A

Algorithm: ADiGator, a Toolbox for the Algorithmic Differentiation of Mathematical Functions in MATLAB Using Source Transformation via Operator Overloading

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

University of Florida
Gainesville, FL 32611

A toolbox called *ADiGator* is described for algorithmically differentiating mathematical functions in MATLAB. ADiGator performs source transformation via operator overloading using forward mode algorithmic differentiation and produces a derivative file that can be evaluated to obtain the derivative of the original function at a numeric value of the input. A convenient by product of the file generation is the sparsity pattern of the derivative function. Moreover, as both the input and output to the algorithm are source codes, the algorithm may be applied recursively to generate derivatives of any order. A key component of the algorithm is its ability to statically exploit derivative sparsity at the MATLAB operation level in order to improve run-time performances. The algorithm is applied to four different classes of example problems and is shown to produce run-time efficient derivative codes. Due to the static nature of the approach, the algorithm is well suited and intended for use with problems requiring many repeated derivative computations.

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

General Terms: Automatic Differentiation, Numerical Methods, MATLAB

Additional Key Words and Phrases: algorithmic differentiation, scientific computation, applied mathematics, chain rule, forward mode, overloading, source transformation

ACM Reference Format:
Weinstein, M. J. and Rao, A. V. 2015. Algorithm: ADiGator, a Toolbox for the Algorithmic Differentiation of Mathematical Functions in MATLAB Using Source Transformation via Operator Overloading. ACM Trans. Math. Soft. V, N, Article A (January YYYY), 32 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

The authors gratefully acknowledge support for this research from the U.S. Office of Naval Research (ONR) under Grants N00014-11-1-0068 and N00014-15-1-2048, from the U.S. Defense Advanced Research Projects Agency under Contract HR0011-12-C-0011, and from the U.S. National Science Foundation under grant CBET-1404767. **Disclaimer:** The views, opinions, and findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.
Distribution A. Approved for Public Release; Distribution Unlimited.
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

1. INTRODUCTION

The problem of computing accurate and efficient derivatives is one of great importance in the field of numerical analysis. The desire for a method that accurately and efficiently computes numerical derivatives automatically has led to the field of research known as automatic differentiation or as it has been more recently termed, algorithmic differentiation (AD). AD is defined as the process of determining accurate derivatives of a function defined by computer programs using the rules of differential calculus [Griewank 2008]. Assuming a computer program is differentiable, AD exploits the fact that a user program may be broken into a sequence of elementary operations, where each elementary operation has a corresponding derivative rule. Thus, given the derivative rules of each elementary operation, a derivative of the program is obtained by a systematic application of the chain rule, where any errors in the resulting derivative are strictly due to round-off.

Algorithmic differentiation may be performed either using the forward or reverse mode. In either mode, each link in the calculus chain rule is implemented until the derivative of the output dependent variables with respect to the input independent variables is obtained. The fundamental difference between the forward and reverse modes is the order in which the chain rule is applied. In the forward mode, the chain rule is applied from the input independent variables of differentiation to the final output dependent variables of the program, while in the reverse mode the chain rule is applied from the final output dependent variables of the program back to the independent variables of differentiation. Forward and reverse mode AD 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 and derivative values 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. In a source transformation approach, typically a compiler-type software is required to transform a user-defined function source code into a derivative source code, where the new program contains derivative statements interleaved with the function statements of the original program. The generated derivative source code may then be evaluated numerically in order to compute the desired derivatives.

Many applications that require the computation of derivatives are iterative (for example, nonlinear optimization, root finding, differential equation integration, estimation, etc.) and thus require the same derivative to be computed at many different points. In order for AD to be tractable for such applications, the process must be computationally efficient. It is thus often advantageous to perform an a priori analysis of the problem at compile-time in order to decrease derivative computation run times. Source transformation tools are therefore quite desirable due to their ability to perform optimizations at compile-time which then improve derivative computation run times. Typical optimizations performed by source transformation tools are those of dead code elimination and common sub-expression elimination.

Another way in which derivative run-time efficiencies may be gained is by the exploitation of derivative sparsity. When applying AD, one may view the chain rule as a sequence of matrix multiplications, where many of the matrices are inherently sparse. This inherent sparsity is typically exploited either at run-time by making use of dynamic sparse data structures, or at compile-time by utilizing matrix compression techniques. Using a set of dynamic data structures, each derivative matrix is represented by its non-zero values together with the locations of the non-zeros. The chain rule

is then carried out at run-time by performing sparse matrix multiplications. Thus, at each link in the chain rule, sparsity patterns are propagated, and only non-zero derivative elements are operated upon. For applications requiring many repeated derivative computations, non-zero derivative *values* change from one iteration to the next. Derivative sparsity patterns, however, are constant across all iterations. Thus, a dynamic approach to sparsity exploitation must perform redundant sparsity propagation computations at run-time. The typical alternative to a dynamic approach is to exploit sparsity by means of matrix compression. The most commonly used matrix compression technique is the Curtis-Powell-Reid (CPR) approach of Curtis et al. [1974], which has its roots in finite differencing. The CPR approach is based upon the fact that, given two inputs, if no output is dependent upon both inputs, then both inputs may be perturbed at the same time in order to approximate the output derivative with respect to each of the two inputs. Thus, if the output derivative sparsity pattern is known, it may be determined at compile-time which inputs may be perturbed at the same time. When used with finite-differencing, CPR compression effectively reduces the number of function evaluations required to build the output derivative matrix. When used with the forward mode of AD, CPR compression effectively reduces the column dimension (number of directional derivatives) of the matrices which are propagated and operated upon when carrying out the chain rule. Similar exploitations may be performed by reducing the row dimension of the matrices which are propagated and operated upon in the reverse mode. Unlike a dynamic approach, the use of matrix compression does not require any sparsity analysis to be performed at run-time. Rather, all sparsity analysis may be performed at compile-time in order to reduce derivative computation run times. Matrix compression techniques, however, are not without their flaws. In order to use matrix compression, one must first know the output derivative sparsity pattern. Moreover, only the sparsity of the program as a whole may be exploited, rather than sparsity at each link in the chain. This can pose an issue when output derivative matrices are incompressible (for instance, output matrices with a full row in the forward mode, or output matrices with a full column in the reverse mode), in which case one must partially separate the problem in order to take advantage of sparsity.

In recent years, MATLAB [Mathworks 2014] has become extremely popular as a platform for numerical computing due largely to its built in high-level matrix operations and user friendly interface. The interpreted nature of MATLAB and its high-level language make programming intuitive and debugging easy. The qualities that make MATLAB appealing from a programming standpoint, however, tend to pose problems for AD tools. In the MATLAB language, there exist many ambiguous operators (for example, $+$, $*$) which perform different mathematical procedures depending upon the shapes (for example, scalar, vector, matrix, etc.) of the inputs to the operators. Moreover, user variables are not required to be of any fixed size or shape. Thus, the proper mathematical procedure of each ambiguous operator must be determined at run-time by the MATLAB interpreter. This mechanism poses a major problem for both source transformation and operator overloaded AD tools. Source transformation tools must determine the proper rules of differentiation for all function operations at compile-time. Given an ambiguous operation, however, the corresponding differentiation rule is also ambiguous. In order to cope with this ambiguity, MATLAB source transformation AD tools must either determine fixed shapes for all variables, or print derivative procedures which behave differently depending upon the meaning of the corresponding ambiguous function operations. As operator overloading is applied at run-time, operator ambiguity is a non-issue when employing an operator overloaded AD tool. The mechanism that the MATLAB interpreter uses to determine the meanings of ambiguous operators, however, imposes a great deal of run-time overhead on operator overloaded tools.

The first comprehensive AD tool written for MATLAB was the operator overloaded tool, ADMAT [Coleman and Verma 1998a; 1998b]. The ADMAT implementation may be used in both the forward and reverse mode to compute gradients, Jacobians and Hessians. Later, the ADMAT tool was interfaced with the ADMIT tool [Coleman and Verma 2000], providing support for the computation of sparse Jacobians and Hessians via compression techniques. The next operator overloading approach was developed as a part of the INTLAB toolbox [Rump 1999], which utilizes MATLAB's `sparse` class in order to store and compute first and second derivatives, thus dynamically exploiting Jacobian/Hessian sparsity. More recently, the MAD package [Forth 2006] has been developed. While MAD also employs operator overloading, unlike previously developed MATLAB AD tools, MAD utilizes the `derivvec` class to store directional derivatives within instances of the `fmad` class. By utilizing a special class to store directional derivatives, the MAD toolbox is able to compute n^{th}-order derivatives by stacking overloaded objects within one another. MAD may be used with either sparse or dense derivative storage, with or without matrix compression. In addition to operator overloaded methods that evaluate derivatives at a numeric value of the input argument, the hybrid source transformation and operator overloaded package ADiMat [Bischof et al. 2003] has been developed. ADiMat employs source transformation to create a derivative source code using either the forward or reverse mode. 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. 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 2011], however, does not depend upon operator overloading but still maintains the efficiencies of the `derivvec` class.

The toolbox ADiGator (**A**utomatic **Di**fferentiation by **Gator**s) described in this paper performs source transformation via the non-classical methods of operator overloading and source reading for the forward mode algorithmic differentiation of MATLAB programs. Motivated by the iterative nature of the applications requiring numerical derivative computation, a great deal of emphasis is placed upon performing an a priori analysis of the problem at compile-time in order to minimize derivative computation run time. Moreover, the algorithm neither relies upon sparse data structures at run-time nor relies on matrix compression in order to exploit derivative sparsity. Instead, an overloaded class is used at compile-time to determine sparse derivative structures for each MATLAB operation. Simultaneously, the sparse derivative structures are exploited to print run-time efficient derivative procedures to an output source code. The printed derivative procedures may then be evaluated numerically in order to compute the desired derivatives. The resulting code is quite similar to that produced by the vertex elimination methods of Forth et al. [2004; Tadjouddine et al. [2003], yet the approach is unique. As the result of the source transformation is a stand-alone MATLAB procedure (that is, the resulting derivative code depends only upon the native MATLAB library at run-time), the algorithm may be applied recursively to generate n^{th}-order derivative programs. Hessian symmetry, however, is not exploited. Finally,

it is noted that the previous research given in Patterson et al. [2013] and Weinstein and Rao [2015] focused on the methods upon which the ADiGator tool is based, while this paper focuses on the software implementation of these previous methods and the utility of the software.

This paper is organized as follows. In Section 2, a row/column/value triplet notation used to represent derivative matrices is introduced. In Section 3, an overview of the implementation of the algorithm is given in order to grant the reader a better understanding of how to efficiently utilize the software as well as to identify various coding restrictions to which the user must adhere. Key topics such as the used overloaded class and the handling of flow control are discussed. In Section 4, a discussion is given on the use of overloaded objects to represent cell and structure arrays. In Section 5, a technique is presented which eliminates redundant derivative computations being printed when performing high-order derivative transformations. In Section 6, a discussion is given on the storage of indices upon which the generated derivative programs are dependent. In Section 7, a special class of vectorized functions is considered, where the algorithm may be used to transform vectorized function codes into vectorized derivative codes. In Section 8, the user interface to the ADiGator algorithm is described. In Section 9, the algorithm is tested against other well known MATLAB AD tools on a variety of examples. In Section 10, a discussion is given on the efficiency of the algorithm and finally, in Section 11, conclusions are drawn.

