is required to repeatedly evaluate the derivative of the original function. The approach is demonstrated on several examples and is found to be highly efficient when compared with well known MATLAB automatic differentiation programs.

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:
Weinstein, M. J. and Rao, A. V. 2014. A Combined Source Transformation and Operator Overloaded Method for Generating Derivative Programs of Mathematical Functions in MATLAB. ACM Trans. Math. Soft. V, N, Article A (January YYYY), 46 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Automatic differentiation, or as it has more recently been termed, algorithmic differentiation, (AD) is the process of determining accurate derivatives of a function defined by computer programs [Griewank 2008] using the rules of differential calculus. The objective of AD is to employ the rules of differential calculus in an algorithmic manner in order to efficiently obtain a derivative that is accurate to machine precision. AD exploits the fact that a computer program that contains an implementation of a mathematical function $y = f(x)$ can be decomposed into a sequence of elementary function

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. **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. J. Weinstein and A. V. Rao, Department of Mechanical and Aerospace Engineering, P.O. Box 116250, University of Florida, Gainesville, FL 32611-6250; e-mail: {mweinstein,anilvrao}@ufl.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 dependent output(s) with respect to the independent input(s) is encountered. The fundamental difference between forward and reverse modes is the direction in which the derivative calculations are performed. In the forward mode, the derivative calculations are performed from the dependent input variables of differentiation to the output independent variables of the program, while in reverse mode the derivative calculations are performed from the independent output variables of the program back to the dependent input variables.

Forward and reverse mode automatic differentiation methods are classically 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. Any object of the custom class typically contains properties that include the function value and derivatives of the object at a particular numerical value of the input. Furthermore, when any operation is performed on an object of the class, both function and derivative calculations are executed from within the overloaded operation. Well known implementations of forward and reverse mode AD that utilize operator overloading include *MXYZPTLK* [Michelotti 1991], *ADOL-C* [Griewank et al. 1996], *COSY INFINITY* [Berz et al. 1996], *ADOL-F* [Shiriaev and Griewank 1996], *FADBAD* [Bendtsen and Stauning 1996], *IMAS* [Rhodin 1997], *AD01* [Pryce and Reid 1998], *ADMIT-1* [Coleman and Verma 1998b], *ADMAT* [Coleman and Verma 1998a], *INTLAB* [Rump 1999], *FAD* [Aubert et al. 2001], *MAD* [Forth 2006], and *CADA* [Patterson et al. 2013]

In a source transformation approach, a function source code is transformed into a derivative source code, where, when evaluated, the derivative source code computes a desired derivative. An AD tool based on source transformation may be thought of as an AD preprocessor, consisting of both a compiler and a library of differentiation rules. As with any preprocessor, source transformation is achieved via four fundamental steps: parsing of the original source code, transformation of the program, optimization of the new program, and the printing of the optimized program. In the parsing phase, the original source code is read and transformed into a set of data structures which define the procedures and variable dependencies of the code. This information may then be used to determine which operations require a derivative computation, and the specific derivative computation may be found by means of the mentioned library. In doing so, the data representing the original program is augmented to include information on new derivative variables and the procedures required to compute them. This transformed information then represents a new derivative program, which, after an optimization phase, may be printed to a new derivative source code. While the implementation of a source transformation AD tool is much more complex than that of an operator overloaded tool, it usually leads to faster run-time speeds. Moreover, due to the fact that a source transformation tool produces source code, it may, in theory, be applied recursively to produce n^{th}-order derivative files, though Hessian symmetry may not be exploited. Well known implementations of forward and reverse mode AD that utilize source transformation include *DAFOR* [Berz 1987], *GRESS* [Horwedel 1991], *PADRE2* [Kubota 1991], *Odysée* [Rostaing-Schmidt 1993], *TAF* [Giering and Kaminski 1996], *ADIFOR* [Bischof et al. 1992; Bischof et al. 1996], *PCOMP* [Dobmann et al. 1995], *ADiMat* [Bischof et al. 2002], *TAPENADE* [Hascoët and Pascual 2004], *ELIAD* [Tadjouddine et al. 2003] and *MSAD* [Kharche and Forth 2006].

In recent years, *MATLAB* [Mathworks 2010] has become extremely popular as a platform for developing automatic differentiation tools. ADMAT/ADMIT [Coleman and

Verma 1998a; 1998b] was the first automatic differentiation program written in MATLAB. The *ADMAT/ADMIT* package utilizes operator overloading to implement both the forward and reverse modes to compute either sparse or dense Jacobians and Hessians. The next operator overloading approach was developed as a part of the *INTLAB* toolbox [Rump 1999], which implements the sparse forward mode to compute first and second derivatives. More recently, the package *MAD* [Forth 2006] has been developed. While *MAD* also employs operator overloading, unlike previously developed *MATLAB* AD tools, *MAD* utlizes the **derivvec** class to store directional derivatives within instances of the **fmad** class. In addition to operator overloaded methods that evaluate derivatives at a numeric value of the input argument, the hybrid source transfomation and operator overloaded package *ADiMat* [Bischof et al. 2003] has been developed. *ADiMat* employs source transformation to create a derivative source code. The derivative code may then be evaluated in a few different ways. If only a single directional derivative is desired, then the generated derivative code may be evaluated independently on numeric inputs in order to compute the derivative; this is referred to as the scalar mode. Thus, a Jacobian may be computed by a process known as strip mining, where each column of the Jacobian matrix is computed separately. In order to compute the entire Jacobian in a single evaluation of the derivative file, it is required to use either an overloaded derivative class or a collection of *ADiMat* specific run-time functions. Here it is noted that the derivative code used for both the scalar and overloaded modes is the same, but the generated code required to evaluate the entire Jacobian *without* overloading is slightly different as it requires that different *ADiMat* function calls be printed. The most recent *MATLAB* source transformation AD tool to be developed is *MSAD*, which was designed to test the benefits of using source transformation together with *MAD*'s efficient data structures. The first implementation of *MSAD* [Kharche and Forth 2006] was similar to the overloaded mode of *ADiMat* in that it utilized source transformation to generate derivative source code which could then be evaluated using the **derivvec** class developed for *MAD*. The current version of *MSAD* [Kharche 2012], however, does not depend upon operator overloading but still maintains the efficiencies of the **derivvec** class.

While the interpreted nature of MATLAB makes programming intuitive and easy, it also makes source transformation AD quite difficult. For example, the operation `c = a*b` takes on different meanings depending upon whether `a` or `b` is a scalar, vector, or matrix, and the differentiation rule is different in each case. *ADiMat* deals with such ambiguities differently depending upon which *ADiMat* specific run-time environment is being used. In both the scalar and overloaded modes, a derivative rule along the lines of `dc = da*b + a*db` is produced. Then, if evaluating in the scalar mode, `da` and `db` are numeric arrays of the same dimensions as `a` and `b`, respectively, and the expression `dc = da*b + a*db` may be evaluated verbatim. In the overloaded vector mode, `da` and `db` are overloaded objects, thus allowing the overloaded version of `mtimes` to determine the meaning of the `*` operator. In its non-overloaded vector mode, *ADiMat* instead produces a derivative rule along the lines of `dc = adimat_mtimes(da,a,db,b)`, where `adimat_mtimes` is a run-time function which distinguishes the proper derivative rule. Different from the vector modes of *AdiMat*, the most recent implementation of *MSAD* does not rely on an overloaded class or a separate run-time function to determine the meaning the `*` operator. Instead, *MSAD* utilizes shape and size propagation rules to attempt to determine the dimensions of `a` and `b`. In the event that the dimensions cannot be determined, *MSAD* places conditional statements on the dimensions of `a` and `b` directly within the derivative code, where each branch of the conditional statement contains the proper differentiation rule given the dimension information.

In this research a new method is described for automatic differentiation in MATLAB. The method performs source transformation via operator overloading and source reading techniques such that the resulting derivative source code can be evaluated using commands from only the native MATLAB library. The approach developed in this paper utilizes the recently developed operator overloaded method described in Patterson et al. [2013]. Different from traditional operator overloading in MATLAB where the derivative is obtained at a particular numeric value of the input, the method of Patterson et al. [2013] uses the forward mode of automatic differentiation to print to a file a derivative function, which, when evaluated, computes a sparse representation of the derivative of the original function. Thus, by evaluating a function program on the overloaded class, a reusable derivative program that depends solely upon the native MATLAB library is created. Different from a traditional source transformation tool, the method of [Patterson et al. 2013] requires that all object sizes be known, thus such ambiguities as the meaning of the $*$ operator are eliminated as the sizes of all variables are known at the time of code generation. This approach was shown to be particularly appealing for problems where the same user program is to be differentiated at a set of different numeric inputs, where the overhead associated with the initial overloaded evaluation becomes less significant with each required numerical evaluation of the derivative program.

The method of Patterson et al. [2013] is limited in that it cannot transform MATLAB function programs that contain flow control (that is, conditional, or iterative statements) into derivative programs containing the same flow control statements. Indeed, a key issue that arises in any source code generation technique is the ability to handle flow control statements that may be present in the originating program. In a typical operator overloaded approach, flow is dealt with during execution of the program on the particular instance of the class. That is, because any typical overloaded objects contain numeric function information, any flow control statements are evaluated in the same manner as if the input argument had been numeric. In a typical forward mode source transformation approach, flow control statements are simply copied over from the original program to the derivative program. In the method of Patterson et al. [2013], however, the differentiation routine has no knowledge of flow control nor is any numeric function information known at the time the overloaded operations are performed. Thus, a function code that contains conditional statements that depend upon the numeric values of the input cannot be evaluated on instances of the class. Furthermore, if an iterative statement exists in the original program, all iterations will be evaluated on overloaded objects and separate calculations corresponding to each iteration will be printed to the derivative file.

In this paper a new approach for generating derivative source code in MATLAB is described. The approach of this paper combines the previously developed overloaded **cada** class of Patterson et al. [2013] with source-to-source transformation in such a manner that any flow control in the original MATLAB source code is preserved in the derivative code. Two key aspects of the method are developed in order to allow for differentiation of programs that contain flow control. First, because the method of Patterson et al. [2013] is not cognizant of any flow control statements which may exist in a function program, it is neither possible to evaluate conditional statements nor is it possible to differentiate the iterations of a loop without unrolling the loop and printing each iteration of the loop to the resulting derivative file. In this paper, however, an intermediate source program is created where flow control statements are replaced by transformation routines. These transformation routines act as a pseudo-overloading of the flow control statements which they replace, thus enabling evaluation of the intermediate source program on instances of the **cada** class, where the control of the flow of the program is given to the transformation routines. Second, due to the sparse nature of the **cada** class, the result of any overloaded operation is limited to a single derivative sparsity pattern, where different sparsity patterns may arise depending upon conditional branches or loop iterations. This second issue is similar to the issues experienced when applying the vertex elimination tool *ELIAD* to *Fortran* programs containing conditional branches. The solution using *ELIAD* was to determine the union of derivative sparsity patterns across each conditional branch [Tadjouddine et al. 2003]. Similarly, in this paper, an overloaded union operator is developed which is used to determine the union of the derivative sparsity patterns that are generated by all possible conditional branches and/or loop iterations in the original source code, thus making it possible to print derivative calculations that are valid for all possible branches/loop iterations.

This paper is organized as follows. In Section 2 the notation and conventions used throughout the paper are described. In Section 3 brief reviews are provided of the previously developed *CADA* differentiation method and **cada** class. In Section 4 the concepts of overloaded unions and overmapped objects are described. In Section 5 a detailed explanation is provided of the method used to perform user source to derivative source transformation of user functions via the **cada** class. In Section 6 four examples are given to demonstrate the capabilities of the proposed method and to compare the method against other well known MATLAB AD tools. In Section 7 a discussion is given of the results obtained in Section 6. Finally, in Section 8 conclusions on our work are given.

2. NOTATION AND CONVENTIONS

In this paper we employ the following notation. First, without loss of generality, consider a vector function of a vector $\mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, where a vector is denoted by a lower-case bold letter. Thus, if $\mathbf{x} \in \mathbb{R}^n$, then $\mathbf{x}$ and $\mathbf{f}(\mathbf{x})$ are column vectors with the following forms, respectively:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n, \tag{1}$$

Consequently, $\mathbf{f}(\mathbf{x})$ has the form

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_m(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^m, \tag{2}$$

where $x_i,\ (i = 1, \ldots, n)$ and $f_j(\mathbf{x}),\ (j = 1, \ldots, m)$ are, respectively, the elements of $\mathbf{x}$ and $\mathbf{f}(\mathbf{x})$. The *Jacobian* of the vector function $\mathbf{f}(\mathbf{x})$, denoted $\mathbf{Jf}(\mathbf{x})$, is then an $m \times n$ matrix

$$\mathbf{Jf}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_m} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \in \mathbb{R}^{m \times n}. \tag{3}$$

Assuming the Jacobian $\mathbf{Jf}(\mathbf{x})$ contains N_{nz} non-zero elements, we denote $\mathbf{i}_{\mathbf{x}}^{\mathbf{f}} \in \mathbb{Z}_+^{N_{nz}}$, $\mathbf{j}_{\mathbf{x}}^{\mathbf{f}} \in \mathbb{Z}_+^{N_{nz}}$, to be the row and column locations of the non-zero elements of $\mathbf{Jf}(\mathbf{x})$. Furthermore, we denote $\mathbf{d}_{\mathbf{x}}^{\mathbf{f}} \in \mathbb{R}^{N_{nz}}$ to be the non-zero elements of $\mathbf{Jf}(\mathbf{x})$ such that

$$d_{\mathbf{x}}^{\mathbf{f}}(k) = \frac{\partial f_{[i_{\mathbf{x}}^{\mathbf{f}}(k)]}}{\partial x_{[j_{\mathbf{x}}^{\mathbf{f}}(k)]}}, \quad (k = 1 \ldots N_{nz}), \tag{4}$$

where $d_{\mathbf{x}}^{\mathbf{f}}(k)$, $i_{\mathbf{x}}^{\mathbf{f}}(k)$, and $j_{\mathbf{x}}^{\mathbf{f}}(k)$ refer to the k^{th} elements of the vectors $\mathbf{d}_{\mathbf{x}}^{\mathbf{f}}$, $\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}$, and $\mathbf{j}_{\mathbf{x}}^{\mathbf{f}}$, respectively.

Because the method described in this paper performs both the analysis and overloaded evaluation of source code, it is necessary to develop conventions and notation for these processes. First, MATLAB variables will be denoted using typewriter text (for example, `y`). Next, when referring to a MATLAB source code fragment, an upper-case non-bold face text will be used (for example, A), where a subscript may be added in order to distinguish between multiple similar code fragments (for example, A_i, $A_{i,j}$). Then, given a MATLAB source code fragment A of a program P, the fragments of code that are evaluated before and after the evaluation of A are denoted $Pred(A)$ and $Succ(A)$, respectively, where $Pred(A)$, A, and $Succ(A)$ are mutually disjoint sets such that $Pred(A) \cap A = A \cap Succ(A) = Pred(A) \cap Succ(A) = \emptyset$ and $Pred(A) \cup A \cup Succ(A) = P$. Next, any overloaded object will be denoted using a calligraphic character (for example, $\mathcal{Y}$). Furthermore, an overloaded object that is assigned to a variable is referred to as an *assignment object*, while an overloaded object that results from an overloaded operation but is not assigned to a variable is referred to as an *intermediate object*. It is noted that an intermediate object is not necessarily the same as an *intermediate variable* as defined in Griewank [2008]. Instead, intermediate objects as defined in this paper are the equivalent of statement-level intermediate variables. To clarify, consider the evaluation of the line of MATLAB code `y = sin(x) + x` evaluated on the overloaded object $\mathcal{X}$. This evaluation will first result in the creation of the intermediate object $\mathcal{V} = \sin(\mathcal{X})$ followed by the assignment object $\mathcal{Y} = \mathcal{V} + \mathcal{X}$, where $\mathcal{Y}$ is then assigned to the variable `y`.

3. REVIEW OF THE *CADA* DIFFERENTIATION METHOD

Patterson et al. [2013] describes a forward mode operator overloading method for transforming a mathematical MATLAB program into a new MATLAB program, which, when evaluated, computes the non-zero derivatives of the original program. The method of Patterson et al. [2013] relies upon a MATLAB class called **cada**. Unlike conventional operator overloading methods that operate on numerical values of the input argument, the *CADA* differentiation method does not store numeric function and derivative values. Instead, instances of the **cada** class store only the size of the function, non-zero derivative locations, and symbolic identifiers. When an overloaded operation is called on an instance of the **cada** class, the proper function and non-zero derivative calculations are printed to a file. It is noted that the method of Patterson et al. [2013] was developed for MATLAB functions that contain no flow control statements and whose derivative sparsity pattern is fixed (that is, the nonzero elements of the Jacobian of the function are the same on each call to the function). In other words, given a MATLAB program containing only a single basic block and with fixed input sizes and derivative sparsity patterns, the method of Patterson et al. [2013] can generate a MATLAB code that contains the statements that compute the non-zero derivative of the original function. In this section we provide a brief description of a slightly modified version of the **cada** class and the method used by overloaded operations to print derivative calculations to a file.

3.1. The cada Class

During the evaluation of a user program on **cada** objects, a derivative file is simultaneously being generated by storing only sizes, symbolic identifiers, and sparsity patterns within the objects themselves and writing the computational steps required to compute derivatives to a file. Without loss of generality, we consider an overloaded object $\mathcal{Y}$, where it is assumed that there is a single independent variable of differentiation,

$\mathbf{x}$. The properties of such an object are given in Table I. Assuming that the overloaded object $\mathcal{Y}$ has been created, then the proper calculations will have been printed to a file to compute both the value of $\mathbf{y}$ and the non-zero derivatives $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$ (if any) of the Jacobian $\mathbf{J}\mathbf{y}(\mathbf{x})$. In this paper we refer to the printed variable which is assigned the value of $\mathbf{y}$ as a *function variable* and the printed variable which is assigned the value of $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$ as a *derivative variable*. We also make the distinction between *strictly symbolic* and not strictly symbolic function variables, where, if a function variable is strictly symbolic, at least one element of the function variable is unknown at the time of code generation. Thus, a function variable which is *not* strictly symbolic is considered to be a constant with no associated derivative (where constants are propagated within overloaded operations and calculations are still printed to file). Furthermore, it is noted that the value assigned to any derivative variable may be mapped into a two-dimensional Jacobian using the row and column indices stored in the `deriv.nzlocs` field of the corresponding overloaded object.

Table I: Properties of an Overloaded Object $\mathcal{Y}$. Each overloaded object has the fields `id`, `func` and `deriv`. The `func` field contains information of the *function variable* which is assigned the value of $\mathbf{y}$ in the generated program, and the `deriv` field contains information on the derivative variable which is assigned the value of $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$ in the generated program (where $\mathbf{x}$ is the independent variable of differentiation).

`id`	unique integer value that identifies the object	
`func`	structure containing the following information on the function variable associated with this object:	
	`name`	string representation of function variable
	`size`	1×2 array containing the dimensions of $\mathbf{y}$
	`zerolocs`	$N_z \times 1$ array containing the linear index of any known zero locations of $\mathbf{y}$, where N_z is the number of known zero elements - this field is only used if the function variable is strictly symbolic.
	`value`	if the function variable is *not* strictly symbolic, then this field contains the values of each element of $\mathbf{y}$
`deriv`	structure array containing the following information on the derivative variable associated with this object:	
	`name`	string representation of what the derivative variable is called
	`nzlocs`	$N_{nz} \times 2$ array containing the row and column indices, $\mathbf{i}_{\mathbf{x}}^{\mathbf{y}} \in \mathbb{Z}_+^{N_{nz}}$ and $\mathbf{j}_{\mathbf{x}}^{\mathbf{y}} \in \mathbb{Z}_+^{N_{nz}}$, of any possible non-zero entries of the Jacobian $\mathbf{J}\mathbf{y}(\mathbf{x})$

3.2. Example of the *CADA* Differentiation Method

The **cada** class utilizes forward-mode algorithmic differentiation to propagate derivative sparsity patterns while performing overloaded evaluations on a section of user written code. All standard unary mathematical functions (for example, polynomial, trigonometric, exponential, etc.), binary mathematical functions (for example, plus, minus, times, etc.), and organizational functions (for example, reference, assignment, concatenation, transpose, etc.) are overloaded. These overloaded functions result in the appropriate function and non-zero derivative calculations being printed to a file while simultaneously determining the associated sparsity pattern of the derivative function.

In this section the *CADA* differentiation method is explained by considering a MATLAB function such that $\mathbf{x} \in \mathbb{R}^n$ is the input and is the variable of differentiation. Let $\mathbf{v}(\mathbf{x}) \in \mathbb{R}^m$ contain $N_{nz} \leq m \times n$ non-zero elements in the Jacobian, $\mathbf{Jv}(\mathbf{x})$, and let $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}} \in \mathbb{R}^{N_{nz}}$ and $\mathbf{j}_{\mathbf{x}}^{\mathbf{v}} \in \mathbb{R}^{N_{nz}}$ be the row and column indices, respectively, corresponding to the non-zero elements of $\mathbf{Jv}(\mathbf{x})$. Furthermore, let $\mathbf{d}_{\mathbf{x}}^{\mathbf{v}} \in \mathbb{R}^{N_{nz}}$ be a vector of the non-zero elements of $\mathbf{Jv}(\mathbf{x})$. Assuming no zero function locations are known in $\mathbf{v}$, the **cada** instance, $\mathcal{V}$, would posses the following relevant properties: (i) `func.name='v.f'`; (ii) `func.size=`$[m \;\; 1]$; (iii) `deriv.name='v.dx'`; and (iv) `deriv.nzlocs=`$[\mathbf{i}_{\mathbf{x}}^{\mathbf{v}} \;\; \mathbf{j}_{\mathbf{x}}^{\mathbf{v}}]$. It is assumed that, since the object $\mathcal{V}$ has been created, the function variable, `v.f`, and the derivative variable, `v.dx`, have been printed to a file in such a manner that, upon evaluation, `v.f` and `v.dx` will be assigned the numeric values of $\mathbf{v}$ and $\mathbf{d}_{\mathbf{x}}^{\mathbf{v}}$, respectively. Suppose now that the unary array operation $g : \mathbb{R}^m \rightarrow \mathbb{R}^m$ (for example, `sin`, `sqrt`, etc.) is encountered during the evaluation of the MATLAB function code. If we consider the function $\mathbf{w} = g(\mathbf{v}(\mathbf{x}))$, it can easily be seen that $\mathbf{Jw}(\mathbf{x})$ and $\mathbf{Jv}(\mathbf{x})$ will contain possible non-zero elements in the same locations. Additionally, the non-zero derivative values $\mathbf{d}_{\mathbf{x}}^{\mathbf{w}}$ may be calculated as

$$d_{\mathbf{x}}^{\mathbf{w}}(k) = g'(v_{[i_{\mathbf{x}}^{\mathbf{v}}(k)]}) d_{\mathbf{x}}^{\mathbf{v}}(k), \quad (k = 1 \ldots N_{nz}), \tag{5}$$

where $g'(\cdot)$ is the derivative of $g(\cdot)$ with respect to the argument of the function g. In the file that is being created, the derivative variable, `w.dx`, would be written as follows. Assume that the index $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}$ is printed to a file such that it will be assigned to the variable `findex` within the derivative program. Then the derivative computation would be written as `w.dx = dg(v.f(findex)).*v.dx`, and the function computation would simply be written as `w.f = g(v.f)`, where `dg`$(\cdot)$ and `g`$(\cdot)$ represent the MATLAB operations corresponding to $g'(\cdot)$ and $g(\cdot)$, respectively. The resulting overloaded object, $\mathcal{W}$, would then have the same properties as $\mathcal{V}$ with the exception of `id`, `func.name`, and `deriv.name`. Here we emphasize that, in the above example, the derivative calculation is only valid for a vector, $\mathbf{v}$, whose non-zero elements of $\mathbf{Jv}(\mathbf{x})$ lie in the row locations defined by $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}$, and that the values of $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}$ and m are known at the time of the overloaded operation.

3.3. Motivation for a New Method to Generate Derivative Source Code

The aforementioned discussion identifies the fact that the *CADA* method as developed in Patterson et al. [2013] may be used to transform a mathematical program containing a simple basic block into a derivative program, which, when executed in MATLAB, will compute the non-zero derivatives of a fixed input size version of the original program. The method may also be applied to programs containing unrollable loops, where, if evaluated on **cada** objects, the resulting derivative code would contain an unrolled representation of the loop. Function programs containing indeterminable conditional statements (that is, conditional statements which are dependent upon strictly symbolic objects), however, cannot be evaluated on instances of the **cada** class as multiple branches may be possible. The remainder of this paper is focused on developing the methods to allow for the application of the **cada** class to differentiate function programs containing indeterminable conditional statements and loops, where the flow control of the originating program is preserved in the derivative program.

4. OVERMAPS AND UNIONS

Before proceeding to the description of the method, it is important to describe an *overmap* and a *union*, both which are integral parts of the method itself. Specifically, due to the nature of conditional blocks and loop statements, it is often the case that different assignment objects may be written to the same variable, where the determination of which object is assigned depends upon which conditional branch is taken

Table II: Overloaded Variable Properties for Overloaded Union Example.

Object Property	$\mathcal{U}$	$\mathcal{V}$	$\mathcal{W} = \mathcal{U} \cup \mathcal{V}$
function size:	m_u	m_v	m_w
possible function non-zero locations:	$\mathbf{i}^{\mathbf{u}}$	$\mathbf{i}^{\mathbf{v}}$	$\mathbf{i}^{\mathbf{w}}$
possible derivative non-zero locations:	$[\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}}]$	$[\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}}]$	$[\mathbf{i}^{\mathbf{w}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{w}}_{\mathbf{x}}]$

or which iteration of the loop is being evaluated. The issue is that any overloaded operation performed on such a variable may print different derivative calculations depending upon which object is assigned to the variable. In order to print calculations that are valid for all objects which may be assigned to the variable (that is, all of the variable's immediate predecessors), a **cada** *overmap* is assigned to the variable, where an overmap has the following properties:

- Function Size: The function row/column size is the maximum row/column size of all possible row/column sizes.
- Function Sparsity: The function is only considered to have a known zero element if every possible function is known to have same known zero element.
- Derivative Sparsity: The Jacobian is only considered to have a known zero element if every possible Jacobian has the same known zero element.

Furthermore, the overmap is defined as the *union* of all possible objects that may be assigned to the variable.

Example of an Overloaded Union. In order to illustrate the concept of the union of two overloaded objects, consider two configurations of the same variable, $\mathbf{y} = \mathbf{u}(\mathbf{x}) \in \mathbb{R}^{m_u}$ or $\mathbf{y} = \mathbf{v}(\mathbf{x}) \in \mathbb{R}^{m_v}$, where $\mathbf{x} \in \mathbb{R}^n$. Suppose further that $\mathcal{U}$ and $\mathcal{V}$ are **cada** instances that possess the properties shown in Table II, and that $\mathcal{W} = \mathcal{U} \cup \mathcal{V}$ is the union of $\mathcal{U}$ and $\mathcal{V}$. Then the size property of $\mathcal{W}$ will be $m_w = \max(m_u, m_v)$. The non-zero function locations of $\mathcal{W}$ are then determined as follows. Let $\bar{\mathbf{u}}$ and $\bar{\mathbf{v}}$ be vectors of length m_w, where

$$\begin{aligned} \bar{\mathbf{u}}_i &= \begin{cases} 1, & i \in \mathbf{i}^{\mathbf{u}}, \\ 0, & \text{otherwise}, \end{cases} \\ & \qquad\qquad\qquad i = 1, \ldots, m_w. \\ \bar{\mathbf{v}}_i &= \begin{cases} 1, & i \in \mathbf{i}^{\mathbf{v}}, \\ 0, & \text{otherwise}, \end{cases} \end{aligned} \tag{6}$$

The possible non-zero function locations of $\mathcal{W}$, $\mathbf{i}^{\mathbf{w}}$, are then defined by the non-zero locations of $\bar{\mathbf{w}} = \bar{\mathbf{u}} + \bar{\mathbf{v}}$.

Next, the locations of all possible non-zero *derivatives* of $\mathcal{W}$ can be determined in a manner similar to the approach used to determine the non-zero function locations of $\mathcal{W}$. Specifically, let $\bar{\mathbf{U}}^{\mathbf{x}}$, $\bar{\mathbf{V}}^{\mathbf{x}}$ be $m_w \times m_x$ matrices whose elements are given as

$$\begin{aligned} \bar{U}^{\mathbf{x}}_{i,j} &= \begin{cases} 1, & (i,j) \in [\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}}], \\ 0, & \text{otherwise}, \end{cases} \\ & \qquad\qquad\qquad i = 1, \ldots, m_w, \quad j = 1, \ldots, m_x. \\ \bar{V}^{\mathbf{x}}_{i,j} &= \begin{cases} 1, & (i,j) \in [\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}}], \\ 0, & \text{otherwise}, \end{cases} \end{aligned} \tag{7}$$

Finally, suppose we let $\bar{\mathbf{W}}^{\mathbf{x}} = \bar{\mathbf{U}}^{\mathbf{x}} + \bar{\mathbf{V}}^{\mathbf{x}}$. Then the possible non-zero derivative locations of $\mathcal{W}$, $[\mathbf{i}^{\mathbf{x}}_{\mathbf{w}}, \mathbf{j}^{\mathbf{x}}_{\mathbf{w}}]$, are defined to be the row and column indices corresponding to the non-zero locations of the matrix $\bar{\mathbf{W}}^{\mathbf{x}}$.

5. SOURCE TRANSFORMATION VIA THE OVERLOADED *CADA* CLASS

A new method is now described for generating derivative files of mathematical functions implemented in MATLAB, where the function source code may contain flow control statements. Source transformation on such function programs is performed using the overloaded **cada** class together with unions and overmaps as described in Section 4. That is, all function/derivative computations are printed to the derivative file as described in Section 3 while flow control is preserved by performing overloaded unions where code fragments join, for example, on the output of conditional fragments, on the input of loops, etc. The method thus has the feature that the resulting derivative code depends solely on the functions from the native MATLAB library (that is, derivative source code generated by the method does *not* depend upon overloaded statements or separate run-time functions). Furthermore, the structure of the flow control statements is transcribed to the derivative source code. In this section we describe in detail the various processes that are used to carry out the source transformation. An outline of the source transformation method is shown in Fig. 1.

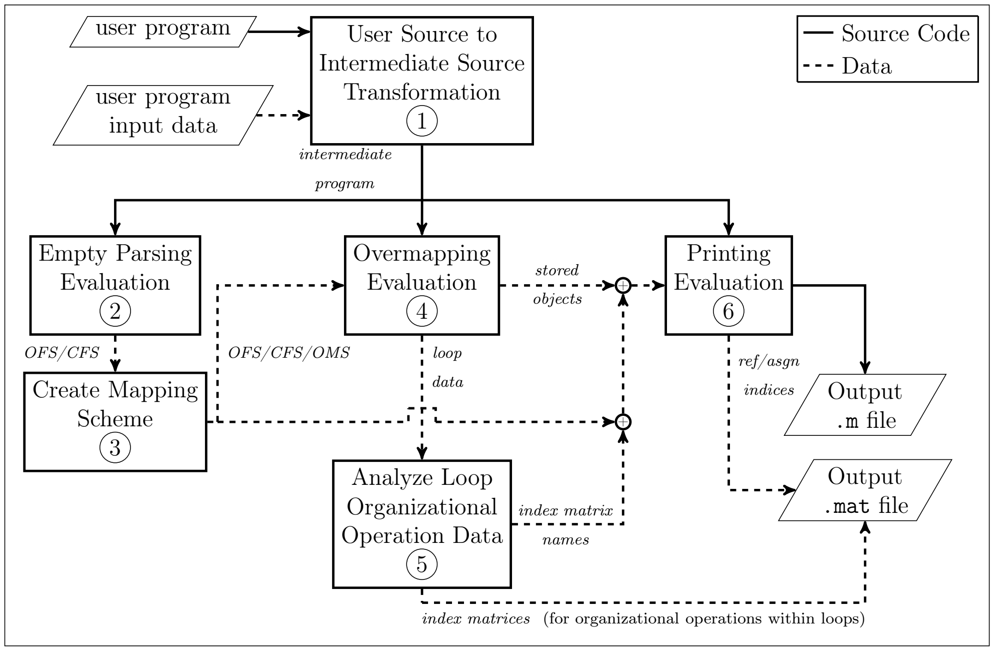


Fig. 1: Source Transformation via Operator Overloading Process

From Fig. 1 it is seen that the inputs to the transformation process are the user program to be differentiated together with the information required to create **cada** instances of the inputs to the user program. Using Fig. 1 as a guide, the source transformation process starts by performing a transformation of the original source code to an intermediate source code (process ①). This initial transformation then results in a source code on which an overloaded analysis may be performed. Automatic differentiation is then effected by performing *three* overloaded evaluations of the intermediate source code. During the first evaluation (process ②), a record of all objects and locations relative to flow control statements is built to form an *object flow structure* (OFS) and a *control flow structure* (CFS), but no derivative calculations are performed.

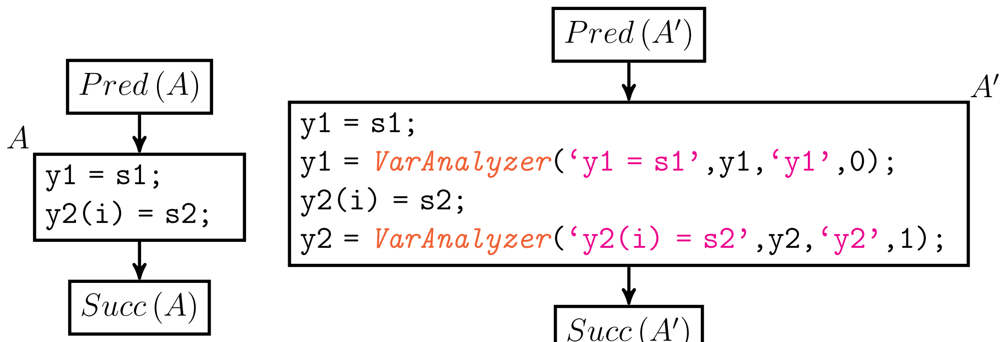
Next, an *object mapping scheme* (OMS) is created (process ③) using the DFS and CFS obtained from the first evaluation, where the OMS defines where overloaded unions must be performed and where overloaded objects must be saved. During the second evaluation (process ④), the intermediate source code is evaluated on **cada** instances, where each overloaded **cada** operation does not print any calculations to a file. During this second evaluation, overmaps are built and are stored in global memory (shown as *stored objects* output of process ④) while data is collected regarding any organizational operations contained within loops (shown as *loop data* output of process ④). The organizational operation data is then analyzed to produce special derivative mapping rules for each operation (process ⑤). During the third evaluation of the intermediate source program (process ⑥), all of the data produced from processes ②–⑤ is used to print the final derivative program to a file. In addition, a MATLAB binary file is written that contains the reference and assignment indices required for use in the derivative source code. The details of the steps required to transform a function program containing no function/sub-function calls into a derivative program are given in Sections 5.1–5.6 and correspond to the processes ①–⑥, respectively, shown in Fig. 1. Section 5.7 then shows how the methods developed in Sections 5.1–5.6 are applied to programs containing multiple function calls.

5.1. User Source-to-Intermediate-Source Transformation

The first step in generating derivative source code is to perform source-to-source transformation on the original program to create an intermediate source code, where the intermediate source code is an augmented version of the original source code that contains calls to transformation routines. The resulting intermediate source code may then be evaluated on overloaded objects to effectively analyze and apply AD to the program defined by the original user code. This initial transformation is performed via a purely lexical analysis within MATLAB, where first the user code is parsed line by line to determine the location of any flow control keywords (that is, `if`, `elseif`, `else`, `for`, `end`). The code is then read again, line by line, and the intermediate program is printed by copying sections of the user code and applying different augmentations at the statement and flow control levels.

Figure 2 shows the transformation of a basic block in the original program, A, into the basic block of the intermediate program, A'. At the basic block level, it is seen that each user assignment is copied exactly from the user program to the intermediate program, but is followed by a call to the transformation routine *VarAnalyzer*. It is also seen that, after any object is written to a variable, the variable analyzer is provided the assignment object, the string of code whose evaluation resulted in the assignment object, the name of the variable to which the object was written, and a flag stating whether or not the assignment was an array subscript assignment. After each assignment object is sent to the *VarAnalyzer*, the corresponding variable is immediately rewritten on the output of the *VarAnalyzer* routine. By sending all assigned variables to the variable analyzer it is possible to distinguish between assignment objects and intermediate objects and determine the names of the variables to which each assignment object is written. Additionally, by rewriting the variables on the output of the variable analyzer, full control over the overloaded workspace (that is, the collection of all variables active in the intermediate program) is given to the source transformation algorithm. Consequently, all variables (and any operations performed on them) are forced to be overloaded.

Figure 3 shows the transformation of the k^{th} conditional fragment encountered in the original program to a transformed conditional fragment in the intermediate program. In the original conditional fragment, C_k, each branch contains the code fragment $B_{k,i}$ ($i = 1, \ldots, N_k$), where the determination of which branch is evaluated depends



```
y1 = s1;
y2(i) = s2;
```

```
y1 = s1;
y1 = VarAnalyzer('y1 = s1',y1,'y1',0);
y2(i) = s2;
y2 = VarAnalyzer('y2(i) = s2',y2,'y2',1);
```

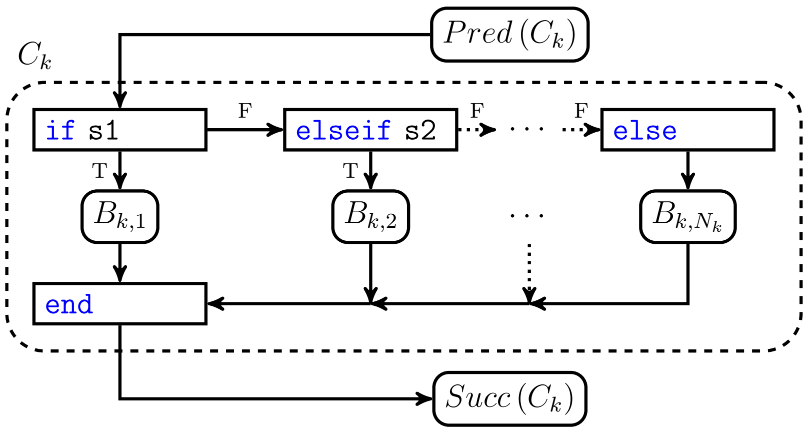
(a) Original Basic Block (b) Transformed Basic Block

Fig. 2: Transformation of User Source Basic Block A to Intermediate Source Basic Block A'. The quantities `s1` and `s2` represent generic MATLAB expressions.