2. SPARSE DERIVATIVE NOTATIONS

The algorithm of this paper utilizes a row/column/value triplet representation of derivative matrices. In this section, the triplet representation is given for a general matrix function of a vector, $\mathbf{F}(\mathbf{x}) : \mathbb{R}^{n_x} \to \mathbb{R}^{q_f \times r_f}$. The derivative of $\mathbf{F}(\mathbf{x})$ is the three dimensional object, $\partial\mathbf{F}/\partial\mathbf{x} \in \mathbb{R}^{q_f \times r_f \times n_x}$. In order to gain a more tractable two-dimensional derivative representation, we first let $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^{m_f}$ be the one-dimensional transformation of the function $\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{q_f \times r_f}$,

$$\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \mathbf{F}_1(\mathbf{x}) \\ \vdots \\ \mathbf{F}_{r_f}(\mathbf{x}) \end{bmatrix}, \quad \mathbf{F}_k = \begin{bmatrix} F_{1,k}(\mathbf{x}) \\ \vdots \\ F_{q_f,k}(\mathbf{x}) \end{bmatrix}, \quad (k = 1, \ldots, r_f), \tag{1}$$

where $m_f = q_f r_f$. The *unrolled* representation of the three-dimensional derivative $\partial\mathbf{F}/\partial\mathbf{x}$ is then given by the two-dimensional Jacobian

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_{n_x}} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_{n_x}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{m_f}}{\partial x_1} & \frac{\partial f_{m_f}}{\partial x_2} & \cdots & \frac{\partial f_{m_f}}{\partial x_{n_x}} \end{bmatrix} \in \mathbb{R}^{m_f \times n_x}. \tag{2}$$

Assuming the first derivative matrix $\partial\mathbf{f}/\partial\mathbf{x}$ contains $p_x^f \leq m_f n_x$ *possible* non-zero elements, the row and column locations of the possible non-zero elements of $\partial\mathbf{f}/\partial\mathbf{x}$ are denoted by the index vector pair $(\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{f}}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f}$, where

$$\mathbf{i}_{\mathbf{x}}^{\mathbf{f}} = \begin{bmatrix} i_{\mathbf{x}}^{\mathbf{f}}(1) \\ \vdots \\ i_{\mathbf{x}}^{\mathbf{f}}(p_x^f) \end{bmatrix}, \ \mathbf{j}_{\mathbf{x}}^{\mathbf{f}} = \begin{bmatrix} j_{\mathbf{x}}^{\mathbf{f}}(1) \\ \vdots \\ j_{\mathbf{x}}^{\mathbf{f}}(p_x^f) \end{bmatrix}$$

correspond to the row and column locations, respectively. In order to ensure uniqueness of the row/column pairs $\left(i^{\mathbf{f}}_{\mathbf{x}}(k), j^{\mathbf{f}}_{\mathbf{x}}(k)\right)$ (where $i^{\mathbf{f}}_{\mathbf{x}}(k)$ and $j^{\mathbf{f}}_{\mathbf{x}}(k)$ refer to the k^{th} elements of the vectors $\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}$ and $\mathbf{j}^{\mathbf{f}}_{\mathbf{x}}$, respectively, $k = 1, \ldots, p^f_x$) the following column-major restriction is placed upon the order of the index vectors:

$$i^{\mathbf{f}}_{\mathbf{x}}(1) + n_x\left(j^{\mathbf{f}}_{\mathbf{x}}(1) - 1\right) < i^{\mathbf{f}}_{\mathbf{x}}(2) + n_x\left(j^{\mathbf{f}}_{\mathbf{x}}(2) - 1\right) < \cdots < i^{\mathbf{f}}_{\mathbf{x}}(p^f_x) + n_x\left(j^{\mathbf{f}}_{\mathbf{x}}(p^f_x) - 1\right). \quad (3)$$

Henceforth it shall be assumed that this restriction is always satisfied for row/column index vector pairs of the form of $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}})$, however it may not be explicitly stated. To refer to the possible non-zero *elements* of $\partial\mathbf{f}/\partial\mathbf{x}$, the vector $\mathbf{d}^{\mathbf{f}}_{\mathbf{x}} \in \mathbb{R}^{p^f_x}$ is used such that

$$d^{\mathbf{f}}_{\mathbf{x}}(k) = \frac{\partial f_{\left[i^{\mathbf{f}}_{\mathbf{x}}(k)\right]}}{\partial x_{\left[j^{\mathbf{f}}_{\mathbf{x}}(k)\right]}}, \quad (k = 1, \ldots, p^f_x), \quad (4)$$

where $d^{\mathbf{f}}_{\mathbf{x}}(k)$ refers to the k^{th} element of the vector $\mathbf{d}^{\mathbf{f}}_{\mathbf{x}}$. Using this sparse notation, the Jacobian $\partial\mathbf{f}/\partial\mathbf{x}$ may be fully defined given the row/column/value triplet $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{d}^{\mathbf{f}}_{\mathbf{x}}) \in \mathbb{Z}^{p^f_x}_+ \times \mathbb{Z}^{p^f_x}_+ \times \mathbb{R}^{p^f_x}$ together with the dimensions m_f and n_x. Moreover, the three-dimensional derivative matrix $\partial\mathbf{F}(\mathbf{x})/\partial\mathbf{x}$ is uniquely defined given the triplet $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{d}^{\mathbf{f}}_{\mathbf{x}})$ together with the dimensions q_f, r_f, and n_x.

3. OVERVIEW OF THE ADIGATOR ALGORITHM

Without loss of generality, consider a function $\mathbf{f}(\mathbf{v}(\mathbf{x}))$, where $\mathbf{f} : \mathbb{R}^{m_v} \to \mathbb{R}^{m_f}$ and $\partial\mathbf{v}/\partial\mathbf{x}$ is defined by the triplet $(\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{d}}_{\mathbf{x}}) \in \mathbb{Z}^{p^v_x}_+ \times \mathbb{Z}^{p^v_x}_+ \times \mathbb{R}^{p^v_x}$. Assume now that $\mathbf{f}(\cdot)$ has been coded as a MATLAB function, `F`, where the function `F` takes $\mathbf{v} \in \mathbb{R}^{m_v}$ as its input and returns $\mathbf{f} \in \mathbb{R}^{m_f}$ as its output. Given the MATLAB function `F`, together with the index vector pair $(\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}})$ and the dimensions m_v and n_x, the ADiGator algorithm determines the index vector pair $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}})$ and the dimension m_f. Moreover, a MATLAB derivative function, `F'`, is generated such that `F'` takes $\mathbf{v}$ and $\mathbf{d}^{\mathbf{v}}_{\mathbf{x}}$ as its inputs, and returns $\mathbf{f}$ and $\mathbf{d}^{\mathbf{f}}_{\mathbf{x}}$ as its outputs. In order to do so, the algorithm uses a process which we have termed source transformation via operator overloading. For a more detailed description of the method, the reader is referred to [Weinstein and Rao 2015] and [Patterson et al. 2013]. An overview of this process is now given in order to both grant the user a better understanding of how to efficiently utilize the ADiGator tool as well as to identify various assumptions and limitations of the algorithm.

At its core, the ADiGator algorithm utilizes operator overloading to propagate derivative non-zero locations while simultaneously printing the procedures required to compute the corresponding non-zero derivatives. In order to deal with cases where the function `F` contains flow control (loops, conditional statements, etc.), however, a higher-level approach is required. To elaborate, one cannot simply evaluate a function `F` on overloaded objects and gather information pertaining to any flow control present in `F`. In order to allow for flow control, user-defined programs are first transformed into intermediate function programs, where the intermediate source code is an augmented version of the original source code which contains calls to ADiGator transformation routines [Weinstein and Rao 2015]. The forward mode of AD is then affected by performing three overloaded passes on the intermediate program. On the first overloaded pass, a record of all operations, variables, and flow control statements is built. On the second overloaded pass, derivative sparsity patterns are propagated, and overloaded unions are performed where code branches join.[3] On the third and final overloaded pass, derivative sparsity patterns are again propagated forward, while the procedures

[3]This second overloaded pass is only required if there exists flow control in the user-defined program.

required to compute the output non-zero derivatives are printed to the derivative program. During this third overloaded pass, a great deal of effort is taken to make the printed procedures as efficient as possible by utilizing the known derivative sparsity patterns at each link in the chain rule.

3.1. User Source to Intermediate Source Transformations

The first step in the ADiGator algorithm is to transform the user-defined source code into an intermediate source code. This process is applied to the user provided main function, as well as any user-defined external functions (or sub-functions) which it calls. For each function contained within the user-defined program, a corresponding intermediate function, `adigatortempfunc#`, is created such that `#` is a unique integer identifying the function. The initial transformation process is carried out by reading the user-defined function line-by-line and searching for keywords. The algorithm looks for the following code behaviors and routines:

- *Variable assignments*. All variable assignments are determined by searching for the '=' character. Each variable assignment (as well as the calculations on the right-hand-side of the equal sign) are copied exactly from the user function to the intermediate function. Moreover, each variable assignment copied to the intermediate program is followed by a call to the ADiGator variable analyzer routine.
- *Flow control*. The algorithm only allows for `if/elseif/else`, `for`, and `while` statements. These statements (and corresponding `end` statements) are found by searching for their respective keywords and replaced with various transformations which allow the ADiGator algorithm to control the flow of the intermediate functions. Additionally, within `for` and `while` loops, `break` and `continue` statements are identified.
- *External function calls*. Prior to the user source to intermediate source transformation, it is determined of which functions the user-defined program is composed. Calls to these functions are searched for within the user-defined source code and replaced with calls to the corresponding `adigatortempfunc` function. User sub-functions are treated in the same manner.
- *Global variables.* Global variables are allowed to be used with the ADiGator algorithm *only* as a means of passing auxiliary data and are identified by the `global` statement.
- *Comments*. Any lines beginning with the '%' character are identified as comments and copied as inputs to the `adigatorVarAnalyzer` routine in the intermediate function. These comments will then be copied over to the generated derivative file.
- *Error statements*. Error statements are identified and replaced by calls to the `adigatorError` routine in the intermediate function. The error statements are then copied verbatim to the generated derivative file.

If the user-defined source code contains any statements that are not listed above (with the exception of operations defined in the overloaded library), then the transformation will produce an error stating that the algorithm cannot process the statement.

3.2. Overloaded Operations

Once the user-defined program has been transformed to the intermediate program, the forward mode of AD is affected by performing multiple overloaded passes on the intermediate program. In the presence of flow control, three overloaded passes (parsing, overmapping, and printing) are required, otherwise only two (parsing and printing) are required. In each overloaded pass, all overloaded objects are tracked by assigning each object a unique integer `id` value. In the parsing evaluation, information similar to conventional data flow graphs and control flow graphs is obtained by propagating overloaded objects with unique `id` fields. In the overmapping evaluation, forward

mode AD is used to propagate derivative sparsity patterns, and overloaded unions are performed in areas where flow control branches join. In the printing evaluation, each basic block of function code is evaluated on its set of overmapped input objects. In this final overloaded pass, the overloaded operations perform two tasks: propagating derivative sparsity patterns and printing the procedures required to compute the non-zero derivatives at each link in the forward chain rule. In this section we briefly introduce the overloaded `cada` class, the manner in which it is used to exploit sparsity at compile-time, a specific type of known numeric objects, and the manner in which the overloaded class handles logical references/assignments.