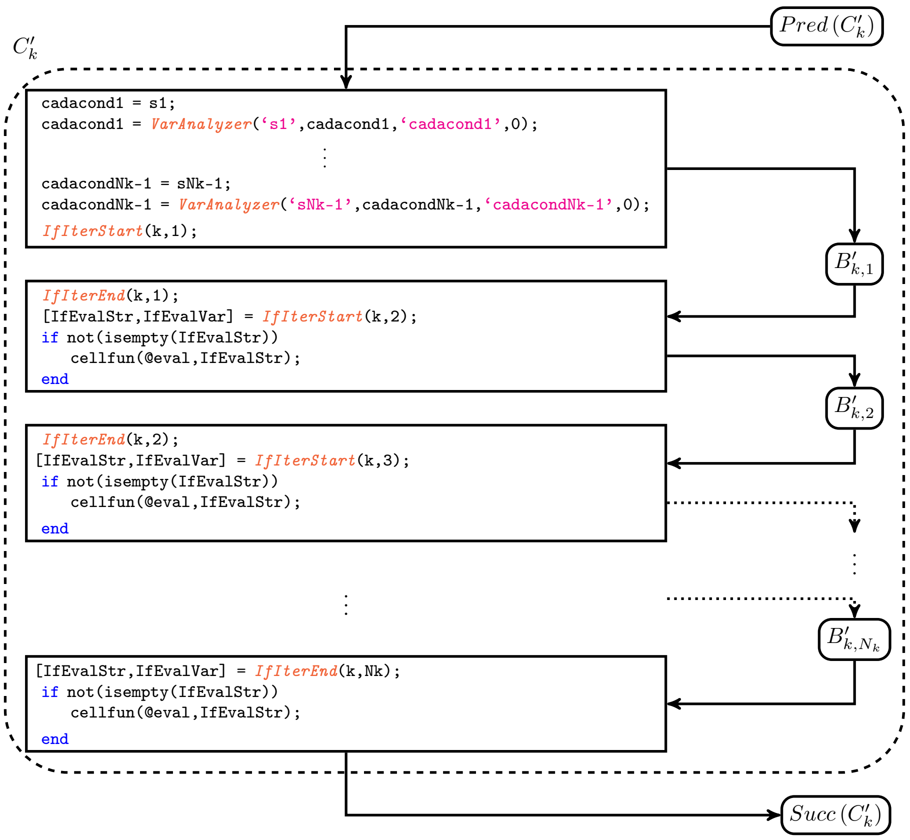
upon the logical values of the statements `s1`,`s2`,$\cdots$,`sNk-1`. In the transformed conditional fragment, it is seen that no flow control surrounds any of the branches. Instead, all conditional variables are evaluated and sent to the *IfInitialize* transformation routine. Then the transformed branch fragments $B'_{k,1},\cdots,B'_{k,N_k}$ are evaluated in a linear manner, with a call to the transformation routines *IfIterStart* and *IfIterEnd* before and after the evaluation of each branch fragment. By replacing all conditional statements with transformation routines, the control of the flow is given to the transformation routines. When the intermediate program is evaluated, the *IfInitialize* routine can determine whether each conditional variable returns true, false, or indeterminate, and then the *IfIterStart* routine can set different global flags prior to the evaluation of each branch in order to emulate the conditional statements. As the overloaded analysis of each branch $B'_{k,i}$ is performed in a linear manner, it is important to ensure that each branch is analyzed independent of the others. Thus, prior to any `elseif`/`else` branch the overloaded workspace may be modified using the outputs, `IfEvalStr` and `IfEvalVar`, of the *IfIterStart* transformation routine. Furthermore, following the overloaded evaluation and analysis of the conditional fragment C'_k, it is often the case that overmapped outputs must be brought into the overloaded workspace. Thus, the final *IfIterEnd* routine has the ability to modify the workspace via its outputs, `IfEvalStr` and `IfEvalVar`.

Figure 4 shows an original loop fragment, L_k, and the corresponding transformed loop fragment L'_k, where k is the k^{th} loop encountered in the original program. Similar to the approach used for conditional statements, Fig. 4 shows that control over the flow of the loop is given to the transformation routines. Transferring control to the transformation routines is achieved by first evaluating the loop index expression and feeding the result to the *ForInitialize* routine. The loop is then run on the output, `adigatorForVar_k`, of the *ForInitialize* routine, with the loop variable being calculated as a reference on each iteration. Thus the *ForInitialize* routine has the ability to unroll the loop for the purposes of analysis, or to evaluate the loop for only a single iteration. Furthermore, the source transformation algorithm is given the ability to modify the inputs and outputs of the loop. This modification is achieved via the outputs, `ForEvalStr` and `ForEvalVar`, of the transformation routines *ForInitialize* and *ForIterEnd*.

It is important to note that the code fragments $B_{k,i}$ of Fig. 3 and I_k of Fig. 4 do not necessarily represent basic blocks, but may themselves contain conditional fragments



(a) Original Conditional Fragment



```
cadacond1 = s1;
cadacond1 = VarAnalyzer('s1',cadacond1,'cadacond1',0);
                          ⋮
cadacondNk-1 = sNk-1;
cadacondNk-1 = VarAnalyzer('sNk-1',cadacondNk-1,'cadacondNk-1',0);
IfIterStart(k,1);
```

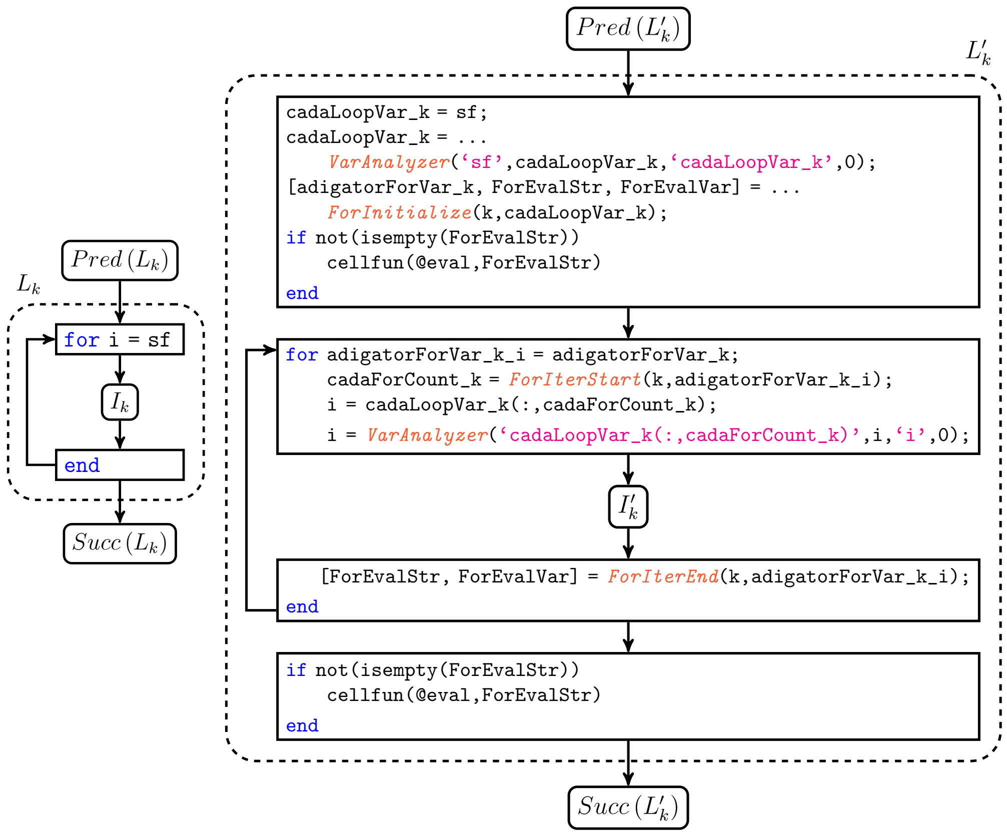
```
IfIterEnd(k,1);
[IfEvalStr,IfEvalVar] = IfIterStart(k,2);
if not(isempty(IfEvalStr))
   cellfun(@eval,IfEvalStr);
end
```

```
IfIterEnd(k,2);
[IfEvalStr,IfEvalVar] = IfIterStart(k,3);
if not(isempty(IfEvalStr))
   cellfun(@eval,IfEvalStr);
end
```

```
[IfEvalStr,IfEvalVar] = IfIterEnd(k,Nk);
if not(isempty(IfEvalStr))
   cellfun(@eval,IfEvalStr);
end
```

(b) Transformed Conditional Fragment

Fig. 3: Transformation of User Source Conditional Fragment C_k to Intermediate Source Conditional Fragment C'_k. Here, C_k is the k^{th} conditional fragment encountered in the user program, $B_{k,i}$ $(i = 1, \ldots, N_k)$ are the fragments of code contained within each branch of the conditional fragment C_k, and the quantities `s1`, `s2`, `sNk-1` represent MATLAB logical expressions.



```
for i = sf
end
```

```
cadaLoopVar_k = sf;
cadaLoopVar_k = ...
    VarAnalyzer('sf',cadaLoopVar_k,'cadaLoopVar_k',0);
[adigatorForVar_k, ForEvalStr, ForEvalVar] = ...
    ForInitialize(k,cadaLoopVar_k);
if not(isempty(ForEvalStr))
    cellfun(@eval,ForEvalStr)
end
```

```
for adigatorForVar_k_i = adigatorForVar_k;
    cadaForCount_k = ForIterStart(k,adigatorForVar_k_i);
    i = cadaLoopVar_k(:,cadaForCount_k);
    i = VarAnalyzer('cadaLoopVar_k(:,cadaForCount_k)',i,'i',0);
```

```
    [ForEvalStr, ForEvalVar] = ForIterEnd(k,adigatorForVar_k_i);
end
```

```
if not(isempty(ForEvalStr))
    cellfun(@eval,ForEvalStr)
end
```

(a) Original Loop Fragment

(b) Transformed Loop Fragment

Fig. 4: Transformation of User Source Loop Fragment L_k to Intermediate Source Loop Fragment L'_k. The quantity L_k refers to the k^{th} loop fragment encountered in the user program, I_k is the fragment of code contained within the loop, and the quantity `sf` denotes an arbitrary loop index expression.

and/or loop fragments. In order to account for nested flow control, the user source to intermediate source transformation is performed in a recursive process. Consequently, if the fragments $B_{k,i}$ and/or I_k contain loop/conditional fragments, then the transformed fragments $B'_{k,i}$ and/or I'_k are made to contain transformed loop/conditional fragments.

5.2. Parsing of the Intermediate Program

After generating the intermediate program, the next step is to build a record of all objects encountered in the intermediate program as well as the locations of any flow control statements. Building this record results in an *object flow structure* (OFS) and *control flow structure* (CFS). In this paper, the OFS and CFS are analogous to the data flow graphs and control flow graphs of conventional compilers. Unlike conventional data flow graphs and control flow graphs, however, the OFS and the CFS are based on the locations and dependencies of all overloaded objects encountered in the intermediate program, where each overloaded object is assigned a unique integer value. The labeling of overloaded objects is achieved by using the global integer variable

`OBJECTCOUNT`, where `OBJECTCOUNT` is set to unity prior to the evaluation of the intermediate program. Then, as the intermediate program is evaluated on overloaded objects, each time an object is created the `id` field of the created object is assigned the value of `OBJECTCOUNT`, and `OBJECTCOUNT` is incremented. As control flow has been removed from the intermediate program, there exists only a single path which it may take and thus if an object takes on the `id` field equal to i on one evaluation of the intermediate program, then that object will take on the `id` field equal to i on *any* evaluation of the intermediate program. Furthermore, if a loop is to be evaluated for multiple iterations, then the value of `OBJECTCOUNT` prior to the first iteration is saved, and `OBJECTCOUNT` is set to the saved value prior to the evaluation of *any* iteration of the loop. By resetting `OBJECTCOUNT` at the start of each loop iteration, we ensure that the `id` assigned to all objects within a loop are iteration independent.

Because it is required to know the OFS and CFS *prior* to performing any derivative operations, the OFS and CFS are built by evaluating the intermediate program on a set of empty overloaded objects. During this evaluation, no function or derivative properties are built. Instead, only the `id` field of each object is assigned and the OFS and CFS are built. Using the unique `id` assigned to each object, the OFS is built as follows:

- Whenever an operation is performed on an object $\mathcal{O}$ to create a new object, $\mathcal{P}$, we record that the $\mathcal{O}.\texttt{id}^{th}$ object was last used to create the $\mathcal{P}.\texttt{id}^{th}$ object. Thus, after the entire program has been evaluated, the location (relative to all other created objects) of the last operation performed on $\mathcal{O}$ is known.
- If an object $\mathcal{O}$ is written to a variable, `v`, it is recorded that the $\mathcal{O}.\texttt{id}^{th}$ object was written to a variable with the name `'v'`. Furthermore, it is noted if the object $\mathcal{O}$ was the result of an array subscript assignment or not.

Similarly, the CFS is built based off of the `OBJECTCOUNT` in the following manner:

- Before and immediately after the evaluation of each branch of each conditional fragment, the value of `OBJECTCOUNT` is recorded. Using these two values, it can be determined which objects are created within each branch (and subsequently which variables are written, given the OFS).
- Before and immediately after the evaluation of a single iteration of each `for` loop, the value of `OBJECTCOUNT` is recorded. Using the two recorded values it can then be determined which objects are created within the loop.

The OFS and CFS then contain *most* of the information contained in conventional data flow graphs and control flow graphs, but are more suitable to the recursive manner in which flow control is handled in this paper.

5.3. Creating an Object Overmapping Scheme

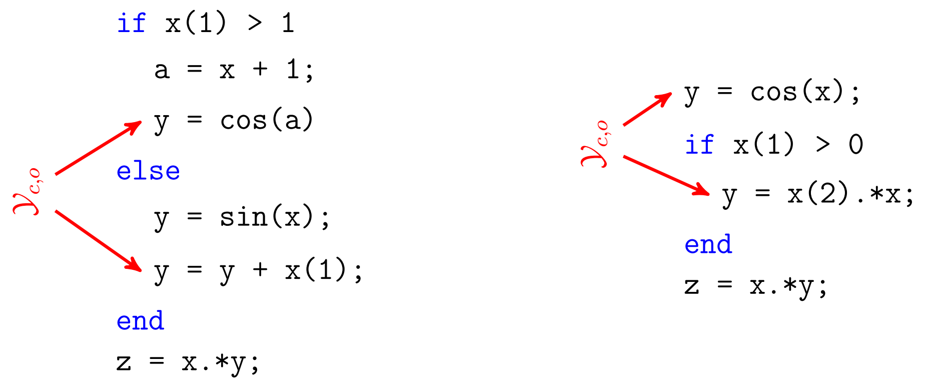
Using the OFS and CFS obtained from the empty evaluation, it is then required that an *object mapping scheme* (OMS) be built, where the OMS will be used in both the overmapping and printing evaluations. The OMS is used to tell the various transformation routines when an overloaded union must be performed to build an overmap, where the overmapped object is being stored, and when the overmapped object must be written to a variable in the overloaded workspace. The determination of where unions must be performed is based off of where, in the original user code, the program joins. That is, unions must be performed at the exit of conditional fragments and at the entrance of loops. Similarly, the OMS must also tell the transformation routines when an object must be saved, to where it will be saved, and when the saved object must be written to a variable in the overloaded workspace. We now look at how the OMS must be built in order to deal with both conditional branches and loop statements.

5.3.1. Conditional Statement Mapping Scheme. It is required to print conditional fragments to the derivative program such that different branches may be taken depending upon numerical input values. The difficulty that arises using the **cada** class is that, given a conditional fragment, different branches can write different assignment objects to the same output variable, and each of these assignment objects can contain differing function and/or derivative properties. To ensure that any operations performed on these variables *after* the conditional block are valid for any of the conditional branches, a conditional overmap must be assigned to all variables defined within a conditional fragment and used later in the program (that is, the outputs of the conditional fragment). Given a conditional fragment C'_k containing N_k branches, where each branch contains the fragment $B'_{k,i}$ (as shown in Fig. 3), the following rules are defined:

Mapping Rule 1. If a variable y is written within C'_k and read within $Succ\,(C'_k)$, then a single conditional overmap, $\mathcal{Y}_{c,o}$, must be created and assigned to the variable y prior to the execution of $Succ\,(C'_k)$.

Mapping Rule 1.1. For each branch fragment $B'_{k,i}$ ($i = 1, \ldots, N_k$) within which y is written, the last object assigned to y within $B'_{k,i}$ must belong to $\mathcal{Y}_{c,o}$. See Fig. 5a for illustrative example.

Mapping Rule 1.2. If there exists a branch fragment $B'_{k,i}$ within which y is *not* written, *or* there exists no `else` branch, then the last object which is written to y within $Pred\,(C'_k)$ must belong to $\mathcal{Y}_{c,o}$. See Fig. 5b for illustrative example.



```
if x(1) > 1
   a = x + 1;
   y = cos(a)
else
   y = sin(x);
   y = y + x(1);
end
z = x.*y;
```

(a) Example of Mapping Rule 1.1. The marked variables belong to the conditional overmap $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.1. Since the unmarked variables are not outputs of the conditional fragment, they do not belong to a conditional overmap.

```
y = cos(x);
if x(1) > 0
   y = x(2).*x;
end
z = x.*y;
```

(b) Example of Mapping Rule 1.2. The marked variables belong to the conditional overmap $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.2.

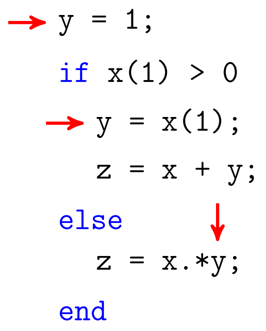
Fig. 5: Illustrative Examples of Mapping Rule 1.

Using Mapping Rule 1, it is determined for each conditional fragment which variables require a conditional overmap and which assignment objects belong to which conditional overmaps. By adhering to Mapping Rule 1, it is the case that each conditional fragment is analyzed independently of the others. For instance, if a program contains two successive conditional fragments C_1 and C_2, $C_2 \not\subset C_1$, containing N_1 and N_2 branches, respectively, then the program contains $N_1 N_2$ possible branches (assuming both C_1 and C_2 contain `else` branches). Rather than analyzing all $N_1 N_2$ possible branches, the proposed mapping rule states to first analyze the N_1 branches of C_1 and to then use the overmapped outputs to analyze the N_2 branches of C_2. Similarly, if a program contains a nested conditional fragment $C_2 \subset C_1$, then the inner fragment C_2 is first analyzed and the overmapped outputs are used for the analysis of the outer fragment C_1.

The second issue with evaluating transformed conditional fragments of the form shown in Fig. 3b stems from the fact that flow control is removed in the intermediate program. Thus, each branch is analyzed via overloaded evaluation in a successive order. To properly emulate the flow control and force each transformed conditional branch to be analyzed independently, the following mapping rule is defined for each transformed conditional fragment C'_k:

Mapping Rule 2. If a variable `y` is written within $Pred\,(C'_k)$, rewritten within a branch fragment $B'_{k,i}$ and read within a branch fragment $B'_{k,j}$ $(i < j)$, then the last object written to `y` within $Pred\,(C'_k)$ must be saved. Furthermore, the saved object must be assigned to the variable `y` prior to the evaluation of the fragment $B'_{k,j}$. See Fig. 6 for illustrative example.

In order to adhere to Mapping Rule 2, we use the OFS and CFS to determine the `id` of all objects which must be saved, where they will be saved to, and the `id` of the assignment objects who must be replaced with the saved objects.



```
y = 1;
if x(1) > 0
   y = x(1);
   z = x + y;
else
   z = x.*y;
end
```

Fig. 6: Illustrative Example of Mapping Rule 2. The variable `y` is an input to the `else` branch, but rewritten in the `if` branch. Since both branches are analyzed successively, the input version of `y` (`y = 1`) must be saved prior to the overloaded evaluation of the `if` branch and brought back into the overloaded workspace prior to the overloaded evaluation of the `else` branch.

5.3.2. Loop Mapping Scheme. In order to print derivative calculations within loops in the derivative program, a single loop iteration must be evaluated on a set of overloaded inputs to the loop, where the input function sizes and/or derivative sparsity patterns may change with each loop iteration. In order to print calculations which are valid for all possible loop inputs, a set of loop overmapped inputs are found by analyzing all possible loop iterations (that is, unrolling for the purpose of analysis) in the overmapping evaluation phase. The loop may then be printed as a rolled loop to the derivative program by evaluating on the set of overmapped loop inputs. The first mapping rule is now given for a transformed loop L'_k of the form shown in Fig. 4.

Mapping Rule 3. If a variable is written within I'_k, then it must belong to a *loop overmap*. Moreover, during the printing evaluation of I'_k, the overloaded workspace must only contain loop overmaps.

Mapping Rule 3.1. If a variable y is written within $Pred\,(L'_k)$, read within I'_k, and then rewritten within I'_k (that is, y is an iteration dependent input to I'_k), then the last objects written to y within $Pred\,(L'_k)$ and I'_k must share the same loop overmap. Furthermore, during the printing evaluation, the loop overmap must be written to the variable y prior to the evaluation of I'_k. See Fig. 7a for illustrative example.

Mapping Rule 3.2. Any assignment object which results from an array subscript assignment belongs to the same loop overmap as the last object which was written to the same variable. See Fig. 7b for illustrative example.

Mapping Rule 3.3. If an assignment object is created within multiple nested loops, then it still only belongs to one loop overmap. See Fig. 7c for illustrative example.

Mapping Rule 3.4. Any object belonging to a conditional overmap within a loop must share the same loop overmap as all objects which belong to the conditional overmap. See Fig. 7d for illustrative example.

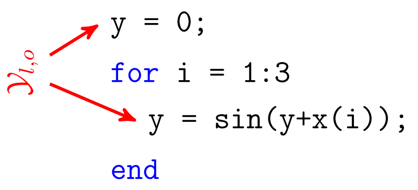
Using this set of rules together with the developed OFS and CFS, it is determined, for each loop L'_k, which assignment objects belong to which loop overmaps. Unlike the handling of conditional fragments, outer loops are made to dominate the overloaded analysis as a result of Mapping Rule 3. While this rule may result in the collection of unnecessary data, it simplifies the analysis of any flow control nested within the loop. For instance, in the case of a conditional fragment nested within a loop (such as that shown in Fig. 7d), the loop overmap is always made to contain the iteration dependent conditional overmap. In the overmapping evaluation of such a code fragment, the iteration dependent conditional overmaps would be built on each iteration in order to properly propagate sparsity patterns. In the printing evaluation, however, conditional overmaps are unnecessary as it is ensured that all possible conditional overmaps belong to the loop overmap. Additionally, this provides a measure of safety for the printing evaluation by ensuring that all assignment objects created within loops are in the overmapped form.

The second mapping rule associated with loops results from the fact that the overmapped outputs of a loop are not necessarily the same as the *true* outputs of a loop. Loop overmaps are eliminated from the overloaded workspace by replacing them with assignment objects that result from the overloaded evaluation of all iterations of the loop. This final mapping rule is stated as follows:

Mapping Rule 4:. For any outer loop L'_k (that is, there does not exist L'_j such that $L'_k \subset L'_j$), if a variable y is written within I'_k, and read within $Succ\,(L'_k)$, then the last object written to y within the loop during the overmapping evaluation must be saved. Furthermore, the saved object must be written to y prior to the evaluation of $Succ\,(L'_k)$ in both the overmapping and printing evaluations. See Fig. 8 for illustrative example.

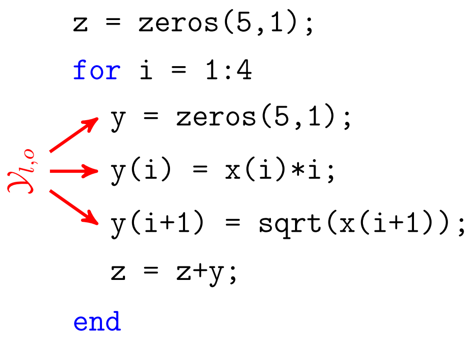
The `id` field of any assignment objects subject to Mapping Rule 4 are marked so that the assignment objects may be saved at the time of their creation and accessed at a later time.

It is noted that the presented mapping rules do not address cases of loops containing `break` or `continue` statements. We now briefly introduce how the proposed method handles such cases. For the sake of conciseness, however, they will neither be discussed in detail nor addressed in the remaining sections. In the presence of `break/continue`



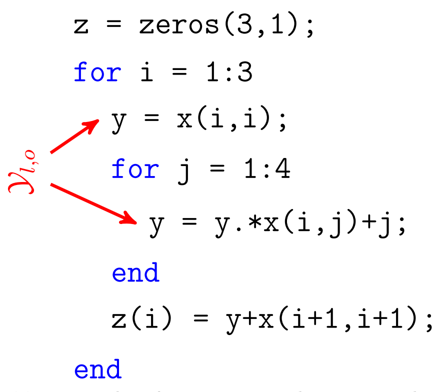
```
y = 0;
for i = 1:3
    y = sin(y+x(i));
end
```

(a) Example of Mapping Rule 3.1. The variable `y` is an iteration dependent input to the loop, thus the objects assigned to the marked variables must be joined to create the corresponding overmapped input. During the overmapping evaluation phase, this is achieved by joining the four different objects assigned to `y`: the original input, together with the result of `sin(y + x(i))` for the three values of `i`.



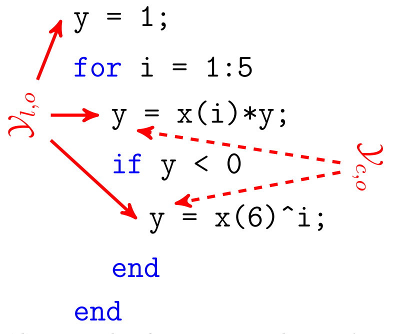
```
z = zeros(5,1);
for i = 1:4
    y = zeros(5,1);
    y(i) = x(i)*i;
    y(i+1) = sqrt(x(i+1));
    z = z+y;
end
```

(b) Example of Mapping Rule 3.2. Since the variable `y` is initialized to zero then assigned to via subscript index assignment twice, the objects which result from all three assignments are made to belong to the loop overmap $\mathcal{Y}_{l,o}$. Thus, the loop overmap $\mathcal{Y}_{l,o}$ is built by joining 12 overloaded objects, three for each loop iteration.



```
z = zeros(3,1);
for i = 1:3
    y = x(i,i);
    for j = 1:4
        y = y.*x(i,j)+j;
    end
    z(i) = y+x(i+1,i+1);
end
```

(c) Example of Mapping Rule 3.3. In this example, the variable `y` is written within a nested loop and is also an iteration dependent input to the nested loop. As a result of Mapping Rules 3.1 and 3.3, the corresponding loop overmap is made to contain the 15 different objects written to `y`: three from the outer loop assignment and 12 from the nested loop assignment.

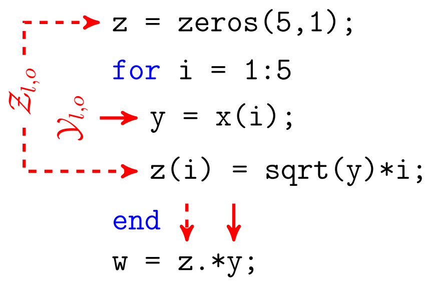


```
y = 1;
for i = 1:5
    y = x(i)*y;
    if y < 0
        y = x(6)^i;
    end
end
```

(d) Example of Mapping Rule 3.4. As a result of Mapping Rule 1.2, the conditional overmap $\mathcal{Y}_{c,o}$ is made to contain the objects assigned to `y` at both places within the loop, where a new conditional overmap is built on each iteration of the loop in the overmapping evaluation. As a result of Mapping Rule 3.4, the loop overmap $\mathcal{Y}_{l,o}$ is made to contain the object assigned to `y` prior to the loop, as well as the objects assigned to `y` across all overloaded evaluations of the loop.

Fig. 7: Illustrative Examples of Mapping Rule 3.

statements, the *true* outputs of the loop (as addressed in Mapping Rule 4) are considered to be the union between all possible outputs of the loop (that is, the outputs which result from any `break` statement firing across all iterations, the outputs which result from any `continue` statement firing on the final iteration, and the outputs which result from no `break` statements firing on any iteration and no `continue` statements firing on



```
z = zeros(5,1);
for i = 1:5
  y = x(i);
  z(i) = sqrt(y)*i;
end
w = z.*y;
```

Fig. 8: Illustrative Example of Mapping Rule 4. In this example, both the variables y and z are outputs of the loop. Thus, the objects assigned to the variables y and z on the fifth and final iteration of the loop must be stored in the overmapping evaluation phase and returned as outputs in both the overmapping evaluation and printing evaluation of the loop. Here it can be seen that the derivative sparsity pattern of the *true* overloaded output corresponding to y will contain less non-zeros than the loop overmap $\mathcal{Y}_{l,o}$, while the true overloaded output corresponding to z is equal to the loop overmap $\mathcal{Z}_{l,o}$.

the final iteration). In the presence of `continue` statements, the overmapped loop inputs (as addressed in Mapping Rule 3.1) must be made to contain all possible inputs to the loop (that is, any initial inputs together with any iteration dependent inputs which result from a `continue` statement firing, or no `continue` statements firing).

5.4. Overmapping Evaluation

The purpose of the overmapping evaluation is to build the aforementioned conditional overmaps and loop overmaps, as well as to collect data regarding organizational operations within loop statements. This overmapping evaluation is performed by evaluating the intermediate program on overloaded **cada** objects, where no calculations are printed to file, but rather only data is collected. Recall now from Mapping Rules 1 and 3 that any object belonging to a conditional and/or loop overmap must be an assignment object. Additionally, all assignment objects are sent to the *VarAnalyzer* routine immediately after they are created. Thus, building the conditional and loop overmaps may be achieved by performing overloaded unions within the variable analyzer routine immediately after the assignment objects are created. The remaining transformation routines must then control the flow of the program and manipulate the overloaded workspace such that the proper overmaps are built. We now describe the tasks performed by the transformation routines during the overmapping evaluation of conditional and loop fragments.

5.4.1. Overmapping Evaluation of Conditional Fragments. During the overmapping evaluation of a conditional fragment, it is required to emulate the corresponding conditional statements of the original program. First, those branches on which overmapping evaluations are performed must be determined. This determination is made by analyzing the conditional variables given to the *IfInitialize* routine (`cadacond1`,...,`cadacondn-1` of Fig. 3), where each of these variables may take on a value of *true*, *false*, or *indeterminate*. Any value of *indeterminate* implies that the variable may take on the value of either *true* or *false* within the derivative program. Using this information, it can be determined if overmapping evaluations are *not* to be performed within any of the branches. For any such branches within which overmapping evaluations are not to be performed, empty evaluations are performed in a manner similar to those performed in the parsing evaluation. Next, it is required that all branches of the conditional fragment be evaluated independently (that is, we must adhere to Mapping Rule 2). Thus, for a conditional fragment C'_k, if a variable is written within $Pred\,(C'_k)$, rewritten within $B'_{k,i}$, and then read within $B'_{k,j}$ $(i < j)$, then the variable analyzer will use the developed OMS to save the last assignment object written to the variable within $Pred\,(C'_k)$. The saved object may then be written to the overloaded workspace prior to the evaluation of any dependent branches using the outputs of the *IfIterStart* routines corresponding to the dependent branches. Finally, in order to ensure that any overmaps built after the conditional fragment are valid for all branches of the conditional fragment, all conditional overmaps associated with the conditional fragment must be assigned to the overloaded workspace. For some conditional overmap, these assignments are achieved ideally within the *VarAnalyzer* at the time which the last assignment object belonging to the conditional overmap is assigned. If, however, such assignments are not possible (due to the variable being read prior to the end of the conditional fragment), then the conditional overmap may be assigned to the overloaded workspace via the outputs of the last *IfIterEnd* routine.

5.4.2. Overmapping Evaluation of Loops. As seen from Fig. 4, all loops in the intermediate program are preceded by a call to the *ForInitialize* routine. In the overmapping evaluation, this routine determines the size of the second dimension of the object to be looped upon, and returns `adigatorForVar_k` such that all loop iterations will be analyzed successively. During these loop iterations, the loop overmaps are built and organizational operation data is collected. Here we stress that at no time during the overmapping evaluation are any loop overmaps active in the overloaded workspace. Thus, after the loop has been evaluated for all iterations, the objects which result from the evaluation of the last iteration of the loop will be active in the overloaded workspace. Furthermore, on this last iteration, the *VarAnalyzer* routine saves any objects subject to Mapping Rule 4 for use in the printing evaluation.

5.5. Organizational Operations within `For` Loops

Consider that all organizational operations may be written as one or more references or assignments: horizontal or vertical concatenation can be written as multiple subscript index assignments, reshapes can be written as either a reference or a subscript index assignment, etc. Consider now that the derivative operation corresponding to a function reference/assignment is given by performing the same reference/assignment on the first dimension of the Jacobian. In the method of this paper, however, derivative variables are written as vectors of non-zeros. Thus, the derivative procedures corresponding to function references/assignments cannot be written in terms of *function* reference/assignment indices. Moreover, when dealing with loops, it is often the case that function reference/assignment indices change on loop iterations, and thus the corresponding derivative procedures must be made to be iteration dependent. In this section we describe how organizational operations are handled within loops.

Example of an Organizational Operation within a Loop. Consider the following example that illustrates the method used to deal with organizational operations within loops. Suppose that, within a loop, there exists an organizational operation

$$\mathbf{w}_i = \mathbf{f}(\mathbf{v}_i, \mathbf{j}_i^{\mathbf{v}}, \mathbf{j}_i^{\mathbf{w}}), \tag{8}$$

where, on iteration $i \in [1, \ldots, N]$, the elements $\mathbf{j}_i^{\mathbf{v}}$ of $\mathbf{v}_i$ are assigned to the elements $\mathbf{j}_i^{\mathbf{w}}$ of $\mathbf{w}_i$. Let the overmapped versions of $\mathcal{V}_i$ and $\mathcal{W}_i$, across all calls to the overloaded organizational operation, $\mathcal{F}$, be denoted by $\bar{\mathcal{V}}$ and $\bar{\mathcal{W}}$ with associated overmapped Jacobians, $\mathbf{J}\bar{\mathbf{v}}(\mathbf{x})$ and $\mathbf{J}\bar{\mathbf{w}}(\mathbf{x})$, respectively. Further, let the non-zeros of the overmapped Jacobians be defined by the vectors $\mathbf{d}_{\mathbf{x}}^{\bar{\mathbf{v}}} \in \mathbb{R}^{nz_{\bar{\mathbf{v}}}}$ and $\mathbf{d}_{\mathbf{x}}^{\bar{\mathbf{w}}} \in \mathbb{R}^{nz_{\bar{\mathbf{w}}}}$.

Now, given the overmapped sparsity patterns of $\bar{\mathcal{V}}$ and $\bar{\mathcal{W}}$, together with the reference and assignment indices, $\mathbf{j}_i^{\mathbf{v}}$ and $\mathbf{j}_i^{\mathbf{w}}$, the derivative reference and assignment indices $\mathbf{k}_i^{\bar{\mathbf{v}}}$, $\mathbf{k}_i^{\bar{\mathbf{w}}} \in \mathbb{Z}^{m_i}$ are found such that, on iteration i, the elements $\mathbf{k}_i^{\bar{\mathbf{v}}}$ of $\mathbf{d}_{\mathbf{x}}^{\bar{\mathbf{v}}}$ are assigned to the elements $\mathbf{k}_i^{\bar{\mathbf{w}}}$ of $\mathbf{d}_{\mathbf{x}}^{\bar{\mathbf{w}}}$. That is, on iteration i, the valid derivative rule for the operation $\mathcal{F}$ is given as

$$d^{\bar{\mathbf{w}}}_{\mathbf{x}[k_i^{\bar{\mathbf{w}}}(l)]} = d^{\bar{\mathbf{v}}}_{\mathbf{x}[k_i^{\bar{\mathbf{v}}}(l)]} \quad l = 1, \dots, m_i \tag{9}$$

In order to write these calculations to file as concisely and efficiently as possible, a sparse *index matrix*, $\mathbf{K} \in \mathbb{Z}^{nz_{\bar{\mathbf{w}}} \times N}$ is defined such that, in the i^{th} column of $\mathbf{K}$, the elements of $\mathbf{k}_i^{\bar{\mathbf{v}}}$ lie in the row locations defined by the elements of $\mathbf{k}_i^{\bar{\mathbf{w}}}$. The following derivative rule may then be written to a file:

```
w.dx(logical(K(:,i))) = v.dx(nonzeros(K(:,i)));
```
(10)

Where `K` corresponds to the *index matrix* $\mathbf{K}$, `i` is the loop iteration, and `w.dx,v.dx` are the derivative variables associated with $\bar{\mathcal{W}}, \bar{\mathcal{V}}$.

5.5.1. Collecting and Analyzing Organizational Operation Data. As seen in the example above, derivative rules for organizational operations within loops rely on index matrices to print valid derivative variable references and assignments. Here it is emphasized that, given the proper index matrix, valid derivative calculations are printed to file in the manner shown in Eq. 10 for *any* organizational operation contained within a loop. The aforementioned example also demonstrates that the index matrices may be built given the overmapped inputs and outputs (for example, $\bar{\mathcal{W}}$ and $\bar{\mathcal{V}}$), together with the iteration dependent function reference and assignment indices (for example, $\mathbf{j}_i^{\mathbf{v}}$ and $\mathbf{j}_i^{\mathbf{w}}$). While there exists no single operation in MATLAB which is of the form of Eq. (8), it is noted that the function reference and assignment indices may be easily determined for *any* organizational operation by collecting certain data at each call to the operation within a loop. Namely, for any single call to an organizational operation, any input reference/assignment indices and function sizes of all inputs and outputs are collected for each iteration of a loop within which the operation is contained. Given this information, the function reference and assignment indices are then determined for each iteration to rewrite the operation to one of the form presented in Eq. (8). In order to collect this data, each overloaded organizational operation must have a special routine written to store the required data when it is called from within a loop during the overmapping evaluations. Additionally, in the case of multiple nested loops, the data from each child loop must be neatly collected on each iteration of the parent loop's *ForIterEnd* routine. Furthermore, because it is required to obtain the overmapped versions of all inputs and outputs and it is sometimes the case that an input and/or output does not belong to a loop overmap, it is also sometimes the case that unions must be performed within the organizational operations themselves in order to build the required overmaps.

After having collected all of this information in the overmapping evaluation, it is then required that it be analyzed and derivative references and/or assignments be created prior to the printing of the derivative file. This analysis is done by first eliminating any redundant indices/sizes by determining which loops the indices/sizes are dependent upon. Index matrices are then built according to the rules of differentiation of the particular operation. These index matrices are then stored to memory and the proper string references/assignments are stored to be accessed by the overloaded operations. Additionally, for operations embedded within multiple loops it is often the case that the index matrix is built to span multiple loops and that, prior to an inner loop, a reference and/or reshape must be printed in order for the inner loop to have an index matrix of the form given in the presented example. For example, consider an object $\mathcal{W}$ which results from an organizational operation embedded within two loops, where the overmapped object $\bar{\mathcal{W}}$ contains n possible non-zero derivatives. If the outer loop runs for $i_1 = 1, \ldots, m_1$ iterations, the inner loop runs for $i_2 = 1, \ldots, m_2$ iterations, and the function reference/assignment indices are dependent upon both loops, then an index matrix $\mathbf{K}_1 \in \mathbb{Z}^{nm_2 \times m_1}$ will be built. Then, prior to the printing of the inner loop, a statement such as

```
K2 = reshape(K1(:,i1),n,m2);                    (11)
```

must be printed, where `K1` is the outer index matrix, `i1` is the outer loop iteration, and `n` and `m2` are the dimensions of the inner index matrix `K2`. The strings which must be printed to allow for such references/reshapes are then also stored into memory to be accessed prior to the printing of the nested `for` loop statements.

5.6. Printing Evaluation

During the printing evaluation we finally print the derivative code to a file by evaluating the intermediate program on instances of the **cada** class, where each overloaded operation prints calculations to the file in the manner presented in Section 3. In order to print `if` statements and `for` loops to the file that contains the derivative program, the transformation routines must both print flow control statements and ensure the proper overloaded objects exist in the overloaded workspace. Manipulating the overloaded workspace in this manner enables all overloaded operations to print calculations that are valid for the given flow control statements. Unlike in the overmapping evaluations, if the overloaded workspace is changed, then the proper re-mapping calculations must be printed to the derivative file such that all function variables and derivative variables within the printed file reflect the properties of the objects within the active overloaded workspace. We now introduce the concept of an overloaded remap and explore the various tasks performed by the transformation routines to print derivative programs containing both conditional and loop fragments.