3.2.1. The Overloaded `cada` Class. The overloaded class is introduced by first considering a variable $\mathbf{Y}(\mathbf{x}) \in \mathbb{R}^{q_y \times r_y}$, where $\mathbf{Y}(\mathbf{x})$ is assigned to the identifier 'Y' in the user's code. It is then assumed that there exist some elements of $\mathbf{Y}(\mathbf{x})$ which are identically zero for any $\mathbf{x} \in \mathbb{R}^{n_x}$. These elements are identified by the strictly increasing index vector $\bar{\mathbf{i}}^{\mathbf{y}} \in \mathbb{Z}_+^{\bar{p}^f}$, where

$$\mathbf{y}_{[\bar{i}^{\mathbf{y}}(k)]} = 0, \quad \forall \mathbf{x} \in \mathbb{R}^{n_x} \quad (k = 1, \ldots, \bar{p}^f), \tag{5}$$

and $\mathbf{y}(\mathbf{x})$ is the unrolled column-major vector representation of $\mathbf{Y}(\mathbf{x})$. It is then assumed that the possible non-zero elements of the unrolled Jacobian, $\partial \mathbf{y}/\partial \mathbf{x} \in \mathbb{R}^{m_y \times n_x}$ ($m_y = q_y r_y$), are defined by the row/column/value triplet $(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{d}_{\mathbf{x}}^{\mathbf{y}}) \in \mathbb{Z}_+^{p_x^y} \times \mathbb{Z}_+^{p_x^y} \times \mathbb{R}^{p_x^y}$. The corresponding overloaded object, denoted $\mathcal{Y}$, would then have the following function and derivative properties:

Function		Derivative	
name:	'Y.f'	name:	'Y.dx'
size:	(q_y, r_y)	nzlocs:	$(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}})$
zerolocs:	$\bar{\mathbf{i}}^{\mathbf{y}}$		

Assuming that the object $\mathcal{Y}$ is instantiated during the printing pass, the procedures will have been printed to the derivative file such that, upon evaluation of the derivative file, `Y.f` and `Y.dx` will be assigned the values of $\mathbf{Y}$ and $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$, respectively. It is important to stress that the values of (q_y, r_y), $\bar{\mathbf{i}}^{\mathbf{y}}$, and $(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}})$ are all assumed to be fixed at the time of derivative file generation. Moreover, by adhering to the assumption that these values are fixed, it is the case that all overloaded operations must result in objects with fixed sizes and fixed derivative sparsity patterns (with the single exception to this rule given in Section 3.2.4). It is also noted that all user objects are assumed to be scalars, vectors, or matrices. Thus, while MATLAB allows for one to use n-dimensional arrays, the ADiGator algorithm may only be used with two dimensional arrays.

3.2.2. Exploiting Sparsity at the Operation Level. Holding to the assumption that all input sizes and sparsity patterns are fixed, any files that are generated by the algorithm are only valid for a single input size and derivative sparsity pattern. Fixing this information allows the algorithm to accurately propagate derivative sparsity patterns during the generation of derivative files. Moreover, rather than relying on compression techniques to exploit sparsity of the program as a whole, sparsity is exploited at *every* link in the forward chain rule. Typically this is achieved by only applying the chain rule to vectors of non-zero derivatives (for example, $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$). To illustrate this point, we consider the simple function line:

```
W = sin(Y);.
```

The chain rule for the corresponding operation $\mathbf{W}(\mathbf{x}) = \sin(\mathbf{Y}(\mathbf{x}))$ is then given by

$$\frac{\partial \mathbf{w}}{\partial \mathbf{x}} = \begin{bmatrix} \cos(y_1) & 0 & \cdots & 0 \\ 0 & \cos(y_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \cos(y_{m_y}) \end{bmatrix} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \tag{6}$$

where $\mathbf{w} \in \mathbb{R}^{m_y}$ is the unrolled column-major vector representation of $\mathbf{W}$. Given $(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}}) \in \mathbb{Z}_+^{p_x^y} \times \mathbb{Z}_+^{p_x^y}$, Eq. (6) may sparsely be carried out by the procedure

$$d_{\mathbf{x}}^{\mathbf{w}}(k) = \cos(y_{[i_{\mathbf{x}}^{\mathbf{y}}(k)]}) d_{\mathbf{x}}^{\mathbf{y}}(k), \quad k = 1, \ldots, p_x^y. \tag{7}$$

Moreover, the index vector pair which identifies the possible non-zero locations of $\partial \mathbf{w}/\partial \mathbf{x}$ is identical to that of $\partial \mathbf{y}/\partial \mathbf{x}$. During the printing evaluation, the overloaded `sin` routine would have access to $(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}})$ and print the procedures of Eq. (7) to the derivative file as the MATLAB procedure

```
W.dx = cos(Y(Index1)).*Y.dx;,
```

where the variable `Index1` would be assigned the value of the index vector $\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}$ and written to memory at the time the derivative procedure is printed. Thus, sparsity is exploited at compile-time, such that the chain rule is carried out at run-time by only operating on vectors of non-zero derivatives. Similar derivative procedures are printed for all array operations (for instance `sqrt`, `log`, `+`, `.*`).

The case where the chain rule is not simply applied to vectors of non-zero derivatives at run-time is that of matrix operations (for example, summation, matrix multiplication, etc.). In general, the inner derivative matrices of such operations contain rows with more than one non-zero value. Thus, the chain rule may not, in general, be carried out by performing element-wise array multiplications on vectors. Derivative sparsity, however, may still be exploited for such operations. For instance, consider the matrix operation $\mathbf{Z}(\mathbf{x}) = \mathbf{A}\mathbf{Y}(\mathbf{x})$, $\mathbf{A} \in \mathbb{R}^{q_z \times q_y}$, with associated chain rule

$$\frac{\partial \mathbf{Z}}{\partial x_k} = \mathbf{A} \frac{\partial \mathbf{Y}}{\partial x_k}, \quad (k = 1, \ldots, n_x). \tag{8}$$

Suppose now that

$$\mathbf{B} \equiv \begin{bmatrix} \frac{\partial \mathbf{Y}}{\partial x_1} & \cdots & \frac{\partial \mathbf{Y}}{\partial x_{n_x}} \end{bmatrix} \in \mathbb{R}^{q_y \times r_y n_x}. \tag{9}$$

Then

$$\mathbf{C} \equiv \mathbf{A}\mathbf{B} = \begin{bmatrix} \mathbf{A}\frac{\partial \mathbf{Y}}{\partial x_1} & \cdots & \mathbf{A}\frac{\partial \mathbf{Y}}{\partial x_{n_x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{Z}}{\partial x_1} & \cdots & \frac{\partial \mathbf{Z}}{\partial x_{n_x}} \end{bmatrix} \in \mathbb{R}^{q_z \times r_z n_x}, \tag{10}$$

where the matrices $\mathbf{B}$ and $\mathbf{C}$ have the same column-major linear indices as $\partial \mathbf{y}/\partial \mathbf{x}$ and $\partial \mathbf{z}/\partial \mathbf{x}$, respectively. Now consider that, given the index vector pair $(\mathbf{i}_{\mathbf{x}}^{\mathbf{y}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{y}})$, the sparsity pattern of $\mathbf{B}(\mathbf{x})$ is known. Moreover, if there exist any columns of $\mathbf{B}$ which are known to be zero, then the matrix multiplication of Eq. (10) performs redundant computations on columns whose entries are all zero. We now allow the strictly increasing index vector $\mathbf{k}_{\mathbf{x}}^{\mathbf{y}} \in \mathbb{Z}_+^{s_x^y}$, $s_x^y \leq r_y n_x$, to denote the columns of $\mathbf{B}$ which are *not* zero, and let

$$\mathbf{D} \equiv \begin{bmatrix} \mathbf{B}_{[k_{\mathbf{x}}^{\mathbf{y}}(1)]} & \cdots & \mathbf{B}_{[k_{\mathbf{x}}^{\mathbf{y}}(s_x^y)]} \end{bmatrix} \in \mathbb{R}^{q_y \times s_x^y} \tag{11}$$

be the collection of possibly non-zero columns of $\mathbf{B}$. All of the elements of $\mathbf{d}_{\mathbf{x}}^{\mathbf{z}}$ must then be contained within the matrix

$$\mathbf{E} \equiv \mathbf{A}\mathbf{D} = \begin{bmatrix} \mathbf{C}_{[k_{\mathbf{x}}^{\mathbf{y}}(1)]} & \cdots & \mathbf{C}_{[k_{\mathbf{x}}^{\mathbf{y}}(s_x^y)]} \end{bmatrix} \in \mathbb{R}^{q_z \times s_x^y}. \tag{12}$$

Thus, given a function line

```
Z = A*Y;,
```

the non-zero derivatives of $\partial \mathbf{z}/\partial \mathbf{x}$ would be computed via the MATLAB procedure

```
D = zeros(qy,syx);
D(Index2) = Y.dx;
E = A*D;
Z.dx = E(Index3);,
```

where `A`, `D`, `E`, `Y.dx`, `Z.dx`, `qy`, and `syx` correspond to $\mathbf{A}$, $\mathbf{D}$, $\mathbf{E}$, $\mathrm{d}_{\mathbf{x}}^{\mathbf{y}}$, $\mathrm{d}_{\mathbf{x}}^{\mathbf{z}}$, q_y, and s_x^y, respectively. Moreover, the variable `Index2` would be assigned the index vector which maps $\mathrm{d}_{\mathbf{x}}^{\mathbf{y}}$ into the proper elements of $\mathbf{D}$ and `Index3` would be assigned the index vector which maps $\mathrm{d}_{\mathbf{x}}^{\mathbf{z}}$ into the proper elements of $\mathbf{E}$. As with the previous example, the values of `Index2` and `Index3` would be written to memory at the time the derivative procedure is printed.

3.2.3. Known Numeric Objects. A common error that occurs when using operator overloading in MATLAB is given as

‘`Conversion to double from someclass not possible.`’.

This typically occurs when attempting to perform a subscript-index assignment such as `y(i) = x`, where `x` is overloaded and `y` is of the `double` class. In order to avoid this error and to properly track all variables in the intermediate program, the ADiGator algorithm ensures that *all* active variables in the intermediate program are overloaded. Moreover, immediately after a numeric variable (`double`, `logical` etc.) is created, it is transformed into a “known numeric object”, whose only relevant properties are its stored numeric value, string name and `id`. The numeric value is then assumed to be fixed. As a direct consequence, *all* operations performed in the intermediate program are forced to be overloaded. At times, this consequence may be adverse as redundant auxiliary computations may be printed to the derivative file. Moreover, in the worst case, one of the operations in question may not have an overloaded routine written, and thus produce an error.

3.2.4. Logical References and Assignments. As stated in Section 3.2.1, the algorithm only allows for operations which result in variables of a fixed size (given a fixed dimensional input). It is often the case, however, that one wishes to perform operations on only certain elements of a vector, where the element locations are determined by the values of the entries of the vector. For instance, one may wish to build the vector $\mathbf{y} \in \mathbb{R}^{n_x}$ such that

$$y_i = \begin{cases} x_i^2 & x_i < 0, \\ x_i & \text{otherwise}, \end{cases} \qquad i = 1, \ldots, n_x. \tag{13}$$

While one could use a conditional statement embedded within a loop to build $\mathbf{y}$, it is often more efficient to use logical array indexing to determine the locations of the negative elements of $\mathbf{x}$. Moreover, due to the fact that the value of $\mathbf{x}$ is not fixed at the time of the ADiGator call, the operation which references only the negative elements of $\mathbf{x}$ results in a variable of unknown dimension. In order to allow for such instances, the algorithm allows for unknown logical array references under the condition that, if a logical index reference is performed, the result of the logical reference must be assigned to a variable via a logical index assignment. Moreover, the same logical index variable must be used for both the reference and assignment. Thus, a valid way of

building y, as defined by Eq. (13), is given as:

```
negindex = x < 0;
xneg = x(negindex);
xnegsq = xneg.^2;
y = x;
y(negindex) = xnegsq;.
```

Here it is noted that the algorithm avoids the conflict of the unknown dimension (as a result of the logical reference) by not performing the logical reference until the time of the logical assignment. The code produced by applying the algorithm to the above code fragment is given as:

```
negindex.f = x.f < 0;
xneg.dx = x.dx;
xneg.f = x.f;
xnegsq.dx = 2*xneg.f.*xneg.dx
xnegsq.f = xneg.f.^2;
y.dx = x.dx;
y.f = x.f;
y.dx(negindex.f) = xnegsq.dx(negindex.f);
y.f(negindex.f) = xnegsq.f(negindex.f);,
```

where it is seen that the logical reference is effectively performed on the variable `xnegsq`, rather than `x`. This method of handling logical array references and assignments allows for all variables of the derivative program to be of a fixed dimension, yet can result in some unnecessary computation (which, for this example, includes the power operations on the non-negative elements).