5.6.1. Re-Mapping of Overloaded Objects. During the printing of the derivative program, it is often the case that an overloaded object must be written to a variable in the intermediate program by means of the transformation routines, where the variable was previously written by the overloaded evaluation of a statement copied from the user program. Because the derivative program is simultaneously being printed as the intermediate program is being evaluated, any time a variable is rewritten by means of a transformation routine, the printed function and derivative variable must be made to reflect the properties of the new object being written to the overloaded workspace. If an object $\mathcal{O}$ is currently assigned to a variable in the intermediate program, and a transformation routine is to assign a different object, $\mathcal{P}$, to the same variable, then the overloaded operation which prints calculations to transform the function variable and derivative variable(s) which reflect the properties of $\mathcal{O}$ to those which reflect the properties of $\mathcal{P}$ is referred to as the re-map of $\mathcal{O}$ to $\mathcal{P}$. To describe this process, consider the case where both $\mathcal{O}$ and $\mathcal{P}$ contain derivative information with respect to an input object $\mathcal{X}$. Furthermore, let `o.f` and `o.dx` be the function variable and derivative variable which reflect the properties of $\mathcal{O}$. Here it can be assumed that, because $\mathcal{O}$ is active in the overloaded workspace, the proper calculations have been printed to the derivative file to compute `o.f` and `o.dx`. If it is desired to *re-map* $\mathcal{O}$ to $\mathcal{P}$ such that the variables `o.f` and `o.dx` are used to create the function variable, `p.f`, and derivative variable, `p.dx`, which reflect the properties of $\mathcal{P}$, the following steps are executed:

- Build the function variable, `p.f`:
  Re-Mapping Case 1.A:. If `p.f` is to have a *greater* row and/or column dimension than those of `o.f`, then `p.f` is created by appending zeros to the rows and/or columns of `o.f`.

Re-Mapping Case 1.B:. If `p.f` is to have a *lesser* row and/or column dimension than those of `o.f`, then `p.f` is created by removing rows and/or columns from `o.f`.
Re-Mapping Case 1.C:. If `p.f` is to have the same dimensions as those of `o.f`, then `p.f` is set equal to `o.f`.

- Build the derivative variable, `p.dx`:
  Re-Mapping Case 2.A:. If $\mathcal{P}$ has more possible non-zero derivatives than $\mathcal{O}$, then `p.dx` is first set equal to a zero vector and then `o.dx` is assigned to elements of `p.dx`, where the assignment index is determined by the mapping of the derivative locations defined by $\mathcal{O}$`.deriv.nzlocs` into those defined by $\mathcal{P}$`.deriv.nzlocs`.
  Re-Mapping Case 2.B:. If $\mathcal{P}$ has less possible non-zero derivatives than $\mathcal{O}$, then `p.dx` is created by referencing off elements of `o.dx`, where the reference index is determined by the mapping of the derivative locations defined by $\mathcal{P}$`.deriv.nzlocs` into those defined by $\mathcal{O}$`.deriv.nzlocs`.
  Re-Mapping Case 2.C:. If $\mathcal{P}$ has the same number of possible non-zero derivatives as $\mathcal{O}$, then `p.dx` is set equal to `o.dx`.

It is noted that, in the 1.A and 2.A cases, the object $\mathcal{O}$ is being re-mapped to the overmapped object $\mathcal{P}$, where $\mathcal{O}$ belongs to $\mathcal{P}$. In the 1.B and 2.B cases, the overmapped object, $\mathcal{O}$, is being re-mapped to an object $\mathcal{P}$, where $\mathcal{P}$ belongs to $\mathcal{O}$. Furthermore, the re-mapping operation is only used to either map an object to an overmapped object which it belongs, or to map an overmapped object to an object which belongs to the overmap.

5.6.2. Printing Evaluation of Conditional Fragments. Printing a valid conditional fragment to the derivative program requires that the following tasks must be performed. First, it is necessary to print the conditional statements, that is, print the statements `if`, `elseif`, `else`, and `end`. Second, it is required that each conditional branch is evaluated independently (as was the case with the overmapping evaluation). Third, after the conditional fragment has been evaluated in the intermediate program (and printed to the derivative file), all associated conditional overmaps must be assigned to the proper variables in the intermediate program. In addition, all of the proper re-mapping calculations must be printed to the derivative file such that when each branch of the derivative conditional fragment is evaluated it will calculate derivative variables and function variables that reflect the properties of the active conditional overmaps in the intermediate program. These three aforementioned tasks are now explained in further detail.

The printing of the conditional branch headings is fairly straightforward due to the manner in which all expressions following the `if`/`elseif` statements in the user program are written to a conditional variable in the intermediate program (as seen in Fig. 3). Thus, each *`IfIterStart`* routine may simply print the branch heading as: `if cadacond1, elseif cadacond2, else` and so on. As each branch of the conditional fragment is then evaluated the overloaded **cada** operations will print derivative and function calculations to the derivative file in the manner described in Section 3. As was the case with the overmapping evaluations, each of the branch fragments must be evaluated independently, where this evaluation is performed in the same manner as described in Section 5.4.1. Likewise, it is ensured that the overloaded workspace will contain all associated conditional overmaps in the same manner as in the overmapping evaluation. Unlike the overmapping evaluation, however, conditional overmaps are not built during the printing evaluations. Instead, the conditional overmaps are stored within memory and may be accessed by the transformation routines in order to print the proper re-mapping calculations to the derivative file. For some conditional fragment C_k' within which the variable `y` has been written, the following calculations are required to print the function variable and derivative variable that reflect the properties of the conditional overmap $\mathcal{Y}_{c,o}$[3]

- If an object $\mathcal{Y}$ belongs to $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.1 and *is not* to be operated on from within the branch it is created, then $\mathcal{Y}$ is re-mapped to $\mathcal{Y}_{c,o}$ from within the *VarAnalyzer* routine at the time of which $\mathcal{Y}$ is assigned to `y`.
- If an object $\mathcal{Y}$ belongs to $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.1 and *is* to be operated on from within the branch it is created, then $\mathcal{Y}$ is stored from within the *VarAnalyzer* and the *IfIterEnd* routine corresponding to the branch performs the re-map of $\mathcal{Y}$ to $\mathcal{Y}_{c,o}$.
- For each branch within which `y` is not written, the *IfIterEnd* routine accesses both $\mathcal{Y}_{c,o}$ and the last object written to `y` within $Pred\,(C'_k)$ (where this will have been saved previously by the *VarAnalyzer* routine). The *IfIterEnd* routine then re-maps the previously saved object to the overmap.
- If the fragment C'_k contains no `else` branch, an `else` branch is imposed and the *last* *IfIterEnd* routine again accesses both $\mathcal{Y}_{c,o}$ and the last object written to `y` within $Pred\,(C'_k)$. This routine then re-maps the previously saved object to the overmap within the imposed `else` branch.

Finally, the last *IfIterEnd* routine prints the `end` statement.

5.6.3. Printing Evaluation of Loops. In order to print a valid loop fragment to the derivative program an overloaded evaluation of a single iteration of the transformed loop is performed in the intermediate program. To ensure that all printed calculations are valid for each iteration of the loop in the derivative program, these overloaded evaluations are performed only on loop overmaps. The printing evaluation of loops is then completed as follows. When an *outermost* loop is encountered in the printing evaluation of the intermediate program, the *ForInitialize* routine must first use the OMS to determine which, if any, objects belong to loop overmaps but are created prior to the loop, that is, loop inputs which are loop iteration dependent. Any of these previously created objects will have been saved within the *VarAnalyzer* routine, and the *ForInitialize* routine may then access both the previously saved objects and the loop overmaps to which they belong and then re-map the saved objects to their overmapped form. As it is required that the overmapped inputs to the loop be active in the overloaded workspace, the *ForInitialize* then uses its outputs, `ForEvalStr` and `ForEvalVar`, to assign the overmapped objects to the proper variables in the overloaded workspace. Next, the *ForIterStart* routine must print out the outermost `for` statement. Here we note that all outer loops must evaluate for a fixed number of iterations, so if the loop is to be run for 10 iterations, then the `for` statement would be printed as `for cadaforcount = 1:10`. The `cadaforcount` variable will then be used in the derivative program to reference columns off of the index matrices presented in Section 5.5 and to reference the user defined loop variable. As the loop is then evaluated on loop overmaps, all overloaded operations will print function and derivative calculations to file in the manner presented in Section 3, with the exception being the organizational operations. The organizational operations will instead use stored global data to print derivative references and assignments using the *index matrices* as described in Section 5.5. Moreover, during the evaluation of the loop, only organizational operations are cognizant of the fact that the evaluations are being performed within the loop. Thus, when operations are performed on loop overmaps, the resulting assignment objects can sometimes be

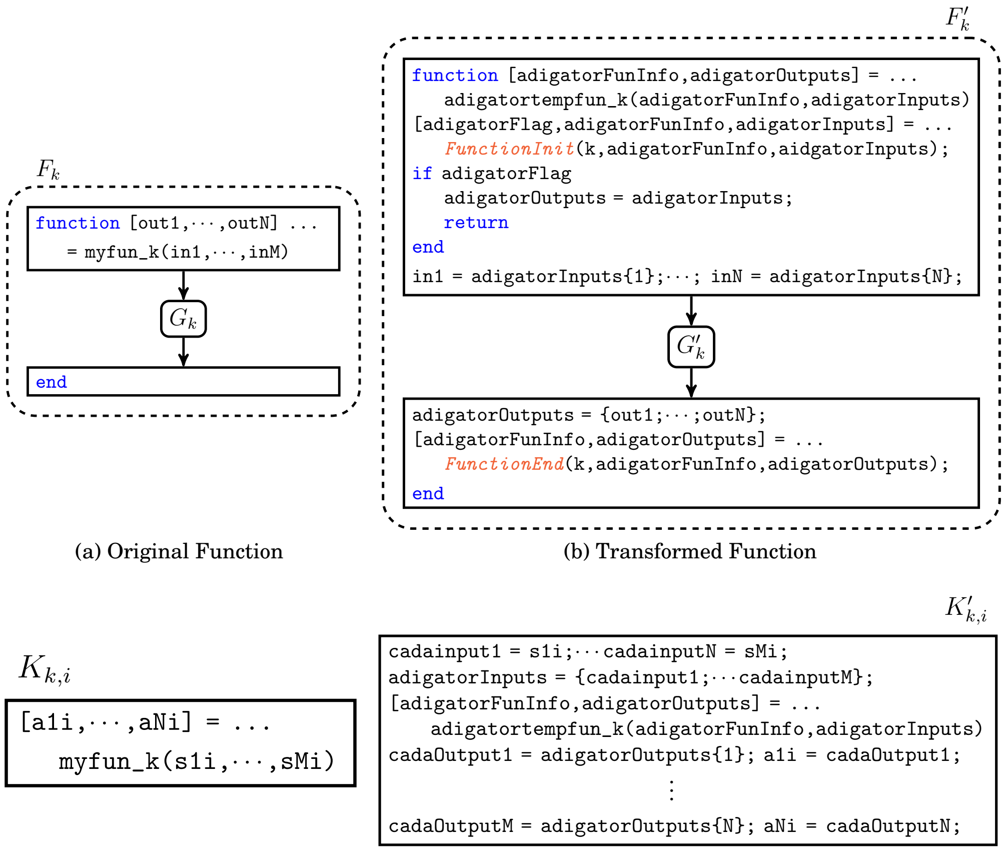
[3]This is assuming that the conditional fragment is not nested within a loop. In the case of a conditional fragment nested within a loop, Mapping Rule 3 takes precedence over Mapping Rule 1 in the printing evaluation.

computed to be different from their overmapped form. In order to adhere to Mapping Rule 3,after each variable assignment within the loop, the assignment object is sent to the *VarAnalyzer* routine which then re-maps the assigned object to the stored loop overmap to which it belongs.

Nested loops are handled different from non-nested loops. When a nested loop is reached during the printing evaluation, the active overloaded workspace will only contain loop overmaps, thus it is not required to re-map any nested loop inputs to their overmapped form. What is required though, is that the *ForInitialize* routine check to see if any index matrix references and/or reshapes must be printed, where if this is the case, the proper references and/or reshapes are stored during the organizational operation data analysis. These references/reshapes are then accessed and printed to file. The printing of the actual nested loop statements is then handled by the *ForIterStart* routine, where, unlike outer loops, nested loops may run for a number of iterations which is dependent upon a parent loop. If it is the case that a nested loop changes size depending upon a parent loop's iteration, then the loop statement is printed as: `for cadaforcount2 = 1:K(cadaforcount1)`, where `cadaforcount2` takes on the nested loop iteration, `cadaforcount1` is the parent loop iteration, and `K` is a vector containing the sizes of the nested loop. Any evaluations performed within the nested loop are then handled exactly as if they were within an outer loop, and when the *ForIterEnd* routine of the nested loop is encountered it prints the `end` statement.

After the outer loop has been evaluated, the outermost *ForIterEnd* routine then prints the `end` statement and must check to see if it needs to perform any re-maps as a result of Mapping Rule 4. In other words, the OMS is used to determine which variables defined within the loop will be read after the loop (that is, which variables are outputs of the loop). The objects that were written to these variables as a result of the last iteration of the loop in the overmapping evaluation are saved and are now accessible by the *ForIterEnd* routine. The loop overmaps are then re-mapped to the saved objects, and the outputs of *ForIterEnd*, `ForEvalStr` and `ForEvalVar`, are used to assign the saved objects to the overloaded workspace. Printing loops in this manner ensures that all calculations printed within the loop are valid for all iterations of the loop. Furthermore, this printing process ensures that any calculations printed after the loop are valid only for the result of the evaluation of all iterations of the loops (thus maintaining sparsity).

5.6.4. Storage of Global Data Required to Evaluate Derivative File. Because the method presented in this paper is geared towards applications in which the derivative is required at a number of different values, it is most efficient to determine all references and assignment indices required to evaluate the derivative file only once prior to all evaluations of the derivative file for a particular application. Thus, unlike the method of Patterson et al. [2013], where all reference and assignment indices are embedded within the derivative program itself, in the method of this paper these reference and assignment indices are written as variable names to a MATLAB binary file (which is an output of the process shown in Fig. 1). These variable names are then used within the derivative program itself and are recalled from global MATLAB memory. Furthermore, in the case where the associated global structure is cleared, it is restored on an ensuing call to the derivative program by loading the aforementioned MATLAB binary file associated with the derivative program. Using this aforementioned approach to keeping track of reference and assignment indices, the overhead associated with reading the indices is incurred only once. Furthermore, the derivative program is printed much more concisely than it would have been had the indices themselves been printed to the derivative file.



```
function [out1,···,outN] ...
   = myfun_k(in1,···,inM)
end
```

(a) Original Function

```
function [adigatorFunInfo,adigatorOutputs] = ...
   adigatortempfun_k(adigatorFunInfo,adigatorInputs)
[adigatorFlag,adigatorFunInfo,adigatorInputs] = ...
   FunctionInit(k,adigatorFunInfo,aidgatorInputs);
if adigatorFlag
   adigatorOutputs = adigatorInputs;
   return
end
in1 = adigatorInputs{1};···; inN = adigatorInputs{N};
```

```
adigatorOutputs = {out1;···;outN};
[adigatorFunInfo,adigatorOutputs] = ...
   FunctionEnd(k,adigatorFunInfo,adigatorOutputs);
end
```

(b) Transformed Function

```
[a1i,···,aNi] = ...
   myfun_k(s1i,···,sMi)
```

(c) Original Function Call

```
cadainput1 = s1i;···cadainputN = sMi;
adigatorInputs = {cadainput1;···cadainputM};
[adigatorFunInfo,adigatorOutputs] = ...
    adigatortempfun_k(adigatorFunInfo,adigatorInputs)
cadaOutput1 = adigatorOutputs{1}; a1i = cadaOutput1;
                          ⋮
cadaOutputM = adigatorOutputs{N}; aNi = cadaOutputN;
```

(d) Transformed Function Call

Fig. 9: Transformation of Original Function F_k to Transformed Function F'_k and Transformation of the i^{th} Call to Function F_k, $K_{k,i}$, to the i^{th} Call to Function F'_k, $K'_{k,i}$.

5.7. Multiple User Functions

At this point, the methods have been described that transform a program containing only a single function into a derivative program containing only a single function. It is often the case, however, that a user program consists of multiple functions and/or sub-functions. The method of this paper handles both the cases of called functions and sub-functions in the same manner. In order to deal with programs containing multiple function calls, the transformations of Fig. 9 are applied during the user source to intermediate source transformation phase. In the parsing phase, it is then determined which functions F'_k are called more than a single time. If a function is called only once, then the input and output objects are saved during the overmapping evaluation phase. Then, in the printing evaluation phase, when the function F'_k is called, the *FunctionInit* routine returns the saved outputs and sets the `adigatorFlag` true, returning the outputs to the calling function. After the calling function has been printed, the function F'_k is then evaluated in the manner discussed in Section 5.6.

In the case that a function F'_k is called multiple times, it is required to build a set of overmapped inputs and overmapped outputs of the function. The building of overmapped inputs and outputs is performed during the overmapping evaluation phase by joining the inputs across each function call, evaluating the called function on each unique set of inputs, and joining all possible outputs. Moreover, the outputs of *each* call are stored in memory for use in the printing evaluation phase. In order to allow for function-call-dependent organizational operations, each call is assigned an iteration count and organizational operations are then handled in the manner presented in Section 5.5. During the printing phase, when the function F_k' is called, the *FunctionInit* routine re-maps the given inputs for the particular call to the overmapped inputs, prints the function call, re-maps the overmapped outputs to the stored outputs, and then returns the stored outputs. The function is then *not* evaluated by setting the `adigatorFlag` to true. After the calling function has finished printing, the function F_k' is then evaluated on the set of stored overmapped inputs and all organizational operations are treated as if they were being called from within a loop.

6. EXAMPLES

The method described in Section 5 (which is implemented in the software *ADiGator*) now is applied to four examples. The first example provides a detailed examination of the process that *ADiGator* uses to handle conditional statements. The second example is the well known Speelpenning problem of Speelpenning [1980] and demonstrates the ability of *ADiGator* to handle a `for` loop while simultaneously providing a comparison between a rolled loop and an unrolled loop. The third example is a polynomial data fitting example which contains a `for` loop and provides a comparison between the efficiency of *ADiGator* with the previously developed operator overloaded tool *MAD* [Forth 2006] and the combination source transformation and operator overloading tool *ADiMat* [Bischof et al. 2003]. The fourth example is a large scale nonlinear programming problem (NLP) whose constraint function contains multiple levels of flow control. This fourth example is used to compare the efficiency of the first- and second-order derivative code generated by *ADiGator* against the previously developed software tools *INTLAB* [Rump 1999], *MAD* [Forth 2006], and *ADiMat* [Bischof et al. 2003]. Comparisons were performed against *INTLAB* version 6, *MAD* version 1.4, and *ADiMat* version 0.5.9. All computations were performed on an Apple Mac Pro with Mac OS-X 10.9.2 (Mavericks) and a 2×2.4 GHz Quad-Core Intel Xeon processor with 24 GB 1066 MHz DDR3 RAM using MATLAB version R2014a.

Example 1: Conditional Statement

In this example we investigate the transformation of a user program containing a conditional statement into a derivative program containing the same conditional statement. The original user program and the transformed intermediate program is shown in Figure 10, where it is seen that the variable `y` is written within both the `if` branch and the `else` branch. Furthermore, because the variable `y` is read after the conditional fragment, a conditional overmap, $\mathcal{Y}_{c,o}$, is required to print derivative calculations that are valid for both the `if` and `else` branches.

Now let $\mathcal{Y}_{\texttt{if}}$ and $\mathcal{Y}_{\texttt{else}}$ be the overloaded objects written to `y` as a result of the overloaded evaluation of the `if` and `else` branches, respectively, in the intermediate program. Using this notation, the required conditional overmap is $\mathcal{Y}_{c,o} = \mathcal{Y}_{\texttt{if}} \cup \mathcal{Y}_{\texttt{else}}$, where this conditional overmap is built during the overmapping evaluation of the intermediate program shown in Figure 10. Fixing the input to be a vector of length five, the sparsity patterns of the Jacobians defined by $\mathcal{Y}_{\texttt{if}}$, $\mathcal{Y}_{\texttt{else}}$, and $\mathcal{Y}_{c,o}$ are shown in Figure 11, and the final transformed derivative program is seen in Figure 12. From Figure 12, it is seen that re-mapping calculations are required from within both the `if` and `else` branches, where these re-mapping calculations represent the mapping of the non-zero elements of the Jacobians shown in Figures 11a and 11b into those shown

User Program

```
function z = myfun(x)

N  = 5;
x1 = x(1);
xN = x(N);

if x1 > xN
  y = x*x1;
else
  y = x*xN;
end

z = sin(y);
```

Intermediate Program

```
function [adigatorFunInfo, adigatorOutputs] = ...
    adigatortempfunc1(adigatorFunInfo,adigatorInputs)
[adigatorFlag, adigatorFunInfo, adigatorInputs] = ...
   FunctionInit(1,adigatorFunInfo,adigatorInputs);
if adigatorFlag; adigatorOutputs = adigatorInputs; return; end;
x = adigatorInputs{1};

N  = 5;
N  = VarAnalyzer('N  = 5;',N,'N',0);
x1 = x(1);
x1 = VarAnalyzer('x1 = x(1);',x1,'x1',0);
xN = x(N);
xN = VarAnalyzer('xN = x(N);',xN,'xN',0);

% ADiGator IF Statement #1: START
cadacond1 = x1 > xN;
cadacond1 = VarAnalyzer('cadacond1 = x1 > xN',cadacond1,'cadacond1',0);
IfInitialize(1,cadacond1,[]);
IfIterStart(1,1);
    y = x*x1;
    y = VarAnalyzer('y = x*x1;',y,'y',0);
IfIterEnd(1,1);
[IfEvalStr, IfEvalVar] = IfIterStart(1,2);
if not(isempty(IfEvalStr)); cellfun(@eval,IfEvalStr); end
    y = x*xN;
    y = VarAnalyzer('y = x*xN;',y,'y',0);
[IfEvalStr, IfEvalVar] = IfIterEnd(1,2);
if not(isempty(IfEvalStr)); cellfun(@eval,IfEvalStr); end
% ADiGator IF Statement #1: END

z = sin(y);
z = VarAnalyzer('z = sin(y);',z,'z',0);

adigatorOutputs = {z};
[adigatorFunInfo, adigatorOutputs] = ...
    FunctionEnd(1,adigatorFunInfo,adigatorOutputs);
```

Fig. 10: User Source-to-Intermediate Source Transformation for Example 1.

in Figure 11c. More precisely, in the `if` branch of the derivative program, the assignment indices `Gator1Indices.Index4` maps the derivative variable of $\mathcal{Y}_{\texttt{if}}$ into elements $[1, 2, 3, 4, 5, 6, 7, 8, 13]$ of the derivative variable of $\mathcal{Y}_{c,o}$. Similarly, in the `else` branch of the derivative program, the assignment index `Gator1Indices.Index8` maps the derivative variable of $\mathcal{Y}_{\texttt{else}}$ into the $[1, 6, 7, 8, 9, 10, 11, 12, 13]$ elements of the derivative variable of $\mathcal{Y}_{c,o}$. Thus, the evaluation of the derivative program shown in Figure 12 always produces a 13 element derivative variable, `z.dx`, which is mapped into a Jacobian of the form shown in Figure 11c using the mapping index written to `z.dx_location`. Due to the nature of the `if` and `else` branches of the derivative program, at least 4 elements of the derivative variable `z.dx` will always be identically zero, where the locations of the zero elements will depend upon which branch of the conditional block is taken.

Example 2: Speelpenning Problem

This example demonstrates the transformation of a function program containing a `for` loop into a derivative program containing the same `for` loop. The function program to

$$\begin{bmatrix} d_1 & 0 & 0 & 0 & 0 \\ d_2 & d_6 & 0 & 0 & 0 \\ d_3 & 0 & d_7 & 0 & 0 \\ d_4 & 0 & 0 & d_8 & 0 \\ d_5 & 0 & 0 & 0 & d_9 \end{bmatrix}$$

(a) $\mathbf{struct}(\mathcal{Y}_{\texttt{if}}.\texttt{deriv})$

$$\begin{bmatrix} d_1 & 0 & 0 & 0 & d_5 \\ 0 & d_2 & 0 & 0 & d_6 \\ 0 & 0 & d_3 & 0 & d_7 \\ 0 & 0 & 0 & d_4 & d_8 \\ 0 & 0 & 0 & 0 & d_9 \end{bmatrix}$$

(b) $\mathbf{struct}(\mathcal{Y}_{\texttt{else}}.\texttt{deriv})$

$$\begin{bmatrix} d_1 & 0 & 0 & 0 & d_9 \\ d_2 & d_6 & 0 & 0 & d_{10} \\ d_3 & 0 & d_7 & 0 & d_{11} \\ d_4 & 0 & 0 & d_8 & d_{12} \\ d_5 & 0 & 0 & 0 & d_{13} \end{bmatrix}$$

(c) $\mathbf{struct}(\mathcal{Y}_{c,o}.\texttt{deriv})$

Fig. 11: Derivative Sparsity Patterns of $\mathcal{Y}_{\texttt{if}}$, $\mathcal{Y}_{\texttt{else}}$, and $\mathcal{Y}_{c,o} = \mathcal{Y}_{\texttt{if}} \cup \mathcal{Y}_{\texttt{else}}$.

be transformed computes the Speelpenning function [Speelpenning 1980] given as

$$y = \prod_{i=1}^{N} x_i, \tag{12}$$

where Eq. (12) is implemented using a `for` loop as shown in Fig. 13. From Fig. 13, two major challenges of transforming the loop are identified. First, it is seen that the variable `y` is an input to the loop, rewritten within the loop, and read after the loop (as the output of the function), thus a loop overmap, $\mathcal{Y}_{c,o}$ must be built to print valid derivative calculations within the loop. Second, it is seen that the reference `x(I)` depends upon the loop iteration, where the object assigned to `x` has possible non-zero derivatives. Thus, an index matrix, $\mathbf{K}$, and an overmapped `subsref` output, $\mathcal{R}_o$, must be built to allow for the iteration dependent derivative reference corresponding to the function reference `x(I)`. To further investigate, let $\mathcal{Y}_i$ be the object written to `y` after the evaluation of the i^{th} iteration of the loop from within the intermediate program. Furthermore, we allow R_i to be the intermediate object created as a result of the overloaded evaluation of `x(I)` within the intermediate program. The overmapped objects $\mathcal{Y}_{l,o}$ and $\mathcal{R}_o$ are now defined as

$$\mathcal{Y}_{l,o} = \bigcup_{i=0}^{N} \mathcal{Y}_i \tag{13}$$

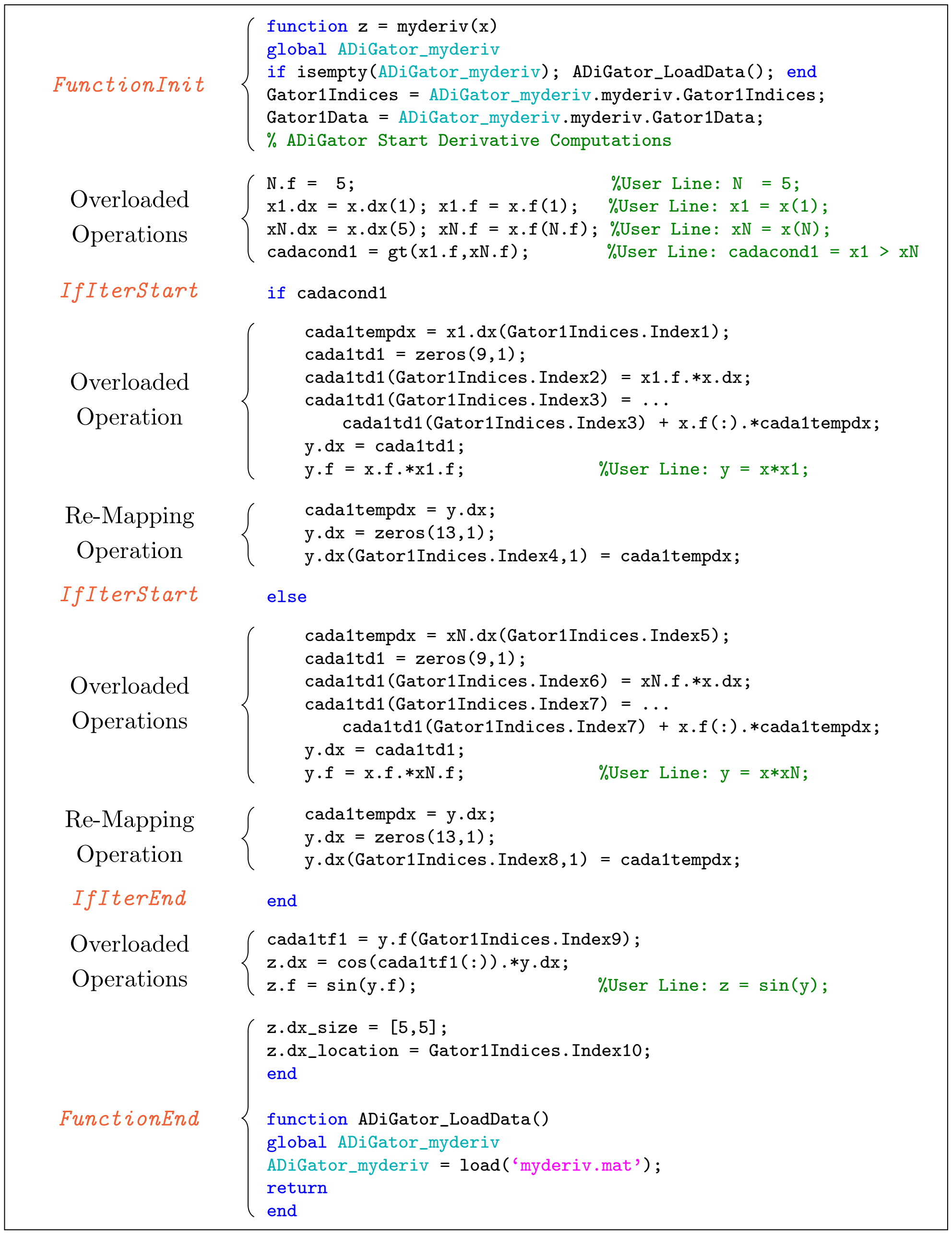
and

$$\mathcal{R}_o = \bigcup_{i=1}^{N} \mathcal{R}_i, \tag{14}$$

where $\mathcal{Y}_0$ represents the object written to `y` prior to the evaluation of the loop in the intermediate program. Now, as the input object, $\mathcal{X}$, has a Jacobian with a diagonal sparsity pattern (which does not change), then each object $\mathcal{R}_i$ will have a $1 \times N$ gradient where the i^{th} element is a possible non-zero. Thus, the union of all such gradients over i results in all N elements being possibly non-zero, that is, a full gradient. Within the derivative program, the reference operation must then result in a derivative variable of length N, where on iteration i, the i^{th} element of the derivative variable corresponding to $\mathcal{X}$ must be assigned to the i^{th} element of the derivative variable corresponding to $\mathcal{R}_o$. This reference and assignment is done as described in Section 5.5 using the index matrix $\mathbf{K} \in \mathbb{Z}^{N \times N}$, where

$$K_{i,j} = \begin{cases} i, & i = j \\ 0, & i \neq j \end{cases} \quad \begin{array}{l} i = 1, \ldots, N \\ j = 1, \ldots, N \end{array}. \tag{15}$$

Now, because $\mathcal{R}_i$ contains a possible non-zero derivative in the i^{th} location of the gradient and $\mathcal{Y}_i = \mathcal{Y}_{i-1} * \mathcal{R}_i$, $\mathcal{Y}_i$ will contain i possible non-zero derivatives in the first i locations. Furthermore, the union of $\mathcal{Y}_i$ over $i = 0, \ldots, N$ results in an object $\mathcal{Y}_{l,o}$ containing N possible non-zero derivatives (that is, a full gradient). Thus, in the derivative


FunctionInit
Overloaded Operations
IfIterStart
Overloaded Operation
Re-Mapping Operation
IfIterStart
Overloaded Operations
Re-Mapping Operation
IfIterEnd
Overloaded Operations
FunctionEnd

```
function z = myderiv(x)
global ADiGator_myderiv
if isempty(ADiGator_myderiv); ADiGator_LoadData(); end
Gator1Indices = ADiGator_myderiv.myderiv.Gator1Indices;
Gator1Data = ADiGator_myderiv.myderiv.Gator1Data;
% ADiGator Start Derivative Computations

N.f =  5;                              %User Line: N  = 5;
x1.dx = x.dx(1); x1.f = x.f(1);   %User Line: x1 = x(1);
xN.dx = x.dx(5); xN.f = x.f(N.f); %User Line: xN = x(N);
cadacond1 = gt(x1.f,xN.f);        %User Line: cadacond1 = x1 > xN

if cadacond1

    cada1tempdx = x1.dx(Gator1Indices.Index1);
    cada1td1 = zeros(9,1);
    cada1td1(Gator1Indices.Index2) = x1.f.*x.dx;
    cada1td1(Gator1Indices.Index3) = ...
        cada1td1(Gator1Indices.Index3) + x.f(:).*cada1tempdx;
    y.dx = cada1td1;
    y.f = x.f.*x1.f;              %User Line: y = x*x1;

    cada1tempdx = y.dx;
    y.dx = zeros(13,1);
    y.dx(Gator1Indices.Index4,1) = cada1tempdx;

else

    cada1tempdx = xN.dx(Gator1Indices.Index5);
    cada1td1 = zeros(9,1);
    cada1td1(Gator1Indices.Index6) = xN.f.*x.dx;
    cada1td1(Gator1Indices.Index7) = ...
        cada1td1(Gator1Indices.Index7) + x.f(:).*cada1tempdx;
    y.dx = cada1td1;
    y.f = x.f.*xN.f;              %User Line: y = x*xN;

    cada1tempdx = y.dx;
    y.dx = zeros(13,1);
    y.dx(Gator1Indices.Index8,1) = cada1tempdx;

end

cada1tf1 = y.f(Gator1Indices.Index9);
z.dx = cos(cada1tf1(:)).*y.dx;
z.f = sin(y.f);                   %User Line: z = sin(y);

z.dx_size = [5,5];
z.dx_location = Gator1Indices.Index10;
end

function ADiGator_LoadData()
global ADiGator_myderiv
ADiGator_myderiv = load('myderiv.mat');
return
end
```

Fig. 12: Transformed Derivative Program for Example 1 Showing from where Each Line is Printed.

program, the object $\mathcal{Y}_0$ must be re-mapped to the object $\mathcal{Y}_{l,o}$ prior to the printing evaluation of the loop in the intermediate program. The result of the printing evaluation of the intermediate program for $N = 10$ is seen in Fig. 14. In particular, Fig. 14 shows the re-mapping of the object $\mathcal{Y}_0$ to $\mathcal{Y}_{l,o}$ immediately prior to the loop. Additionally, it is seen

User Program

```
function y = SpeelFun(x)

y = 1;

for I = 1:length(x)

  y = y*x(I);

end
```

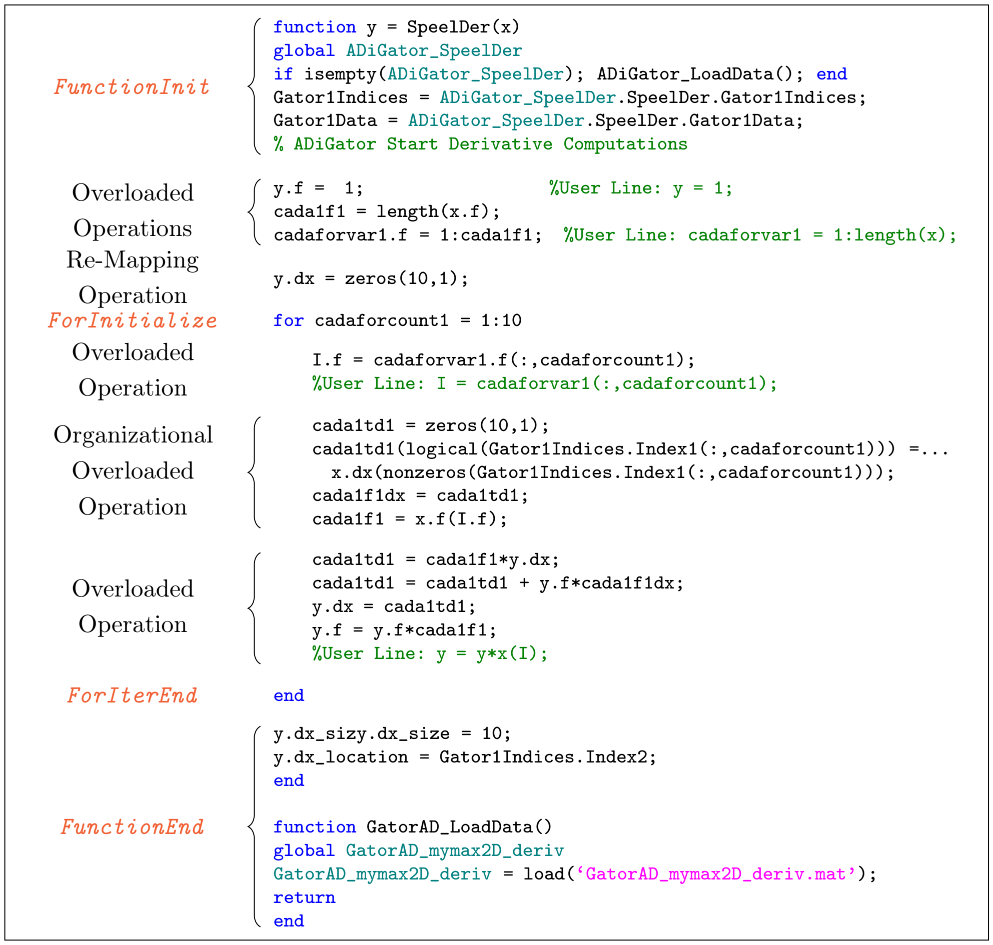
Intermediate Program

```
function [adigatorFunInfo, adigatorOutputs] = ...
    adigatortempfun_1(adigatorFunInfo,adigatorInputs)
[flag, adigatorFunInfo, adigatorInputs] = ...
    FunctionInit(1,adigatorFunInfo,adigatorInputs);
if flag; adigatorOutputs = adigatorInputs; return; end;
x = adigatorInputs{1};

y = 1;
y = VarAnalyzer('y = 1;',y,'y',0);

% ADiGator FOR Statement #1: START
cadaLoopVar_1 = 1:length(x);
cadaLoopVar_1 = ...
    VarAnalyzer('cadaLoopVar_k = 1:length(x);',cadaLoopVar_k,'cadaLoopVar_k',0);
[adigatorForVar_1, ForEvalStr, ForEvalVar] = ForInitialize(1,cadaLoopVar_1);
if not(isempty(ForEvalStr)); cellfun(@eval,ForEvalStr); end
for adigatorForVar_1_i = adigatorForVar_1
    cadaForCount_1 = ForIterStart(1,adigatorForVar_1_i);
    I = cadaLoopVar_1(:,cadaForCount_1);
    I = VarAnalyzer('I = cadaLoopVar_1(:,cadaForCount_1);',I,'I',0);
    y = y*x(I);
    y = VarAnalyzer('y = y*x(I);',y,'y',0);
    [ForEvalStr, ForEvalVar]= ForIterEnd(1,adigatorForVar_1_i);
end
if not(isempty(ForEvalStr)); cellfun(@eval,ForEvalStr); end
% ADiGator FOR Statement #1: END

adigatorOutputs = {y};
[adigatorFunInfo, adigatorOutputs] = ...
    FunctionEnd(1,adigatorFunInfo,adigatorOutputs);
```

Fig. 13: User Source-to-Intermediate-Source Transformation for Speelpenning Problem.

that the derivative variable `cada1f1dx` (which results from the aforementioned reference within the loop) is of length 10, where the reference and assignment of `cada1f1dx` depends upon the loop iteration. This reference and assignment is made possible by the index matrix **K** who is assigned to the variable `Gator1Indices.Index1` within the derivative program.