3.3. Handling of Flow Control

The ADiGator algorithm handles flow control by performing overloaded unions where code fragments join. Namely, the unions are performed on the exit of conditional `if/elseif/else` statements, on the entrance of `for` loop statements, and on both the entrance and exit of user-defined external functions and `while` loops. The union of all possible objects that may be assigned to a variable is then referred to as an *overmapped object*. Overmapped objects have the following key properties:

- *Known numeric overmapped objects*. An overmapped object may only be a known numeric object if all possible variables result in the same numeric value.
- *Function size*. The row/column size of the overmapped object is considered to be 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 the same zero element.
- *Derivative sparsity*. The derivative is only considered to have a known zero element if every possible derivative has the same known zero element.

An in-depth analysis of the methods used to transform flow control is given in Weinstein and Rao [2015]. In this section, an overview is given in order to discuss the various implications of the methods.

3.3.1. `for` Loop Statements. The ADiGator algorithm is able to transform `for` loops from a function program to a derivative program under the stipulation that the loop is executed for a known number of iterations (that is, the loop index expression has a fixed second dimension). The loops are, however, allowed to contain `break` and `continue` statements. In order to transform such a loop, the loop is effectively unrolled

for the purpose of analysis in the overmapping evaluation phase. During this unrolling process, all possible iterations of the loop are evaluated, and unions are performed at the entrance to the loop to build a set of overmapped loop inputs. In the presence of `break`/`continue` statements, unions are also performed on all possible termination points of the loop. Additionally, data pertaining to any organizational operations (for example, `subsref`, `subsasgn`, `horzcat`, etc.) contained within the loop is collected for each independent loop iteration. The derivative loop is then printed by evaluating a single loop iteration on the set of overmapped loop inputs. The two primary implications of this process are given as follows:

(1) The time required to transform a function `for` loop to a derivative `for` loop is proportional to the number of possible loop iterations.
(2) The transformation of a function loop containing variables whose size and/or derivative sparsity patterns vary results in redundant operations being performed in the derivative loop.

Moreover, given a function loop containing variables whose dimensions are iteration dependent, it is often advisable to unroll the loop, assuming the loop does not contain `break`/`continue` statements.

3.3.2. `if/elseif/else` Statements. A conditional statement containing M branches effectively adds M possible branches to a program. Thus, if a program contains two successive conditional statements, each containing M possible branches, the program has M^2 possible branches. Rather than analyzing all possible branches of a program, the ADiGator algorithm instead analyzes each possible branch of a conditional statement, and then creates a set of overmapped outputs. The remainder of the program is then analyzed on the given overmapped outputs. The implications of this process are given as follows:

(1) If a known numeric object is an output of a conditional fragment whose numeric value is dependent upon which branch is taken, then that object may *not* be later used as
   • an array subscript index given to `subsref`, `subsasgn`, or `sparse`
   • the argument of an array instantiation operation (for example, `zeros`, `ones`, etc.)
(2) If the output variable of a conditional fragment changes dimension and/or derivative sparsity pattern depending upon which branch is taken, redundant "zero" calculations will be performed in the derivative program following the conditional statement.

It is advisable when using conditional statements to ensure that the sizes of the outputs do not vary depending upon which branch is taken.

3.3.3. `while` Statements. It is the case that any `while` loop with an iteration limit may be written in MATLAB as a `for` loop containing an `if` and `break` statement. It is often the case, however, that an iteration limit is not easily determined. Moreover, when transforming a `for` loop, the ADiGator algorithm will analyze *all* loop iterations. This may result in costly redundant analysis during the overmapping evaluations, particularly when performing fixed-point iterations. Thus, `while` loops are allowed to be used only for fixed-point iterations of the form of

$$\mathbf{y}^{(k)}(\mathbf{x}) = \mathbf{L}(\mathbf{y}^{(k-1)}(\mathbf{x})), \tag{14}$$

where $\mathbf{L} : \mathbb{R}^{m_y} \to \mathbb{R}^{m_y}$ represents the operations contained within allowable `while` loops and $\mathbf{y}^{(k)} \in \mathbb{R}^{m_y}$ denotes the collection of inputs to the k^{th} iteration of the loop. In order to transform the `while` loop, the ADiGator algorithm seeks to find a static loop

iteration, k, such that $\mathcal{Y}^{(k)} = \mathcal{Y}^{(k-1)}$. Allowing $\mathcal{Y}^{(0)}$ to be the overloaded input to the loop on the first iteration, a static loop iteration is found by iteratively computing

$$\mathcal{Y}^{(k)} = \mathbf{L}(\bigcup_{i=0}^{k-1} \mathcal{Y}^{(i)}) \tag{15}$$

until an iteration k is found such that $\bigcup_{i=0}^{k-1} \mathcal{Y}^{(i)} = \bigcup_{i=0}^{k} \mathcal{Y}^{(i)}$, or a maximum iteration limit is reached. Assuming a static iteration k is found, the loop will be transformed in the printing evaluation by evaluating $\mathbf{L}$ on the overmapped input, $\bigcup_{i=0}^{k} \mathcal{Y}^{(i)}$. The implications of this process are as follows:

(1) If the user code performs iteration-dependent organizational operations within a `while` loop (for example, `y(count)` where `count` is an iteration count variable), then an error will be produced during the overmapping evaluation phase. Such operations do not adhere to the fixed-point constraint of Eq. (14) and are found by evaluating the i^{th} loop iteration on the union of all previous inputs.
(2) The algorithm will attempt to find a static point until a maximum number of loop iterations is reached, where the user may define the maximum number of loop iterations as an ADiGator option. Any hard-coded loop iteration limits in the user-defined function will be transferred to the derivative program, but not used when attempting to find a static iteration.

3.3.4. Called Functions. Consider now the transformation of an external user-defined function `G` that is called numerous times from the main user-defined function `F`. For this discussion, $\bar{\texttt{F}}$ and $\bar{\texttt{G}}$ are used to denote the corresponding intermediate functions, and $\texttt{F}'$ and $\texttt{G}'$ are used to denote the corresponding transformed derivative functions. In order to transform `G` to $\texttt{G}'$, the ADiGator algorithm performs overloaded unions on the entrance and exit of $\bar{\texttt{G}}$ during the overmapping evaluation phase. At each call to $\bar{\texttt{G}}$ from $\bar{\texttt{F}}$ in the overmapping evaluation phase, the overloaded outputs are determined in one of two ways. In the event that the overloaded inputs are identical to those of a previous call, the stored outputs of the previous call are returned. Otherwise, the intermediate function $\bar{\texttt{G}}$ is evaluated on the current overloaded inputs. All flow control and overloaded operations contained within $\bar{\texttt{G}}$ are treated in the same manner as they would be if performed within $\bar{\texttt{F}}$, with the exception of organizational operations. In order to allow for call-dependent organizational operations, all organizational operations performed within $\bar{\texttt{G}}$ are treated in a manner similar to those performed within `for` loops. In the printing evaluation, each time $\bar{\texttt{G}}$ is called from within $\bar{\texttt{F}}$, the stored inputs and outputs are used in order to print a call to the function $\texttt{G}'$ and the proper overloaded outputs are returned. The function $\bar{\texttt{G}}$ is then evaluated on its overmapped inputs in order to create the function source code of $\texttt{G}'$. The implications of this process are as follows:

(1) All functions called within a user program must have a definitive input and output structure. Thus, nested functions may not be used, external functions must have the same number of input/output variables across all calls, and global variables may not be used to pass information between the functions of the user-defined program.
(2) If the input variables to an external function change size and/or derivative sparsity patterns, then the transformed called function will perform redundant computations at run-time. Moreover, the efficiency of both the transformation process and the evaluation of the generated derivative sub-function are dependent upon the variance of the input sizes and sparsity patterns.

(3) Functions may not call themselves from within their own methods (that is, recursion is not permitted).

4. OVERLOADED CELL AND STRUCTURE ARRAYS

In Section 3 it was assumed that all user variables in the originating program are of class `double`. In the intermediate program, all such objects are effectively replaced by objects of the `cada` class [Patterson et al. 2013], where each `cada` object is tracked by a unique `id` value. It is sometimes the case, however, that a user code is made to contain cell and/or structure arrays, where the elements of the arrays contain objects of the `double` class. In the intermediate program, it is then desirable to track the outermost cell and/or structure arrays, rather than each of the objects of which the array is composed. To this end, all cell and structure arrays are replaced with objects of the `cadastruct` class during the overloaded analysis. Each `cadastruct` object is then assigned a unique `id` value, assuming it does not correspond to a scalar structure. In the event that a scalar structure is built, then each of the fields of the scalar structure is treated as a unique variable. The `cadastruct` objects are themselves made to contain objects of the `cada` class, however, the embedded `cada` objects are not tracked (assuming the object does not correspond to a scalar structure). The handling of cell and structure arrays in this manner allows the algorithm to perform overloaded unions of cell and structure arrays and to print loop iteration-dependent cell/structure array references and assignments.

5. HIGHER-ORDER DERIVATIVES

An advantage of the ADiGator algorithm is that, by producing stand-alone derivative source code, the algorithm may be applied recursively to generate n^{th}-order derivative files. If the algorithm was blindly applied in a recursive manner, however, the resulting n^{th}-order code would contain redundant 1^{st} through $(n-1)^{th}$ derivative computations. To illustrate, consider the application of the algorithm to a function which simply computes `y = sin(x)`, and then again on the resulting derivative code. The transformation would performed as follows:

$$\texttt{y = sin(x)} \begin{cases} \texttt{y.dx = cos(x)} \begin{cases} \texttt{y.dx.dx = -sin(x)} \\ \texttt{y.dx.f = cos(x)} \end{cases} \\ \texttt{y = sin(x)} \quad \begin{cases} \texttt{y.dx = cos(x)} \\ \texttt{y.f = sin(x)} \end{cases} \end{cases}$$

Thus, at the second derivative level, the first derivative would be computed twice, once as a function variable and once as a derivative variable. In a classical source transformation approach, such redundant computations would be eliminated in a code optimization phase. The ADiGator algorithm, however, does not have a code optimization phase, but rather performs optimizations at the operation level. What is available, however, is the capability of the algorithm to recognize when it is performing source transformation on code which was previously generated by the algorithm itself. Moreover, the algorithm can recognize the naming scheme used in the previously generated file in order to eliminate any redundant 1^{st} through $(n-1)^{th}$ derivative computations in the n^{th} derivative file.

6. STORAGE OF INDICES USED IN GENERATED CODE

As may be witnessed in Section 3.2, the derivative procedures printed by overloaded operations can be highly dependent upon reference and assignment index vectors being printed to variables in the derivative file. Moreover, at the time of which the procedures that are dependent upon these index vectors are printed to the file, the values

of the index vectors are both known and fixed. Thus, rather than printing procedures which build the index vectors (for example, `i = [1 2 3...]`), the index vectors are written to variable names in a MATLAB binary file. The variables are then brought into global MATLAB memory to be accessed at run-time. By handling index vectors in this manner, they must only be loaded into memory a single time and may then be used to compute the derivative multiple times, thus statically tying sparsity exploitation to the derivative procedure.