Rolled vs. Unrolled Loops for Speelpenning Problem. The following important issue arises in this example due to the fact that the loop remains rolled in the derivative code. Specifically, by imposing the overmapping scheme and maintaining a rolled loop in the derivative code, all derivative computations are forced to be dense on vectors of length N. If, on the other hand, the loop had been unrolled, all derivative computations on iteration i would have been performed sparsely on vectors of length i. Unrolling the loop, however, increases the size of the derivative program. To investigate the trade-off between between a rolled and an unrolled loop, consider for this example the Jacobian-to-function evaluation ratio, CPU(**Jf**)/CPU(**f**), for different values of N using a derivative code generated with a function code that contains a rolled version of the loop and a second derivative code generated using a function code that contains an unrolled version of the loop. The values of CPU(**Jf**)/CPU(**f**) for both the rolled and unrolled cases are shown in Table III for $N = (10, 100, 1000, 10000)$. It is seen that unrolling the loops results in a more efficient derivative code, but this increase in computational efficiency comes at the expense of generating a significantly larger derivative file. On the other hand, it is seen that, in the worst case ($N = 100$), the evaluation of the rolled loop is only about four times slower than that of the unrolled loop.



```
function y = SpeelDer(x)
global ADiGator_SpeelDer
if isempty(ADiGator_SpeelDer); ADiGator_LoadData(); end
Gator1Indices = ADiGator_SpeelDer.SpeelDer.Gator1Indices;
Gator1Data = ADiGator_SpeelDer.SpeelDer.Gator1Data;
% ADiGator Start Derivative Computations

y.f =  1;                  %User Line: y = 1;
cada1f1 = length(x.f);
cadaforvar1.f = 1:cada1f1;  %User Line: cadaforvar1 = 1:length(x);

y.dx = zeros(10,1);

for cadaforcount1 = 1:10

    I.f = cadaforvar1.f(:,cadaforcount1);
    %User Line: I = cadaforvar1(:,cadaforcount1);

    cada1td1 = zeros(10,1);
    cada1td1(logical(Gator1Indices.Index1(:,cadaforcount1))) =...
      x.dx(nonzeros(Gator1Indices.Index1(:,cadaforcount1)));
    cada1f1dx = cada1td1;
    cada1f1 = x.f(I.f);

    cada1td1 = cada1f1*y.dx;
    cada1td1 = cada1td1 + y.f*cada1f1dx;
    y.dx = cada1td1;
    y.f = y.f*cada1f1;
    %User Line: y = y*x(I);

end

y.dx_sizy.dx_size = 10;
y.dx_location = Gator1Indices.Index2;
end

function GatorAD_LoadData()
global GatorAD_mymax2D_deriv
GatorAD_mymax2D_deriv = load('GatorAD_mymax2D_deriv.mat');
return
end
```

Fig. 14: Derivative Program for Speelpenning Problem for $N = 10$ Showing Origin of Each Printed Line of Code.

Table III: Sizes of *GatorAD* Generated Derivative Programs and Ratios of Jacobian-to-Function Computation time, CPU(**Jf**)/CPU(**f**), for the Rolled and Unrolled Speelpenning Function. File sizes were computed as the sum of the sizes of the produced `.m` and `.mat` files, and CPU(**Jf**)/CPU(**f**) were obtained by averaging the values obtained over 1000 Jacobian and function evaluations.

N	CPU(**Jf**)/CPU(**f**) Ratio with Rolled Loop	CPU(**Jf**)/CPU(**f**) Ratio with Unrolled Loop	Program Sizes (kB) with Rolled Loop	Program Sizes (kB) with Unrolled Loop
10	223	65	1.5	2.9
100	1255	297	1.9	23.3
1000	2522	780	6.3	370.8
10000	5960	3831	60.4	82170.3

Example 3: Polynomial Data Fitting

Consider the problem of determining the coefficients of the m-degree polynomial $p(x) = p_1 + p_2 x + p_3 x^2 + \cdots + p_m x^{m-1}$ that best fits the points (x_i, d_i), $(i = 1, \ldots, n)$, $(n > m)$, in the least squares sense. This polynomial data fitting problem leads to an overdetermined linear system $\mathbf{Vp} = \mathbf{d}$, where $\mathbf{V}$ is the Vandermonde matrix,

$$\mathbf{V} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{m-1} \\ \vdots & \vdots & & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{m-1} \end{bmatrix}. \tag{16}$$

The problem of computing the Jacobian $\mathbf{Jp}(\mathbf{x})$ was presented by [Bischof et al. 2002] as a way of demonstrating both the AD tool *ADiMat* and powerful MATLAB operators (particularly, `mldivide`). This same example was considered by [Forth 2006] to demonstrate the effectiveness of the *MAD* sparse forward mode AD class `fmad`. In this example *ADiGator* is compared with the forward sparse mode of *MAD* and the overloaded vector forward mode of *ADiMat*. The code which computes $\mathbf{p}$ as well as the *ADiGator* transformed derivative code for this problem may be seen in Fig. 15.

Table IV shows the Jacobian-to-function evaluation times for the *ADiGator*, *MAD*, and *ADiMat* methods. Because the Jacobian $\mathbf{Jp}(\mathbf{x})$ is full, *MAD* was found to be most efficient in the sparse forward mode and *ADiMat* was found to be most efficient when evaluating the forward mode generated code on derivative objects of the `opt_derivclass`. Furthermore, it is seen that, for all chosen values of n, the derivative program generated by *ADiGator* is evaluated in less time than the time required to evaluate the original function file on sparse `fmad` objects or the time required to evaluate the derivative file generated by *ADiMat* on `opt_derivclass` objects. It is noted, however, that all three AD tools compute the derivative of the backslash operator in a different manner. As seen in Fig. 15, the method employed by the *ADiGator* tool involves squaring the Vandermonde matrix. This approach produces efficient derivative files, however, the accuracy of the computed derivatives deteriorates as the condition number of the Vandermonde matrix increases. In order to analyze the efficiency of the source transformation for this problem, the average function evaluation time as well as the average *ADiGator* source transformation times are also given in Table IV. It is then seen that as the problem increases in size the *ADiGator* source transformation times do not increase significantly, where the increase in times is only due to the increase in non-zero derivative locations which must be propagated via the **cada** class. While perhaps not the ideal comparison,[4] it is noted that the average time required to perform the source transformation using *ADiMat* was 0.936 s, while the source transformation time using *AdiGator* took a maximum of 0.474 s (at $n = 2560$).

An alternative approach for evaluating the efficiency of the *ADiGator* transformation process is to determine the number of Jacobian evaluations required in order to overcome the overhead associated with the source transformation. Specifically, Table IV shows that, for $n = 10$ and $n = 2560$, the Jacobian would need to be evaluated, at least 26 and four times, respectively, before *ADiGator* consumes less time when compared with *MAD*.

[4]In order to perform source transformation using *ADiMat*, the original source code must be sent to a transformation server which performs the transformation and then sends back the derivative code. Thus, there are a few unknown factors which may not be accounted for in the direct comparison of *ADiMat* code generation times to *ADiGator* code generation times.

Function Program

```
function p= fit(x, d, m)
% FIT -- Given x and d, fit() returns p
% such that norm(V*p-d) = min, where
% V = [1, x, x.^2, ... x.^(m-1)].

dim_x = size(x, 1);
if dim_x < m
  error('x must have at least m entries');
end

V = zeros(dim_x,m);
for i = 1:m
  V(:,i) = x.^(i-1);
end

p = V \ d;
```

***ADiGator* Generated Program**

```
function p = fit_x(x,d,m)
global ADiGator_fit_x
if isempty(ADiGator_fit_x); ADiGator_LoadData(); end
Gator1Indices = ADiGator_fit_x.fit_x.Gator1Indices;
Gator1Data = ADiGator_fit_x.fit_x.Gator1Data;
% ADiGator Start Derivative Computations
%User Line: % FIT -- Given x and d, fit() returns p
%User Line: % such that norm(V*p-d) = min, where
%User Line: % V = [1, x, x.^2, ... x.^(m-1)].
dim_x.f = size(x.f,1);
%User Line: dim_x = size(x, 1);
cadaconditional1 = lt(dim_x.f,m);
%User Line: cadaconditional1 = dim_x < m
V.f = zeros(dim_x.f,m);
%User Line: V = zeros(dim_x,m);
cadaforvar1.f = 1:m;
%User Line: cadaforvar1 = 1:m;
V.dx = zeros(30,1);
for cadaforcount1 = 1:4
    i.f = cadaforvar1.f(:,cadaforcount1);
    %User Line: i = cadaforvar1(:,cadaforcount1);
    cada1f1 = i.f - 1;
    cadaconditional1 = cada1f1;
    if cadaconditional1
        cada1f2dx = cada1f1.*x.f(:).^(cada1f1-1).*x.dx;
    else
        cada1f2dx = zeros(10,1);
    end
    cada1f2 = x.f.^cada1f1;
    V.dx(logical(Gator1Indices.Index1(:,cadaforcount1))) = ...
        cada1f2dx(nonzeros(Gator1Indices.Index1(:,cadaforcount1)));
    V.f(:,i.f) = cada1f2;
    %User Line: V(:,i) = x.^(i-1);
end
cada1tf3 = V.f\ d;
cada1td1 = sparse(Gator1Indices.Index2,Gator1Indices.Index3,V.dx,4,100);
cada1td1 = cada1tf3.'*cada1td1;
cada1td1 = cada1td1(:);
cada1td3 = full(cada1td1(Gator1Indices.Index4));
cada1tf4 = V.f.';
cada1td1 = zeros(10,10);
cada1td1(Gator1Indices.Index5) = cada1td3;
cada1td1 = cada1tf4*cada1td1;
cada1td1 = cada1td1(:);
cada1td4 = cada1td1(Gator1Indices.Index6);
cada1tf4 = (V.f*cada1tf3 - d).';
cada1td1 = sparse(Gator1Indices.Index7,Gator1Indices.Index8,V.dx,10,40);
cada1td1 = cada1tf4*cada1td1;
cada1td1 = cada1td1(:);
cada1td5 = full(cada1td1(Gator1Indices.Index9));
cada1td3 = cada1td4;
cada1td3(Gator1Indices.Index10) = cada1td3(Gator1Indices.Index10) + cada1td5;
cada1tf4 = -(V.f.'*V.f);
cada1td1 = zeros(4,10);
cada1td1(Gator1Indices.Index11) = cada1td3;
cada1td1 = cada1tf4\ cada1td1;
cada1td1 = cada1td1(:);
p.dx = cada1td1(Gator1Indices.Index12);
p.f = cada1tf3;
%User Line: p = V \ d
p.dx_size = [4,10];
p.dx_location = Gator1Indices.Index13;
end

function ADiGator_LoadData()
global ADiGator_fit_x
ADiGator_fit_x = load('fit_x.mat');
return
end
```

Fig. 15: Function Program and *GatorAD* Generated Derivative Program ($m = 4, n = 10$, fixed) for Polynomial Data Fitting Example.

Table IV: Ratio of Jacobian-to-Function Computation Time, CPU($\mathbf{Jp}(\mathbf{x})$)/CPU($\mathbf{p}(\mathbf{x})$), for Example 4 ($m = 4$) Using *ADiGator*, *MAD*, and *ADiMat*, Together with *ADiGator* Code Generation Times and Function Evaluation Times. *MAD* was used in the sparse forward mode and *ADiMat* was used in the vector forward mode with the derivative class `opt_derivclass`. All times CPU($\mathbf{Jp}(\mathbf{x})$) and CPU($\mathbf{p}(\mathbf{x})$) were obtained by averaging over 100 trials and *ADiGator* file generation times were obtained by averaging over 10 trials. It is noted that the entries for *ADiMat* are omitted at $n = 1280$ and $n = 2560$ due to large Jacobian CPU times.

Problem Size	CPU($\mathbf{Jp}(\mathbf{x})$)/CPU($\mathbf{p}(\mathbf{x})$)			*ADiGator* File	CPU($\mathbf{p}(\mathbf{x})$)
n	*ADiGator*	*MAD*	*ADiMat*	Gen. Time (s)	(ms)
10	5	49	849	0.114	0.102
20	7	50	1031	0.116	0.110
40	7	48	1887	0.120	0.117
80	7	44	4115	0.118	0.131
160	9	48	23231	0.120	0.130
320	11	46	103882	0.124	0.162
640	17	52	604633	0.140	0.241
1280	53	139	-	0.227	0.398
2560	109	300	-	0.474	0.675

Example 4: Sparse Non-Linear Programming

Consider the following nonlinear programming problem (NLP) that arises from the discretization of a scaled version of the optimal control problem described in [Darby et al. 2011] using a multiple-interval formulation of the Legendre-Gauss-Radau (LGR) orthogonal collocation method as described in Patterson et al. [2013]. The NLP decision vector $\mathbf{z} \in \mathbb{R}^{4N+4}$ is given as

$$\mathbf{z} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{u}, \beta], \tag{17}$$

where N is a parameter that defines the number of collocation points [Garg et al. 2011; Garg et al. 2010; Garg et al. 2011; Patterson and Rao 2012], $\mathbf{x}_1 \in \mathbb{R}^{N+1}$, $\mathbf{x}_2 \in \mathbb{R}^{N+1}$, $\mathbf{x}_3 \in \mathbb{R}^{N+1}$, $\mathbf{u} \in \mathbb{R}^{N}$, and $\beta \in \mathbb{R}$. Furthermore, let $\mathbf{f} : \mathbb{R}^{4N+4} \to \mathbb{R}^{3N}$ be defined as

$$\begin{aligned} f_i(\mathbf{z}) &= \left[\sum_{k=1}^{N+1} D_{i,k} x_{1k} - \frac{\beta}{2} x_{2i} \sin x_{3i}\right], \quad (i = 1, \ldots, N), \\ f_{N+i}(\mathbf{z}) &= \left[\sum_{k=1}^{N+1} D_{i,k} x_{2k} - \frac{\beta}{2}\left(\frac{\zeta - \theta}{c_1} - c_2 \sin x_{3i}\right)\right], \quad (i = 1, \ldots, N), \\ f_{2N+i}(\mathbf{z}) &= \left[\sum_{k=1}^{N+1} D_{i,k} x_{3k} - \frac{\beta}{2} \frac{c_2}{x_{2i}} (u_i - \cos x_{3i})\right], \quad (i = 1, \ldots, N), \end{aligned} \tag{18}$$

where

$$\begin{aligned} \zeta &= \zeta(T(h), h, v), \\ \theta &= \theta(T(h), \rho(h), h, v), \end{aligned} \tag{19}$$

and the functions ζ, θ, T, and ρ, are evaluated at values $h = x_{1i}$ and $v = x_{2i}$, where x_{1i} and x_{2i} are the i^{th} elements of the vectors $\mathbf{x}_1$ and $\mathbf{x}_2$, respectively. Furthermore, the quantity $[D_{ik}] \in \mathbb{R}^{N \times (N+1)}$ in Eq. (18) is the LGR differentiation matrix [Garg et al. 2011; Garg et al. 2010; Garg et al. 2011; Patterson and Rao 2012]. It is noted that $N = N_k K$, where K is the number of mesh intervals that the problem is divided via using the multiple-interval LGR collocation method. The objective of the NLP which arises from the discretization is to minimize the cost function

$$J = \beta \tag{20}$$

subject to the nonlinear algebraic constraints

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{21}$$

and the simple bounds on the decision variables

$$\mathbf{z}_{\min} \leq \mathbf{z} \leq \mathbf{z}_{\max}, \tag{22}$$

where

$$\begin{aligned} x_1 &= b_1 \;,\; x_{N+1} = b_2, \\ x_{N+2} &= b_3 \;,\; x_{2N+2} = b_4 \\ x_{2N+3} &= b_5 \;,\; x_{3N+3} = b_6. \end{aligned} \tag{23}$$

A key aspect of this example is the modification of the functions $T(h)$ and $\rho(h)$ from smooth functions (as used in Darby et al. [2011]) to the following piecewise continuous functions taken from Ref. NOAA [1976]:

$$\rho(h) = c_3 \frac{p(h)}{T(h)} \tag{24}$$

Table V: Constants and Parameters for Example 4.

c_1	3.9240×10^2	c_2	1.6818×10^4	c_3	8.6138×10^1
c_4	2.8814×10^2	c_5	6.4900×10^0	c_6	4.0519×10^9
c_7	2.8808×10^2	c_8	5.2560×10^0	c_9	2.1664×10^2
c_{10}	9.0600×10^8	c_{11}	1.7300×10^0	c_{12}	1.5700×10^{-1}
c_{13}	6.0000×10^{-5}	c_{14}	4.00936×10^0	c_{15}	2.2000×10^0
b_1	0	b_2	2.0000×10	b_3	2.6000×10
b_4	5.9000×10^0	b_5	0	b_6	0

where

$$(T(h), p(h)) = \begin{cases} \left(c_4 - c_5 h, c_6 \left[\frac{T(h)}{c_7}\right]^{c_8}\right) & , \ h < 11, \\ (c_9, c_{10} e^{c_{11} - c_{12} h}) & , \ \text{otherwise}. \end{cases} \tag{25}$$

In the implementation considered in this example, Eq. (25) is represented by a conditional statement nested within a `for` loop. Figure 16 shows the MATLAB code that computes the function $\mathbf{f}(\mathbf{z})$, where it is seen that the source code contains multiple loops, an indeterminate conditional statement (nested within a loop) and a loop iteration dependent conditional statement. Table V shows the constants and parameters used in this example.

In order to solve the NLP of Eqs. (20)–(22), typically either a first-derivative (quasi-Newton) or a second-derivative (Newton) NLP solver is used. In a first-derivative solver the objective gradient and constraint Jacobian, $\mathbf{Jf}(\mathbf{z})$, are used together with a dense quasi-Newton approximation of the Lagrangian Hessian. In a second-derivative solver, the objective gradient and constraint Jacobian are used together with the Hessian of the NLP Lagrangian, where in this case the Lagrangian is given by

$$L = \beta + \boldsymbol{\lambda}^T \mathbf{f}(\mathbf{z}). \tag{26}$$

In this example we now concern ourselves with the computation of both the constraint Jacobian, $\mathbf{Jf}(\mathbf{z})$, and the Lagrangian Hessian, $\mathbf{H}L(\mathbf{z})$. Moreover, the efficiencies with which the method of this paper generates the constraint Jacobian and Lagrangian Hessian source code is presented along with the computational efficiency of the resulting derivative code. In order to analyze the performance of the method as the aforementioned NLP increases in size, the number of LGR points in each mesh interval is set to four (that is, $N_k = 4$) and the number of mesh intervals, K, is varied. Thus, in this example, the total number of LGR points is always $N = 4K$. Finally, the comparative performance of the method developed in this paper against the well known MATLAB automatic differentiation tools *INTLAB*, *MAD*, and *ADiMat* is provided.