7. VECTORIZATION OF THE CADA CLASS

In this section, the differentiation of a special class of vectorized functions is considered, where we define a vectorized function as any function of the form of $\mathbf{F} : \mathbb{R}^{n_x \times N} \to \mathbb{R}^{m_f \times N}$ which performs the vector valued function $\mathbf{f} : \mathbb{R}^{n_x} \to \mathbb{R}^{m_f}$ on each column of its input. That is,

$$\mathbf{F}(\mathbf{X}) = [\, \mathbf{f}(\mathbf{X}_1) \;\; \mathbf{f}(\mathbf{X}_2) \;\; \cdots \;\; \mathbf{f}(\mathbf{X}_N) \,] \in \mathbb{R}^{n \times N}, \tag{16}$$

where $\mathbf{X}_k \in \mathbb{R}^{n_x}$, $k = 1, \ldots, N$,

$$\mathbf{X} = [\, \mathbf{X}_1 \;\; \mathbf{X}_2 \;\; \cdots \;\; \mathbf{X}_N \,] \in \mathbb{R}^{n_x \times N}. \tag{17}$$

It is stressed that the vectorized functions of this section are not limited to a single operation, but rather may be coded as a sequence of operations. Similar to array operations, vectorized functions have a sparse block diagonal Jacobian structure due to the fact that

$$\frac{\partial F_{l,i}}{\partial X_{j,k}} = 0, \quad \forall i \neq k,\ l = 1, \ldots, m_f,\ j = 1, \ldots, n_x. \tag{18}$$

Allowing

$$\mathbf{X}^\dagger = \begin{bmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \vdots \\ \mathbf{X}_N \end{bmatrix} \in \mathbb{R}^{n_x N}, \quad \mathbf{F}^\dagger(\mathbf{X}) = \begin{bmatrix} \mathbf{f}(\mathbf{X}_1) \\ \mathbf{f}(\mathbf{X}_2) \\ \vdots \\ \mathbf{f}(\mathbf{X}_N) \end{bmatrix} \in \mathbb{R}^{m_f N}, \tag{19}$$

the two-dimensional Jacobian $\partial \mathbf{F}^\dagger / \partial \mathbf{X}^\dagger$ is given by the block diagonal matrix

$$\frac{\partial \mathbf{F}^\dagger}{\partial \mathbf{X}^\dagger} = \begin{bmatrix} \frac{\partial \mathbf{F}_1}{\partial \mathbf{X}_1} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \frac{\partial \mathbf{F}_2}{\partial \mathbf{X}_2} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \frac{\partial \mathbf{F}_N}{\partial \mathbf{X}_N} \end{bmatrix} \in \mathbb{R}^{m_f N \times n_x N}, \tag{20}$$

where

$$\frac{\partial \mathbf{F}_i}{\partial \mathbf{X}_i} = \begin{bmatrix} \frac{\partial F_{1,i}}{\partial X_{1,i}} & \frac{\partial F_{1,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{1,i}}{\partial X_{n_x,i}} \\ \frac{\partial F_{2,i}}{\partial X_{1,i}} & \frac{\partial F_{2,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{2,i}}{\partial X_{n_x,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_{m_f,i}}{\partial X_{1,i}} & \frac{\partial F_{m_f,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{m_f,i}}{\partial X_{n_x,i}} \end{bmatrix} \in \mathbb{R}^{m_f \times n_x}, \quad i = 1, \ldots, N. \tag{21}$$

Such functions commonly occur when utilizing collocation methods [Ascher et al. 1995] to obtain numerical solutions of ordinary differential equations, partial differential equations, or integral equations. In such cases, it is the goal to obtain the values of

$\mathbf{X} \in \mathbb{R}^{n_x \times N}$ which solve the equation

$$\mathbf{c}(\mathbf{F}(\mathbf{X}), \mathbf{X}) = \mathbf{0} \in \mathbb{R}^{m_c}, \tag{22}$$

where $\mathbf{F}(\mathbf{X})$ is of the form of Eq. (16). Now, one could apply AD directly to Eq. (22), however, it is often the case that it is more efficient to instead apply AD separately to the function $\mathbf{F}(\mathbf{X})$, where the specific structure of Eq. (20) may be exploited. The results may then be used to compute the derivatives of Eq. (22).

Due to the block diagonal structure of Eq. (20), it is the case that the vectorized problem has an inherently compressible Jacobian with a maximum column dimension of n_x. This compression may be performed via the pre-defined Curtis-Powell-Reid seed matrix

$$\mathbf{S} = \begin{bmatrix} \mathbf{I}_{n_x} \\ \mathbf{I}_{n_x} \\ \vdots \\ \mathbf{I}_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x N \times n_x}, \tag{23}$$

where $\mathbf{I}_{n_x}$ is the $n_x \times n_x$ identity matrix. The ADiGator algorithm in the vectorized mode does not, however, rely upon matrix compression, but rather utilizes the fact that the structure of the Jacobian of Eq. (20) is determined by the structure of the Jacobian of Eq. (21). To exhibit this point, the row/column pairs of the derivative of f with respect to its input are now denoted by $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f}$. The N derivative matrices, $\partial \mathbf{F}_i / \partial \mathbf{X}_i$, may then be represented by the row/column/value triplets $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{d}^{\mathbf{F}_i}_{\mathbf{X}_i}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f} \times \mathbb{R}^{p_x^f}$ together with the dimensions m_f and n_x. All possible non-zero derivatives of $\partial \mathbf{F} / \partial \mathbf{X}$ are then given by

$$\mathbf{D}^{\mathbf{F}}_{\mathbf{X}} = \left[\begin{array}{cccc} \mathbf{d}^{\mathbf{F}_1}_{\mathbf{X}_1} & \mathbf{d}^{\mathbf{F}_2}_{\mathbf{X}_2} & \cdots & \mathbf{d}^{\mathbf{F}_N}_{\mathbf{X}_N} \end{array} \right] \in \mathbb{R}^{p_x^f \times N}. \tag{24}$$

Furthermore, $\partial \mathbf{F} / \partial \mathbf{X}$ may be fully defined given the vectorized row/column/value triplets $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{D}^{\mathbf{F}}_{\mathbf{X}}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f} \times \mathbb{R}^{p_x^f \times N}$, together with the dimensions n_x, n_f, and N. Thus, in order to print derivative procedures of a vectorized function as defined in Eq. (16), it is only required to propagate row/column index vector pairs $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f}$ corresponding to the non-vectorized problem, and to print procedures that compute vectorized non-zero derivatives, $\mathbf{D}^{\mathbf{F}}_{\mathbf{X}} \in \mathbb{R}^{p_x^f \times N}$.

In order to identify vectorized `cada` objects, all vectorized `cada` instances are made to have a value of `Inf` located in the size field corresponding to the vectorized dimension. Then, at each vectorized `cada` operation, sparsity patterns of the non-vectorized problem are propagated (that is, $(\mathbf{i}^{\mathbf{f}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{f}}_{\mathbf{x}})$) and procedures are printed to the derivative file to compute the vectorized function and vectorized derivative values (that is, $\mathbf{F}$ and $\mathbf{D}^{\mathbf{F}}_{\mathbf{X}}$). It is then the case that any operations performed on a vectorized `cada` object must be of the form given in Eq. (16).

Here is is noted that, given a fixed value of N, the non-vectorized mode may easily be used to print the procedures required to compute the non-zero derivatives of $\mathbf{F}(\mathbf{X})$. Typically the derivative files generated by the vectorized and non-vectorized modes will perform the exact same floating point operations at run-time. One may then question the advantages of utilizing the vectorized mode, particularly when more work is required of the user in order to separate vectorized functions. The advantages of the vectorized mode are given as follows:

(1) *Derivative files are vectorized.* Typically functions of the form of Eq. (16) are coded such that the value of N may be any positive integer. By utilizing the vectorized

mode, it is the case that the derivative files are generated such that N may be any positive integer. In contrast, any files generated using the non-vectorized mode are only valid for fixed input sizes. Allowing the dimension N to change is particularly helpful when using collocation methods together with a process known as mesh refinement [Betts 2009] because in such instances the problem of Eq. (22) must often be re-solved for different values of N.

(2) *Compile time is reduced.* By taking advantage of the fact that the sparsity of the vectorized problem (that is, $\mathbf{F}(\mathbf{X})$) is determined entirely by the sparsity of the non-vectorized problem (that is, $\mathrm{f}(\mathrm{x})$), it is the case that sparsity propagation costs are greatly reduced when using the vectorized mode over the non-vectorized mode.

(3) *Run-time overhead is reduced.* In order to exploit sparsity, the algorithm prints derivative procedures which perform many subscript index references and assignments at run-time. Unfortunately, these reference and assignment operations incur run-time penalties proportional to the length of the reference/assignment index vectors [Menon and Pingali 1999]. Moreover, the lengths of the used reference and assignment indices are proportional to the number of non-zero derivatives at each link in the chain rule. When printing derivative procedures in the vectorized mode, however, the ':' character is used as a reference to *all* elements in the vectorized dimension. Thus, the lengths of the required index vectors are proportional to the number of non-zero derivatives of the non-vectorized problem (that is, $\partial \mathrm{f}/\partial \mathrm{x}$), rather than the vectorized problem (that is, $\partial \mathbf{F}/\partial \mathbf{X}$). Indexing reference/assignment run-time overheads are therefore reduced by an order of N when using the vectorized mode rather than the non-vectorized.

8. USER INTERFACE TO ADIGATOR

The computation of derivatives using the ADiGator package is carried out in a multi-step process. First, the user must code their function as a MATLAB program which conforms to the restrictions discussed in Section 3. The user must then fix information pertaining to the inputs of the program (that is, input variable sizes and derivative sparsity patterns). The ADiGator algorithm is then called to transform the user-defined function program into a derivative program, where the derivative program is only valid for the fixed input information. The ADiGator tool is then no longer used and the generated derivative program may be evaluated on objects of the `double` class to compute the desired derivatives.

In order to begin the transformation process, the ADiGator algorithm must create overloaded objects of the form discussed in Section 3.2.1. Thus, the user must provide certain information for each input to their program. Assuming temporarily that all user inputs to the original function program are of the `double` class, then all user inputs must fall into one of three categories:

- *Derivative inputs.* Derivative inputs are any inputs which are a function of the variable of differentiation. Derivative inputs must have a fixed size and fixed derivative sparsity pattern.
- *Known numeric inputs.* Known numeric inputs are any inputs whose values are fixed and known. These inputs will be transformed into the known numeric objects discussed in Section 3.2.3.
- *Unknown auxiliary inputs.* Unknown auxiliary inputs are any inputs which are not a function of the variable of differentiation nor are they of a fixed value. It is required, however, that unknown auxiliary inputs have a fixed size.

For each of the user-defined input variables, the user must identify to which category the input belongs and create an ADiGator input variable. Under the condition that a user-defined program takes a structure or cell as an input, the corresponding ADiGator

input variable is made to be a structure or cell where each cell/structure element corresponding to an object of the `double` class must be identified as one of the three different input types. The ADiGator input variables are thus made to contain all fixed input information and are passed to the ADiGator transformation algorithm. The ADiGator transformation algorithm is then carried out using the `adigator` command which requires the created ADiGator input variables, the name of the main function file of the user-defined program, and the name of which the generated derivative file is to be titled. The generated derivative program then takes as its inputs the function values of derivative inputs, known numeric inputs, and unknown auxiliary inputs, together with the values of the non-zero derivatives of the derivative inputs. Moreover, the generated derivative program returns, for each output variable, the function values, possible non-zero derivative values, and locations of the possible non-zero derivatives. The user interface thus allows a great deal of flexibility for the user-defined function program input/output scheme. Moreover, the user is granted the ability to use any desired input seed matrices.

9. EXAMPLES

In this section, the ADiGator tool is tested by solving four different classes of problems. In Section 9.1, the developed algorithm is used to integrate an ordinary differential equation with a large sparse Jacobian. In Section 9.2, a set of three fixed dimension non-linear system of equations problems are investigated, and in Section 9.3, a large sparse unconstrained minimization problem is presented. Lastly, in Section 9.4 the vectorized mode of ADiGator is showcased by solving the large scale nonlinear programming problem that arises from the discretization of an optimal control problem. For each of the tested problems, comparisons are drawn against methods of finite-differencing, the well-known MATLAB AD tools ADiMat version 0.6.0, INTLAB version 6, and MAD version 1.4, and, when available, hand-coded derivative files. 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.

9.1. Stiff Ordinary Differential Equation

In this section the well-known Burgers' equation is solved using a moving mesh technique as presented in Huang et al. [1994]. The form of Burgers' equation used for this example is given by

$$\dot{u} = \alpha \frac{\partial^2 u}{\partial y^2} - \frac{\partial}{\partial y}\left(\frac{u^2}{2}\right), \quad 0 < y < 1,\ t > 0,\ \alpha = 10^{-4} \tag{25}$$

with boundary conditions and initial conditions

$$\begin{array}{rcll} u(0,t) & = & u(1,t) = 0, & t > 0, \\ u(x,0) & = & \sin(2\pi y) + \frac{1}{2}\sin(\pi y), & 0 \leq 1. \end{array} \tag{26}$$

The partial differential equation (PDE) of Eq. (25) is then transformed into an ordinary differential equation (ODE) via a central difference discretization together with the moving mesh PDE, MMPDE6 (with $\tau = 10^{-3}$), and spatial smoothing is performed with parameters $\gamma = 2$ and $p = 2$. The result of the discretization is then a stiff ODE of the form

$$\mathbf{M}(t,x)\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}), \tag{27}$$

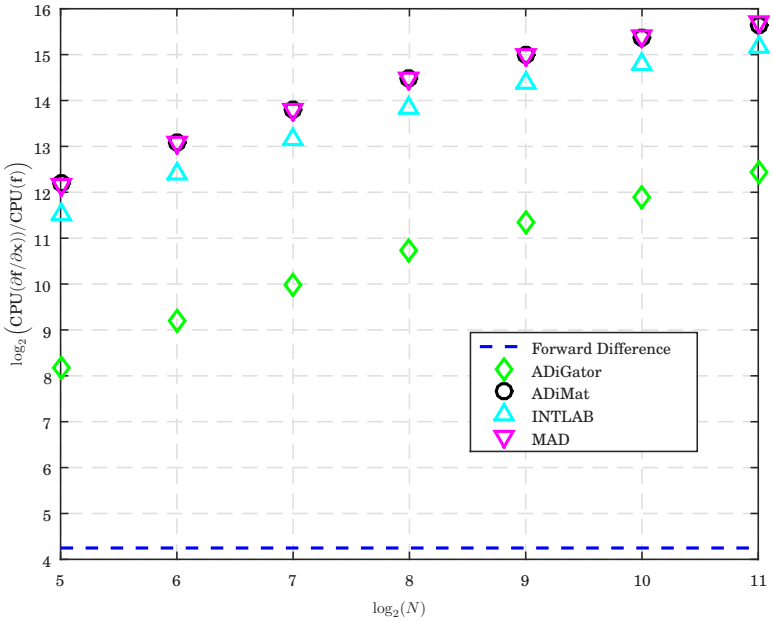
where $\mathbf{M} : \mathbb{R} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_x \times n_x}$ is a mass-matrix function and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$ is the ODE function. This problem is given as an example problem for the MATLAB ODE

suite and is solved with the stiff ODE solver, `ode15s` [Shampine and Reichelt 1997], which allows the user to supply the Jacobian $\partial \mathbf{f}/\partial \mathbf{x}$.

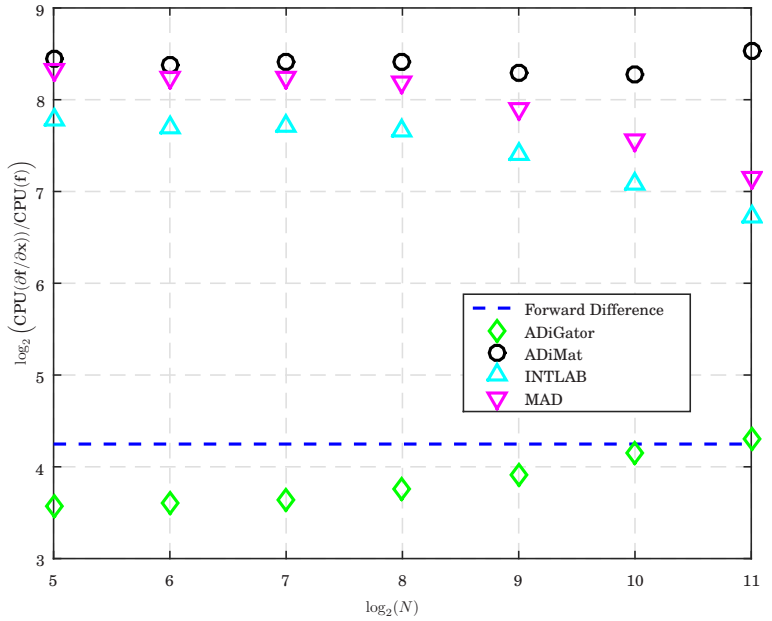
Prior to actually solving the ODE, a study is performed on the efficiency of differentiation of the function $\mathbf{f}(t, \mathbf{x})$ for varying values of n_x, where the code for the function $\mathbf{f}(t, \mathbf{x})$ has been taken verbatim from the MATLAB example file `burgersode`. The Jacobian $\partial \mathbf{f}/\partial \mathbf{x}$ is inherently sparse and compressible, where a Curtis-Powell-Reid seed matrix $\mathbf{S} \in \mathbb{Z}_+^{n_x \times 18}$ may be found for all $n_x \geq 18$. Thus, the Jacobian $\partial \mathbf{f}/\partial \mathbf{x}$ becomes increasingly more sparse as the dimension of n_x is increased. A test was first performed by applying the AD tools ADiGator, ADiMat, INTLAB, and MAD to the function code for $\mathbf{f}(t, \mathbf{x})$ taken verbatim from the MATLAB file `burgersode`. It was found, however, that all tested AD tools perform quite poorly, particularly when compared to the theoretical efficiency of a sparse finite-difference. The reason for the poor performance is due to the fact that the code used to compute $\mathbf{f}$ contains four different explicit loops, each of which runs for $\frac{n_x}{2} - 2$ iterations and performs scalar operations. When dealing with the explicit loops, all tested AD tools incur a great deal of run-time overhead penalties. In order to quantify these run-time overheads, the function file which computes $\mathbf{f}$ was modified such that all loops (and scalar operations within the loops) were replaced by the proper corresponding array operations and vector reference/assignment index operations.[4] A test was then performed by applying AD to the resulting modified file. The results obtained by applying AD to both the original and modified files are given in Fig. 1. Results were obtained using ADiGator in the default mode, ADiMat in the scalar compressed forward mode, INTLAB's `gradient` class, and MAD in the compressed forward mode. Within this figure it is seen that all tested AD tools greatly benefit from the removal of the loop statements. Moreover, it is seen that the ADiGator tool performs relatively well compared to that of a theoretical finite difference. To further investigate the handling of explicit loops, absolute function CPU times and ADiGator file generation times are given in Table I. Within this table, it is seen that the reason the original Burgers' ODE function file is written with loops is that it is slightly more efficient than when the loops are removed. It is, however, also seen that when using the ADiGator tool to generate derivative files, the cost of the transformation of the original code containing loops increases immensely as the value of n_x increases. This increase in cost is due to the fact that the ADiGator tool effectively unrolls loops for the purpose of analysis, and thus must perform a number of overloaded operations proportional to the value of n_x. When applying the ADiGator tool to the file containing no explicit loops, however, the number of required overloaded operations stays constant for all values of n_x. From this analysis, it is clear that explicit loops should largely be avoided whenever using any of the tested AD tools. Moreover, it is clear that the efficiency of applying AD to a MATLAB function is not necessarily proportional to the efficiency of the original function.

The efficiency of the ADiGator tool is now investigated by solving the ODE and comparing solution times obtained by supplying the Jacobian via the ADiGator tool versus supplying the Jacobian sparsity pattern and allowing `ode15s` to use the `numjac` finite-difference tool to compute the required derivatives. It is important to note that the `numjac` finite-difference tool was specifically designed for use with the MATLAB ODE suite, where a key component of the algorithm is to choose perturbation step-sizes at one point based off of data collected from previous time steps [Shampine and Reichelt 1997]. Moreover, it is known that the algorithm of `ode15s` is not extremely reliant upon

[4]This process of replacing loops with array operations is often referred to as "vectorization". In this paper the term "vectorized" has already been used to refer to a specific class of functions in Section 7. Thus, in order to avoid any confusion, use of the term "vectorization" is avoided when referring to functions whose loops have been replaced.



(a) With Explicit Loops



(b) Without Explicit Loops

Fig. 1: Burgers' ODE Jacobian to function CPU ratios. (a) Ratios obtained by differentiating the original implementation of f containing explicit loops. (b) Ratios obtained by differentiating the modified implementation of f containing no explicit loops.

Table I: Burgers' ODE function CPU and ADiGator generation CPU times.

n_x:	32	64	128	256	512	1024	2048
Function File Computation Time (ms)							
with loops:	0.2046	0.2120	0.2255	0.2449	0.2895	0.3890	0.5615
without loops:	0.2122	0.2190	0.2337	0.2524	0.2973	0.3967	0.5736
ADiGator Derivative File Generation Time (s)							
with loops:	2.410	4.205	7.846	15.173	30.130	62.754	137.557
without loops:	0.682	0.647	0.658	0.666	0.670	0.724	0.834

precise Jacobian computations, and thus the `numjac` algorithm is not required to compute extremely accurate Jacobian approximations [Shampine and Reichelt 1997]. For these reasons, it is expected that when using `numjac` in conjunction with `ode15s`, Jacobian to function CPU ratios should be near the theoretical values shown in Fig. 1. In order to present the best case scenarios, tests were performed by supplying `ode15s` with the more efficient function file containing loop statements. When the ADiGator tool was used, Jacobians were supplied by the files generated by differentiating the function whose loops had been removed. In both cases, the ODE solver was supplied with the mass matrix, the mass matrix derivative sparsity pattern, and the Jacobian sparsity pattern. Moreover, absolute and relative tolerances were set equal to 10^{-5} and 10^{-4}, respectively, and the ODE was integrated on the interval $t = [0,\ 2]$. Test results may be seen in Table II, where it is seen that the ODE may be solved more efficiently when using `numjac` for all test cases except $n_x = 2048$. It is also seen that the number of Jacobian evaluations required when using either finite-differences or AD are roughly equivalent. Thus, the `ode15s` algorithm, in this case, is largely unaffected by supplying a more accurate Jacobian.

Table II: Burgers' ODE solution times.

n_x:	32	64	128	256	512	1024	2048
ODE Solve Time (s)							
ADiGator:	1.471	1.392	2.112	4.061	10.472	36.386	139.813
`numjac`:	1.383	1.284	1.958	3.838	9.705	32.847	140.129
Number of Jacobian Evaluations							
ADiGator:	98	92	126	197	305	495	774
`numjac`:	92	92	128	194	306	497	743

9.2. Fixed Dimension Nonlinear Systems of Equations

In this section, analysis is performed on a set of fixed dimension nonlinear system of equations problems taken from the MINPACK-2 problem set [Averick et al. 1991]. While originally coded in Fortran, the implementations used for the tests of this section were obtained from Lenton [2005]. The specific problems chosen for analysis are those of the "combustion of propane fuel" (CPF), "human heart dipole" (HHD), and "coating thickness standardization" (CTS). The CPF and HDD problems represent systems of nonlinear equations $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ ($n = 8$ and $n = 11$, respectively) where it is desired to find $\mathbf{x}^*$ such that $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$. The CTS problem represents a system of nonlinear equations $\mathbf{f} : \mathbb{R}^{134} \to \mathbb{R}^{252}$ where it is desired to find $\mathbf{x}^*$ which minimizes $\mathbf{f}(\mathbf{x})$ in the least-squared sense. The standard methods used to solve such problems are based upon Newton iterations and thus require iterative Jacobian computations.

Prior to solving the nonlinear problems, a test is first performed to gauge the efficiency of the Jacobian computation compared to the other well-known MATLAB AD tools. The implementation of Lenton [2005] provides hand-coded Jacobian files which also provide a convenient base-line for computation efficiency. For each of the problems the ADiGator tool was tested against the ADiMat tool in the scalar compressed forward mode, the INTLAB tool's `gradient` class, the MAD tool in the compressed forward mode, and the hand-coded Jacobian as provided by Lenton [2005]. Moreover, it is noted that the Jacobians of the CPF and HHD functions are incompressible while the Jacobian of the CTS function is compressible with a column dimension of six. Thus, for the CPF and HHD tests, the ADiMat and MAD tools are essentially used in the full modes. The resulting Jacobian to function CPU ratios are given in Table III together with the theoretical ratio for a sparse finite difference (sfd). From Table III it is seen that the ADiGator algorithm performs relatively better on the sparser CPF and HHD functions (whose Jacobians contain 43.8% and 2.61% non-zero entries, respectively) than on the denser HHD problem (whose Jacobian contains 81.25% non-zero entries). Moreover, it is seen that, on the incompressible CPF problem, the ADiGator algorithm performs more efficiently than a theoretical sparse finite difference. Furthermore, in the case of the compressible CTS problem, the ADiGator tool performs more efficiently than the hand-coded Jacobian file.