Table VI shows the ratio of the constraint Jacobian evaluation time to constraint function evaluation time, CPU(**Jf**)/CPU(**f**), using the derivative code generated by *ADiGator* alongside the values of CPU(**Jf**)/CPU(**f**) using *ADiMat*, *INTLAB*, and *MAD*, where *ADiMat* is used in both of its non-overloaded (scalar and vector) modes while being supplied the compressed seed matrix $\mathbf{S} \in \mathbb{R}^{(4N+4)\times 5}$, while *MAD* was used in the compressed mode with the same seed matrix. Also, it is noted that the compressed seed matrix, **S**, has a column dimension $1 + \max(N_k) = 5$, for any K used in this example. From Table VI it is seen that the Jacobian is computed in a smaller amount of time using the *ADiGator* generated code than any of the other methods. Moreover, it is seen that, for all values of K used, the time required to evaluate the derivative code generated by *ADiGator* (and produce the entire Jacobian) is slightly greater than the

Constraint Function $\mathbf{f}(\mathbf{z})$

```
function C = Constraints(z)
global probinfo

nLGR = probinfo.LGR.nLGR;
D    = probinfo.LGR.Dmatrix;

X  = z(probinfo.map.state);
U  = z(probinfo.map.control);
tf = z(probinfo.map.tf);

F       = Dynamics(X(1:nLGR,:),U);
Defects = D*X - tf/2*F;

C = Defects(:);
```

Lagrangian Function $L(\mathbf{z})$

```
function L = Lagrangian(lambda,z)
global probinfo
tf = z(probinfo.map.tf);
C  = Constraints(z);

L  = tf + lambda.'*C;
```

Dynamics Function

```
function daeout = Dynamics(x,u)
global probinfo

CONSTANTS = probinfo.CONSTANTS;
CoF       = CONSTANTS.CoF;

h = x(:,1); v = x(:,2); fpa = x(:,3);

c1  = 392.4;    c2  = 16818;     c3  = 86.138;
c4  = 288.14;   c5  = 6.49;      c6  = 4.0519e9;
c7  = 288.08;   c8  = 5.256;     c9  = 216.64;
c10 = 9.06e8;   c11 = 1.73;      c12 = 0.157;
c13 = 6e-5;     c14 = 4.00936;   c15 = 2.2;

zeros4over = h*0;
rho  = zeros4over; T    = zeros4over;
for i = 1:length(h)
  hi = h(i);
  if hi < 11
    Ti = c4 - c5*hi;
    p     = c6*(Ti./c7).^c8;
  else
    Ti = c9;
    p     = c10* exp(c11 - c12*hi);
  end
  rho(i)  = c3*p./Ti;
  T(i) = Ti;
end
q = 0.5.*rho.*v.*v.*c13;

a = c14*sqrt(T);       M = v./a;
Mp = cell(1,6);
for i = 1:6
  Mp{i} = M.^(i-1);
end

numeratorCD0 = zeros4over; denominatorCD0 = zeros4over;
numeratorK   = zeros4over; denominatorK   = zeros4over;
for i = 1:6
  Mpi = Mp{i};
  if i < 6
    numeratorCD0   = numeratorCD0   + CoF(1,i).*Mpi;
    denominatorCD0 = denominatorCD0 + CoF(2,i).*Mpi;
    numeratorK     = numeratorK     + CoF(3,i).*Mpi;
  end
  denominatorK = denominatorK  + CoF(4,i).*Mpi;
end
Cd0 = numeratorCD0./denominatorCD0;
K   = numeratorK./denominatorK;
FD  = q.*(Cd0+K.*((c2^2).*(c1^2)./(q.^2)).*(u.^2));

FT = zeros4over;
for i = 1:6
  ei = zeros4over;
  for j = 1:6
    ei = ei + CoF(4+j,i).*Mpj;
  end
  FT = FT + ei.*h.^(i-1);
end
FT = FT.*c1/c15;

hdot   = v.*sin(fpa);
vdot   = (FT-FD)/c2 - c1*sin(fpa);
fpadot = c1*(u-cos(fpa))./v;

daeout = [hdot vdot fpadot];
end
```

Fig. 16: Function Program for the NLP in Example 4.

time required to evaluate the derivative code generated by *ADiMat* in the scalar mode (and produce a single column of the compressed Jacobian).

Table VI: Ratio of Jacobian-to-Function Computation Time, CPU($\mathbf{Jf}$)/CPU($\mathbf{f}$), for Example 4 Using *ADiGator*, *ADiMat*, *INTLAB* and *MAD*. *ADiMat* was supplied the compressed seed matrix in both the scalar forward and non-overloaded vector forward modes and *MAD* was used in the compressed forward mode and the sparse forward mode. All times CPU($\mathbf{Jf}$) and CPU($\mathbf{f}$) were obtained by taking the average over 100 Jacobian and function evaluations.

NLP Size		CPU($\mathbf{Jf}(\mathbf{z})$)/CPU($\mathbf{f}(\mathbf{z})$)					
K	$N = 4K$	*ADiGator*	*ADiMat* (scalar)	*ADiMat* (vector)	*INTLAB*	*MAD* (comp)	*MAD* (sparse)
4	16	17	79	142	149	188	186
8	32	21	93	210	220	275	276
16	64	29	116	322	332	417	421
32	128	41	151	502	511	636	650

Table VII now shows the ratio of Lagrangian Hessian evaluation time to Lagrangian function evalaution, CPU($\mathbf{H}L$)/CPU(L), using the derivative code generated by *ADiGator* alongside the values of CPU($\mathbf{H}L$)/CPU(L) using *ADiMat*, *INTLAB* and *MAD*. Unlike the constraint Jacobian, the Lagrangian Hessian is incompressible and as such *MAD* is only used in the sparse forward over forward mode. The non-overloaded modes of *ADiMat* may not be used to compute the Lagrangian Hessian as only the diagonal elements of Hessians may be computed via strip-mining and, currently, source transformation may not be recursively called on *ADiMat* generated code produced for the non-overloaded vector mode (as it is written in terms of *ADiMat* specific run-time functions). The computation of the Lagrangian Hessian using *ADiMat* was thus found to be most efficient when evaluating a Lagrangian gradient code generated in the reverse mode on a set of overloaded objects (the default mode of the *ADiMat* function `admHessian`). From Table VII it is again seen that the *ADiGator* generated code is more efficient than the other methods. It is also seen, however, that this efficiency appears to dissipate (when compared to *INTLAB*) as the NLP grows in size.

Table VII: Ratio of Hessian-to-Function Computation Time, CPU($\mathbf{H}L(\mathbf{z})$)/CPU($\mathbf{f}(\mathbf{z})$), for Example 4 Using *ADiGator*, *ADiMat*, *INTLAB* and *MAD*. *MAD* was used in the sparse forward over forward mode and *ADiMat* was used in the forward operator overloading over reverse source transformation mode. All times CPU($\mathbf{H}L(\mathbf{z})$) and CPU($L(\mathbf{z})$) were obtained by taking the average over 100 Jacobian and function evaluations.

NLP Size		CPU($\mathbf{H}L(\mathbf{z})$)/CPU($L(\mathbf{z})$)			
K	$N = 4K$	*ADiGator*	*INTLAB*	*MAD*	*ADiMat*
4	16	32	198	708	5202
8	32	48	270	1132	9590
16	64	93	382	2287	21206
32	128	330	573	6879	55697

To this point in the example it has been shown that the Jacobian and Hessian programs generated by the method of this paper are efficient when compared with other MATLAB AD tools. The expense of the code generation then comes into question. The time required to generate the constraint Jacobian and Lagrangian Hessian derivative files using *ADiGator* are now given in Table VIII as well as the constraint and Lagrangian function evaluation times. Unlike the previous example, there are two reasons for the increase in derivative file generation times seen in Table VIII. Namely, the increase in time is both due to an increase in the number of propagated non-zero derivative locations, together with an increase in the number of required overloaded operations. That is, since the dimension N is looped upon in the dynamics function of Fig. 16, as N increases the number of required overloaded operations also increases. In an attempt to quantify the efficiency of the transformation process, we first compare the generation times of Table VIII to similar *ADiMat* source transformations. In order to do so, we computed the average time to generate constraint Jacobian and Lagrangian Hessian derivative files using the forward mode of *ADiMat*. These times are now given as 1.40s and 3.42s, respectively, and are thus less than those required by *ADiGator*.

Table VIII: *ADiGator* Constraint Jacobian and Lagrangian Hessian Code Generation Times with Constraint and Lagrangian Function Evaluation Times. The times taken to generate the constraint Jacobian and Lagrangian Hessian files were averaged over 10 trials, where the Lagrangian Hessian generation time is the time taken to perform two successive source transformations, once on the Lagrangian file and once on the resulting derivative file. The constraint and Lagrangian function evaluation times were averaged over 100 trials.

NLP Size		*ADiGator* Code Generation Times (s)		Function Evaluation Times (ms)	
K	$N = 4K$	$\mathbf{Jf}$	$\mathbf{H}L$	$\mathbf{f}$	L
4	16	1.70	6.87	0.381	0.418
8	32	2.02	8.44	0.400	0.435
16	64	2.72	11.54	0.480	0.482
32	128	4.17	17.90	0.537	0.571

In order to further quantify the efficiency of the method of this paper applied to this example, we solved the NLP of Eqs. (20)–(22) using the NLP solver *IPOPT* [Biegler and Zavala 2008; Waechter and Biegler 2006]. *IPOPT* was used in both the quasi-Newton and Newton modes with the following initial guess of the NLP decision vector:

$$\begin{aligned} \mathbf{x}_1 &= \texttt{linspace}(b_1, b_2, N+1) \\ \mathbf{x}_2 &= \texttt{linspace}(b_3, b_4, N+1) \\ \mathbf{x}_3 &= \texttt{linspace}(b_5, b_6, N+1) \\ \mathbf{u} &= \texttt{linspace}(0, 0, N) \\ \beta &= 175, \end{aligned} \tag{27}$$

where the function $\texttt{linspace}(a, b, M)$ provides a set of M linearly equally-spaced points between a and b. The number of constraint Jacobian evaluations required to solve the problem in the quasi-Newton mode and the number of constraint Jacobian and Lagrangian Hessian evaluations required to solve the problem in the Newton mode were then recorded. The recorded values were then used to predict the amount of derivative computation time that would be required when supplying *IPOPT* with derivatives using the *ADiGator*, *ADiMat*, *INTLAB*, and *MAD* tools. The results are now given in Tables IX and X, respectively, where the total derivative computation time was computed as the time to evaluate the Jacobian/Hessian the required amount of times (using the average evaluation times of Tables VI–VII) plus any required source transformation overhead. Moreover, it is also noted that the total *ADiGator* time of Table X does not reflect the time required to generate the constraint Jacobian file since the constraint

Jacobian file is produced in the process of generating the Lagrangian Hessian file. From Table IX it may be seen that, even though the majority of the computation time required by *ADiGator* is spent in the file generation, that the expense of the file generation is worth the time saved in the solving of the NLP. From Table X it may be seen that, for the values of $K = 4$ and $K = 8$, the overhead required to generate the Lagrangian Hessian file using *ADiGator* outweighs the time savings at NLP solve time, and thus *INTLAB* is the most efficient. At the values of $K = 16$ and $K = 32$, however, due to a larger number of required Jacobian and Hessian evaluations, *ADiGator* becomes the more efficient tool.

Table IX: Total Derivative Computation Time to Solve NLP of Example 4 Using *IPOPT* in First-Derivative Mode. Derivative computation times were computed as the estimated total time to perform the number of required Jacobian evaluations plus any source transformation overhead. The transformation overhead of *ADiGator* is the Jacobian code generation time shown in Table VIII and the transformation overhead of *ADiMat* is the time required to generate the forward mode Jacobian code (1.40s). Estimated Jacobian evaluation times of *ADiMat* and *MAD* were computed by using the average times of the scalar compressed and compressed modes, respectively, from Table VI.

NLP Size		Jacobian	Derivative Computation Time (s)			
K	$N = 4K$	Evaluations	*ADiGator*	*ADiMat*	*INTLAB*	*MAD*
4	16	62	2.1	3.3	3.5	4.4
8	32	98	2.8	5.1	8.6	10.8
16	64	132	4.4	8.2	19.6	24.6
32	128	170	7.9	15.1	46.7	58.1

Table X: Total Derivative Computation Time to Solve NLP of Example 4 Using *IPOPT* in Second-Derivative Mode. Derivative computation times were computed as the total time to perform the number of required Jacobian and Hessian evaluations plus any source transformation overhead. The transformation overhead of *ADiGator* is the average Hessian code generation time shown in Table VIII and the transformation overhead of *ADiMat* is the average time required to generate the forward mode constraint Jacobian code (1.40s) together with the average time required to generate the reverse mode Lagrangian gradient code (5.75s). Estimated Jacobian evaluation times of *ADiMat* and *MAD* were computed by using the average times of the scalar compressed and compressed modes, respectively, from Table VI.

NLP Size		Jacobian	Hessian	Derivative Computation Time (s)			
K	$N = 4K$	Evaluations	Evaluations	*ADiGator*	*ADiMat*	*INTLAB*	*MAD*
4	16	31	30	7.5	73.3	4.2	11.1
8	32	39	38	9.6	167.3	7.9	23.0
16	64	53	52	14.5	541.6	17.5	67.2
32	128	152	150	49.5	4790.8	90.8	641.2

7. DISCUSSION

The four examples given in Section 6 demonstrate both the utility and the efficiency of the method presented in this paper. The first example showed the ability of the method to perform source transformation on a user program containing a conditional statement and gave an in depth look at the processes which must take place to do so. The second example demonstrated the manner in which a user program that contains a loop can be transformed into a derivative program that contains the same loop. The second example also provided a discussion of the trade-offs between using a rolled and an unrolled loop. In particular, it was seen in the second example that using an unrolled loop led to a more efficient derivative code at the expense of generating a much larger size derivative file, while using a rolled loop led to slightly less efficient but much more aesthetically pleasing derivative code and a significantly smaller derivative file. The third example showed the efficiency of the method when applied to a large dense problem and compared to the tools of *MAD* and *ADiMat*. It is also seen that the method of this paper is not the most preferred choice if it is required to compute the derivative only a smaller number of times. On the other hand, in applications where a large number of function evaluations is required, the method of this paper is more efficient than other well known automatic differentiation tools. The fourth example shows the performance of the method on a large sparse nonlinear programming problem where the constraint function contains multiple levels of flow control. From this fourth example it is seen that the method can be used to generate efficient first- and second-order derivative code of programs containing loops and conditional statements. Furthermore, it is shown that the presented method is particularly appealing for problems requiring a large number of derivative evaluations.

One aspect of the *CADA* method which was emphasized in [Patterson et al. 2013] was its ability to be repeatably applied to a program, thus creating second- and higher order derivative files. It is noted that, similar to *CADA*, the method of this paper generates standalone derivative code and is, thus, repeatable if higher-order derivatives are desired. Unlike the method of Patterson et al. [2013], however, *ADiGator* has the ability to recognize when it is performing source transformation on a file that was created by *ADiGator* and has the ability to recognize the naming scheme used in the previously generated file. This awareness enables the method to eliminate any redundant 1^{st} through $(n-1)^{th}$ derivative calculations in the n^{th} derivative file. For example, if source transformation were performed twice on a function `y = sin(x)` such that the second transformation is done *without* knowledge of the first derivative transformation, the source transformation would be performed as follows:

$$\texttt{y = sin(x)} \begin{cases} \texttt{dy = cos(x)} \begin{cases} \texttt{ddy = -sin(x)} \\ \texttt{dy = cos(x)} \end{cases} \\ \texttt{y = sin(x)} \begin{cases} \texttt{dy = cos(x)} \\ \texttt{y = sin(x)} \end{cases} \end{cases} \tag{28}$$

It is seen that, in the second derivative file, the first derivative would be printed twice, once as a function variable and once as a derivative variable. On the other hand, the method of this paper would have knowledge of the naming scheme used in the first derivative file, and would thus eliminate this redundant line of code in the file that contains the second derivative.

7.1. Limitations of the Approach

The method of this paper utilizes fixed input sizes and sparsity patterns to exploit sparsity at each required derivative computation and to reduce the required computations at run-time. The exact reasons which allow for the proposed method to generate efficient stand-alone derivative code also add limitations to the approach. The primary limitation being that derivative files may only be created for a fixed input size and sparsity pattern. Thus, if one wishes to change the input size, a new derivative file must be created. Moreover, the times required to generate derivative files are largely based upon the number of required overloaded evaluations in the intermediate program. Thus, if a loop is to run for many iterations (as was the case in the fourth example), the time required to generate the derivative files can become quite large. Requiring that all **cada** objects have a fixed size also limits the functions used in the user program to those which result in objects of a fixed size, thus, in general, logical referencing cannot be used. Another limitation comes from the fact that all objects in the intermediate program are forced to be overloaded, even if they are simply numeric values. Thus, any operation which the intermediate program is dependent upon must be overloaded, even if they are never used to operate on objects which contain derivative information.

8. CONCLUSIONS

A method has been described for generating derivatives of mathematical functions in MATLAB. Given a user program to be differentiated together with the information required to create **cada** instances of the inputs, the developed method may be used to generate derivative source code. The method employs a source transformation via operator overloading approach such that the resulting derivative code computes a sparse representation of the derivative of the user function whose input is a fixed size. A key aspect of the method is that it allows for the differentiation of MATLAB code where the function contains flow control statements. Moreover, the generated derivative code relies solely on the native MATLAB library and thus the process may be repeated to obtain n^{th}-order derivative files. The approach has been demonstrated on four examples and is found to be highly efficient at run-time when compared with well known MATLAB AD programs. Furthermore, while there does exist an inherent overhead associated with generating the derivative files, the overhead becomes less of a factor as the number of required derivative evaluations is increased.

REFERENCES

AUBERT, P., DI CÉSARÉ, N., AND PIRONNEAU, O. 2001. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science 3*, 197–208.

BENDTSEN, C. AND STAUNING, O. 1996. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark. aug.

BERZ, M. 1987. The differential algebra Fortran precompiler DAFOR. Tech. Report AT–3: TN–87–32, Los Alamos National Laboratory, Los Alamos, N.M.

BERZ, M., MAKINO, K., SHAMSEDDINE, K., HOFFSTÄTTER, G. H., AND WAN, W. 1996. COSY INFINITY and its applications in nonlinear dynamics. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, 363–365.

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 (March), 575–582.

BISCHOF, C., LANG, B., AND VEHRESCHILD, A. 2003. Automatic differentiation for MATLAB programs. *Proceedings in Applied Mathematics and Mechanics 2,* 1 Joh Wiley, 50–53.

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.

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 (March), 150–175.

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

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.

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

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

GARG, D., PATTERSON, M. A., DARBY, C. L., FRANÇOLIN, C., HUNTINGTON, G. T., HAGER, W. W., AND RAO, A. V. 2011. 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 (June), 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 (November), 1843–1851.

GIERING, R. AND KAMINSKI, T. 1996. Recipes for adjoint code construction. Tech. Rep. 212, Max-Planck-Institut für Meteorologie.

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 (April–June), 131–167.

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

HORWEDEL, J. E. 1991. GRESS, a preprocessor for sensitivity studies of Fortran programs. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, Philadelphia, PA, 243–250.

KHARCHE, R. V. 2012. Matlab automatic differentiation using source transformation.

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.

KUBOTA, K. 1991. PADRE2, a Fortran precompiler yielding error estimates and second derivatives. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, Philadelphia, PA, 251–262.

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

MICHELOTTI, L. 1991. MXYZPTLK: A C++ hacker's implementation of automatic differentiation. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, A. Griewank and G. F. Corliss, Eds. SIAM, Philadelphia, PA, 218–227.

NOAA. 1976. *U. S. standard atmosphere, 1976*. National Oceanic and Amospheric [sic] Administration : for sale by the Supt. of Docs., U.S. Govt. Print. Off.

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 (March–April), 364–377.

PATTERSON, M. A., WEINSTEIN, M. J., AND RAO, A. V. 2013. An efficient overloaded method for computing derivatives of mathematical functions in matlab. *ACM Transactions on Mathematical Software, 39,* 3 (July), 17:1–17:36.

PRYCE, J. D. AND REID, J. K. 1998. ADO1, a Fortran 90 code for automatic differentiation. Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England.

RHODIN, A. 1997. {IMAS} integrated modeling and analysis system for the solution of optimal control problems. *Computer Physics Communications 107,* 1?3, 21 – 38.

ROSTAING-SCHMIDT, N. 1993. Différentiation automatique: Application à un problème d'optimisation en météorologie. Ph.D. thesis, Université de Nice-Sophia Antipolis.

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

SHIRIAEV, D. AND GRIEWANK, A. 1996. Adol-f automatic differentiation of fortran codes. In *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 375–384.

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., AND PRYCE, J. D. 2003. Hierarchical automatic differentiation by vertex elimination and source transformation. In *Computational Science and Its Applications – ICCSA 2003,* *Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, Eds. Lecture Notes in Computer Science, vol. 2668. Springer, 115–124.

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 (March), 575–582.