Table III: Jacobian to function CPU ratios for CPF, HHD, and CTS problems.

Problem:	CPF	HHD	CTS
Jacobian to Function CPU Ratios, CPU(∂f$/\partial$x)/CPU(f)			
ADiGator:	8.0	21.3	7.3
ADiMat:	197.0	226.3	56.3
INTLAB:	298.5	436.5	85.9
MAD:	474.8	582.6	189.8
hand:	1.3	1.2	11.3
sfd:	12.0	9.0	7.0

Next, the three test problems were solved using the MATLAB optimization toolbox functions `fsolve` (for the CPF and HHD nonlinear root-finding problems) and `lsqnonlin` (for the CTS nonlinear least squares problem). The problems were tested by supplying the MATLAB solvers with the Jacobian files generated by the ADiGator algorithm and by simply supplying the Jacobian sparsity patterns and allowing the optimization toolbox to perform sparse finite-differences. Default tolerances of the optimization toolbox were used. The results of the test are shown in Table IV, which shows the solution times, number of required Jacobian evaluations, and ADiGator file generation times. From this table, it is seen that the CPF and HHD problems solve slightly faster when supplied with the Jacobian via the ADiGator generated files, while the CTS problem solves slightly faster when used with the MATLAB sparse finite-differencing routine. It is also noted that the time required to generate the ADiGator derivative files is actually greater than the time required to solve the problems. For this class of problems, however, the dimensions of the inputs are fixed, and thus the ADiGator generated derivative files must only be generated a single time. Additionally, solutions obtained when supplying Jacobians were more accurate than those obtained using sparse finite-differences, for each of the tested problems. The differences in solutions for the CPF, HHD, and CTS problems were on the order of 10^{-7}, 10^{-14}, and 10^{-13}, respectively.

Table IV: Solution times for fixed dimension nonlinear systems.

Problem:	CPF	HHD	CTS
Solution Time (s)			
ADiGator:	0.192	0.100	0.094
sfd:	0.212	0.111	0.091
Number of Iterations			
ADiGator:	96	38	5
sfd:	91	38	5
ADiGator File Generation Time (s)			
	0.429	0.422	0.291

9.3. Large Scale Unconstrained Minimization

In this section the 2-D Ginzburg-Landau (GL2) minimization problem is tested from the MINPACK-2 test suite [Averick et al. 1991]. The problem is to minimize the Gibbs free energy in the discretized Ginzburg-Landau superconductivity equations. The objective, f, is given by

$$f = \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} -|v_{i,j}|^2 + \frac{1}{2}|v_{i,j}|^4 + \phi_{i,j}(\mathbf{v}, \mathbf{a}^{(1)}, \mathbf{a}^{(2)}), \tag{28}$$

where $\mathbf{v} \in \mathbb{C}^{n_x \times n_y}$ and $(\mathbf{a}^{(1)}, \mathbf{a}^{(2)}) \in \mathbb{R}^{n_x \times n_y} \times \mathbb{R}^{n_x \times n_y}$ are discrete approximations to the order parameter $\mathbf{V} : \mathbb{R}^2 \to \mathbb{C}$ and vector potential $\mathbf{A} : \mathbb{R}^2 \to \mathbb{R}^2$ at the equally spaced grid points $((i-1)h_x,\ (j-1)h_y)$, $1 \le i \le n_x + 1$, $1 \le j \le n_y + 1$. Periodicity conditions are used to express the problem in terms of the variables $v_{i,j}$, $a^{(1)}_{i,j}$, and $a^{(2)}_{i,j}$ for $1 \le i \le n_x + 1$, $1 \le j \le n_y + 1$. Moreover, both the real and imaginary components of $\mathbf{v}$ are treated as variables. Thus, the problem has $4n_xn_y$ variables. For the study conducted in this section, it was allowed that $n = 4n_xn_y$, $n_x = n_y$, and the standard problem parameters of $\kappa = 5$ and $n_v = 8$ were used. The code used for the tests of this section was obtained from Lenton [2005], which also contains a hand-coded gradient file.[5] For the remainder of this section, the objective function will be denoted by f, where $f : \mathbb{R}^n \to \mathbb{R}$ and the gradient function will be denoted g, where $\mathrm{g} : \mathbb{R}^n \to \mathbb{R}^n$.

In order to test the efficiency of the ADiGator tool at both the first and second derivative levels, both the objective and gradient functions, f and g, were differentiated. Thus, three different tests were performed by computing (1) the first derivative of the objective, $\partial f/\partial \mathrm{x}$; (2) the first derivative of the gradient, $\partial \mathrm{g}/\partial \mathrm{x}$; and (3) the second derivative of the objective, $\partial^2 f/\partial \mathrm{x}^2$, where $\partial \mathrm{g}/\partial \mathrm{x} = \partial^2 f/\partial \mathrm{x}^2$. The aforementioned derivatives were computed using the ADiGator, ADiMat, INTLAB, and MAD tools and results are given in Table V. Additionally, Table V provides the theoretical derivative-to-function CPU ratios that would be required if a finite difference was to be used along with the derivative-to-function ratio of the hand-coded gradient file. The results presented in Table V were obtained as follows. For the gradient computation, $\partial f/\partial \mathrm{x}$, the tested AD tools were applied to the objective function where ADiMat was used in the reverse scalar mode, and INTLAB and MAD were used in the sparse forward modes. Additionally, the hand-coded gradient g was evaluated in order to compute the hand-coded ratios, and the ratio given for a finite-difference is equal to $n + 1$. For the Jacobian computation, $\partial \mathrm{g}/\partial \mathrm{x}$, the tested AD tools were applied to the gradient func-

[5]The files obtained from Lenton [2005] unpacked the decision vector by projecting into a three dimensional array. The code was slightly modified to project only to a two-dimensional array in order to allow for use with the ADiGator tool.

tion where ADiMat was used in the forward compressed scalar mode, INTLAB was used in the sparse forward mode, and MAD was used in the forward compressed mode. The ratios given for a sparse finite difference are given as $(c+1)$ times those of the hand-coded gradient ratios, where c is the number of Hessian colors provided in the table. For the Hessian computation, $\partial^2 f/\partial \mathrm{x}^2$, the tested AD tools were applied again to the objective function where ADiMat was used in the compressed forward over scalar reverse mode (with operator overloading for the forward computation, `t1rev` option of `admHessian`), INTLAB was used in the sparse second-order forward mode, and MAD was used in the compressed forward mode over sparse forward mode. The ratios given for a finite-difference are equal to $(n+1)(c+1)$; the number of function evaluations required to approximate the Hessian via a central difference. As witnessed from Table V, the ADiGator tool performs quite well at run-time compared to the other methods. While the hand-coded gradient may be evaluated faster than the ADiGator generated gradient file, the ADiGator generated file is, at worst, only five times slower, and is generated automatically.

Table V: Derivative to function CPU ratios for 2-D Ginzburg-Landau problem. Shown are the function gradient to function CPU ratios, CPU($\partial f/\partial \mathrm{x}$)/CPU($f$), gradient Jacobian to function CPU ratios, CPU($\partial \mathrm{g}/\partial \mathrm{x}$)/CPU($f$), and function Hessian to function CPU ratios, CPU($\partial^2 f/\partial \mathrm{x}^2$)/CPU($f$), for increasing values of n, where $n = 4n_x n_y$ and $n_x = n_y$.

n:	16	64	256	1024	4096	16384
Ratios CPU($\partial f/\partial \mathrm{x}$)/CPU($f$)						
ADiGator:	6.3	6.3	7.0	9.6	10.9	12.0
ADiMat:	86.9	84.6	80.9	68.7	52.9	21.3
INTLAB:	67.9	67.1	65.4	60.1	57.7	41.0
MAD:	123.6	121.2	118.3	112.9	142.3	240.9
fd:	17.0	65.0	257.0	1025.0	4097.0	16385.0
hand:	3.8	4.2	4.2	3.8	3.8	2.5
Ratios CPU($\partial \mathrm{g}/\partial \mathrm{x}$)/CPU($f$)						
ADiGator:	33.0	38.0	39.1	39.3	49.6	50.4
ADiMat:	632.5	853.1	935.6	902.1	731.4	420.4
INTLAB:	518.7	530.4	514.7	460.0	414.1	249.1
MAD:	896.2	876.9	838.9	724.3	579.8	267.8
sfd:	64.9	87.3	100.5	99.8	95.6	66.2
Ratios CPU($\partial^2 f/\partial \mathrm{x}^2$)/CPU($f$)						
ADiGator:	9.7	10.7	13.1	20.5	45.4	62.9
ADiMat:	944.5	926.5	889.2	819.9	727.4	393.0
INTLAB:	102.4	102.3	138.4	2102.4	47260.0	-
MAD:	531.1	527.5	584.3	1947.6	19713.8	-
fd:	289.0	1365.0	6168.0	26650.0	102425.0	426010.0
Hessian Information						
# colors:	16	20	23	25	24	25
% non-zero:	62.50	19.53	4.88	1.22	0.31	0.08

As seen in Table V, the files generated by ADiGator are quite efficient at run-time. Unlike the problems of Section 9.2, however, the optimization problem of this section is not of a fixed-dimension. Moreover, the derivative files generated by ADiGator are only valid for a fixed dimension. Thus, one cannot disregard file generation times. In order to investigate the efficiency of the ADiGator transformation routine, absolute

derivative file generation times together with absolute objective file evaluation times are given in Table VI. This table shows that the cost of generation of the objective gradient file, $\partial f/\partial \mathrm{x}$, and gradient Jacobian file, $\partial \mathrm{g}/\partial \mathrm{x}$, are relatively small, while the cost of generating the objective Hessian file becomes quite expensive at $n = 16384$.[6] Simply revealing the file generation times, however, does not fully put into perspective the trade-off between file generation time costs and run-time efficiency gains. In order to do so, a "cost of derivative computation" metric is formed, based off of the number of Hessian evaluations required to solve the GL2 minimization problem. To this end, the GL2 problem was solved using the MATLAB unconstrained minimization solver, `fmincon`, in the full-Newton mode, and the number of required Hessian computations was recorded. Using the data of Tables V and VI, the metric was computed as the total time to perform the k required Hessian computations, using all of the tested AD tools. The results from this computation are given in Table VII, where two costs are given for using the ADiGator tool, one of which takes into account the time required to generate the derivative files. Due to the relatively low number of required Hessian evaluations, it is seen that the ADiGator tool is not always the best option when one factors in the file generation time. That being said, for this test example, files which compute the objective gradient and Hessian sparsity pattern are readily available. At some point in time, however, someone had to devote a great deal of time and effort towards coding these files. Moreover, when using the ADiGator tool, one obtains the Hessian sparsity pattern and an objective gradient file as a direct result of the Hessian file generation.

Table VI: ADiGator file generation times and objective function evaluation times for 2-D Ginzburg-Landau problem. Shown is the time required for ADiGator to perform the transformations: objective function f to an objective gradient function $\partial f/\partial \mathrm{x}$, gradient function g to Hessian function $\partial \mathrm{g}/\partial \mathrm{x}$, and gradient function $\partial f/\partial \mathrm{x}$ to Hessian function $\partial^2 f/\partial \mathrm{x}^2$.

n:	16	64	256	1024	4096	16384
ADiGator File Generation Time (s)						
$\partial f/\partial \mathrm{x}$:	0.51	0.51	0.52	0.53	0.58	0.90
$\partial \mathrm{g}/\partial \mathrm{x}$:	2.44	2.51	2.51	2.57	2.89	4.33
$\partial^2 f/\partial \mathrm{x}^2$:	2.12	2.13	2.23	2.33	4.85	37.75
Objective Function Evaluation Time (ms)						
f:	0.2795	0.2821	0.2968	0.3364	0.4722	1.2611

9.4. Large Scale Nonlinear Programming

Consider the following nonlinear program (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 Garg et al. [2010]. This problem was studied in Weinstein and Rao [2015] and is revisited in this paper as a means of investigating the use of the vectorized mode of the ADiGator algorithm. The problem is to determine the values of the vectorized variable $\mathbf{X} \in \mathbb{R}^{4\times N}$,

$$\mathbf{X} = [\,\mathbf{Y}\ \ \mathbf{U}\,], \quad \mathbf{Y} \in \mathbb{R}^{3\times N}, \quad \mathbf{U} \in \mathbb{R}^{1\times N}, \tag{29}$$

[6]This expense is due to the fact that the overloaded `sum` operation performs the affine transformations of Eq. (9), which become quite expensive as n is increased.

Table VII: Cost of differentiation for 2-D Ginzburg-Landau problem. A "cost of differentiation" metric is given as the time required to perform k Hessian evaluations, where k is the number of Hessian evaluations required to solve the GL2 optimization problem using `fmincon` with a trust-region algorithm and function tolerance of $\sqrt{\epsilon_{machine}}$. Results are presented for three cases of computing the Hessian: sparse-finite differences over AD, AD over the hand-coded gradient, and AD over AD. Times listed for ADiGator* are the times required to compute k Hessians plus the time required to generate the derivative files, as given in Table VI.

n:	16	64	256	1024	4096	16384
# Hes Eval:	9	11	26	21	24	26
# colors:	16	20	23	25	24	25
Cost of Differentiation: SFD over AD (s)						
ADiGator:	0.27	0.41	1.29	1.76	3.08	10.23
ADiGator*:	0.78	0.92	1.81	2.29	3.66	11.13
ADiMat:	3.71	5.51	14.99	12.63	15.00	18.19
INTLAB:	2.90	4.37	12.11	11.04	16.36	34.94
MAD:	5.29	7.90	21.90	20.73	40.33	205.34
hand:	0.16	0.27	0.78	0.70	1.08	2.17
Cost of Differentiation: AD over Hand-Coded (s)						
ADiGator:	0.08	0.12	0.30	0.28	0.56	1.65
ADiGator*:	2.52	2.63	2.81	2.85	3.45	5.98
ADiMat:	1.59	2.65	7.22	6.37	8.29	13.79
INTLAB:	1.30	1.65	3.97	3.25	4.69	8.17
MAD:	2.25	2.72	6.47	5.12	6.57	8.78
Cost of Differentiation: AD over AD (s)						
ADiGator:	0.02	0.03	0.10	0.14	0.51	2.06
ADiGator*:	2.65	2.68	2.85	3.01	5.94	40.71
ADiMat:	2.38	2.87	6.86	5.79	8.24	12.88
INTLAB:	0.26	0.32	1.07	14.85	535.61	-
MAD:	1.34	1.64	4.51	13.76	223.42	-

*Includes the cost of ADiGator file generation.

the support points $\mathbf{s} \in \mathbb{R}^3$, and the parameter β, which minimize the cost function

$$J = \beta \tag{30}$$

subject to the nonlinear algebraic constraints

$$\mathbf{C}(\mathbf{X}, \mathbf{s}, \beta) = [\mathbf{Y}\ \mathbf{s}]\,\mathbf{D}^{\mathsf{T}} - \frac{\beta}{2}\mathbf{F}(\mathbf{X}) = \mathbf{0} \in \mathbb{R}^{3\times N} \tag{31}$$

and simple bounds

$$\mathbf{X}_{\min} \le \mathbf{X} \le \mathbf{X}_{\max}, \quad \mathbf{s}_{\min} \le \mathbf{s} \le \mathbf{s}_{\max}, \quad \beta_{\min} \le \beta \le \beta_{\max}. \tag{32}$$

The matrix $\mathbf{D}^{N\times(N+1)}$ of Eq. (31) is the LGR differentiation matrix [Garg et al. 2010], and the function $\mathbf{F}(\mathbf{X}) : \mathbb{R}^{4\times N} \to \mathbb{R}^{3\times N}$ of Eq. (31) is the vectorized function

$$\mathbf{F}(\mathbf{X}) = [\,\mathbf{f}(\mathbf{X}_1)\ \ \mathbf{f}(\mathbf{X}_2)\ \ \cdots\ \ \mathbf{f}(\mathbf{X}_N)\,], \tag{33}$$

where $\mathbf{X}_i \in \mathbb{R}^4$ refers to the i^{th} column of $\mathbf{X}$ and $\mathbf{f} : \mathbb{R}^4 \to \mathbb{R}^3$ is defined as

$$\begin{aligned} f_1(\mathbf{x}) &= x_2 \sin x_3, \\ f_2(\mathbf{x}) &= \tfrac{1}{c_1}\left(\zeta(T(x_1), x_1, x_2)) - \theta(T(x_1), \rho(x_1), x_1, x_2)\right) - c_2 \sin x_3, \\ f_3(\mathbf{x}) &= \tfrac{c_2}{x_2}\left(x_4 - \cos x_3\right). \end{aligned} \tag{34}$$

Moreover, the functions $T(h)$ and $\rho(h)$ are modified from the smooth functions of Darby et al. [2011] to the following piecewise continuous functions taken from NOAA [1976]:

$$\rho(h) = c_3 \frac{p(h)}{T(h)}, \tag{35}$$

where

$$(T(h), p(h)) \; = \; \begin{cases} \left(c_4 - c_5 h, c_6 \left[\frac{T(h)}{c_7}\right]^{c_8}\right) \;, & h < 11, \\ \\ (c_9, c_{10} e^{c_{11} - c_{12} h}) \qquad \;, & \text{otherwise}. \end{cases} \tag{36}$$

Unlike the implementation considered in Weinstein and Rao [2015], Eq. (36) is implemented as a sequence of logical reference and assignment operations.

In the first portion of the analysis of this problem, the NLP of Eqs. (30)–(32) is solved for increasing values of N. The number of LGR points in each interval is fixed to four and the number of mesh intervals, K, is varied. The number of LGR points is thus $N = 4K$. The NLP is first solved on an initial mesh with $K = 4$ intervals. The number of mesh intervals is then doubled sequentially, where the result of the solution of the previous NLP is used to generate an initial guess to the next NLP. Mesh intervals are equally spaced for all values of $K = 4, 8, 16, 32, 64, 128, 256, 512, 1048$ and the NLP is solved with the NLP solver IPOPT [Biegler and Zavala 2008; Waechter and Biegler 2006] in both the quasi-Newton (first-derivative) and full-Newton (second-derivative) modes. Moreover, derivatives of the NLP were computed via ADiGator using two different approaches. In the first, non-vectorized, approach, the ADiGator tool is applied directly to the function which computes $\mathbf{C}(\mathbf{X}, \mathbf{s}, \beta)$ of Eq. (31) to compute the constraint Jacobian and the Lagrangian Hessian. In the second, vectorized, approach, the ADiGator tool is applied in the vectorized mode to the function which computes $\mathbf{F}(\mathbf{X})$. The ADiGator computed derivatives of $\mathbf{F}(\mathbf{X})$ are then used to construct the NLP constraint Jacobian and Lagrangian Hessian (using discretization separability as described in Betts [2009]). Results of the tests are shown in Table VIII. In the presented table, solution times are broken into two different categories, initialization time and NLP solve time. When using the non-vectorized approach, the initialization time is the time required to generate derivative files prior to solving each NLP. When using the vectorized approach, derivative files must only be generated a single time (shown as mesh # 0). The initialization time required of the vectorized approach at each subsequent mesh iteration is the time required to compute the derivative of the linear portion of $\mathbf{C}$ (that is, $[\mathbf{X}\ \mathrm{s}]\,\mathbf{D}^{\mathsf{T}}$) plus the time required to determine sparsity patterns of the constraint Jacobian and Lagrangian Hessian, given sparsity patterns of $\partial \mathrm{f}/\partial \mathrm{x}$ and $\partial^2 \mathrm{f}/\partial \mathrm{x}^2$. It is seen in Table VIII for both quasi-Newton and full-Newton solutions that the use of the vectorized mode reduces both initialization times and NLP run times. The reason for the reduction in initialization times is fairly straightforward: when using the vectorized mode, derivative files are valid for any value of N and thus must only be generated a single time. The reason for the reduction in run-times is two-fold. First, by separating the nonlinear portion of $\mathbf{C}$ from the linear portion of $\mathbf{C}$, the first derivatives of the linear portion must only be computed a single time for each mesh, rather than at run-time. Next, as discussed in Section 7, run-time indexing overheads are effectively reduced by an order of N when using the vectorized mode over the non-vectorized. This reduction of run-time indexing overheads is greatly emphasized in the results from the full-Newton mode, where many more indexing operations are required at the second derivative level than at the first.

In the next part of the analysis of this example the vectorized function $\mathbf{F}(\mathbf{X})$ was differentiated for increasing values of N using a variety of well-known MATLAB AD

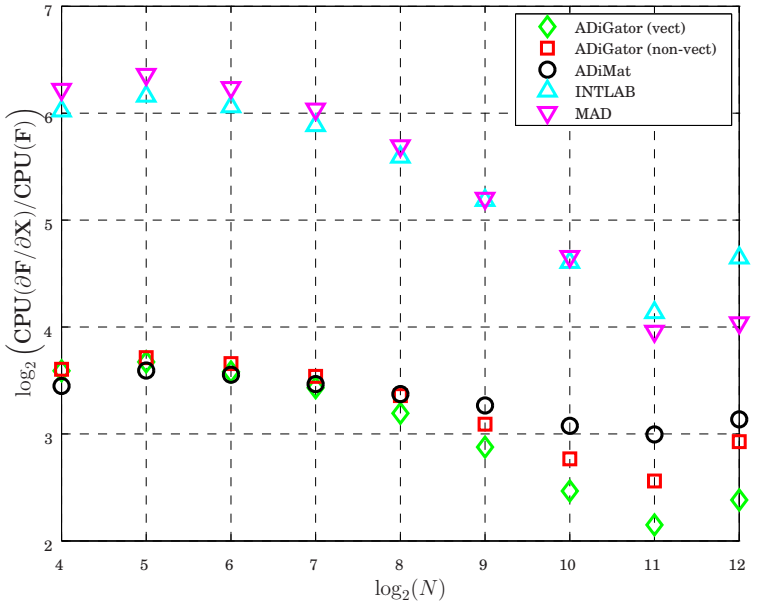
Table VIII: NLP Solution Times for Minimum Time to Climb using IPOPT and ADiGator. The NLP is solved for increasing values of K using IPOPT in both the quasi-Newton and full-Newton modes. For both modes, the ADiGator tool is used in two different ways. In the non-vectorized case (labeled non-vect), ADiGator is applied directly to the functions of the NLP. In the vectorized case (labeled vect), ADiGator is applied in the vectorized mode to the function $\mathbf{F}(\mathbf{X})$.

	mesh #:	0	1	2	3	4	5	6	7	8	9	**Total**
	K:		4	8	16	32	64	128	256	512	1024	
Quasi-Newton	# jac eval:	-	63	63	93	92	92	71	62	145	52	**733**
	Initialization Time (s)											
	non-vect:	-	1.18	1.17	1.18	1.18	1.19	1.23	1.35	1.62	2.27	**12.4**
	vect:	0.97	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	**1.0**
	NLP Solve Time (s)											
	non-vect:	-	0.59	0.61	0.96	1.12	1.45	1.65	2.37	9.41	7.00	**25.2**
	vect:	-	0.56	0.51	0.82	0.95	1.21	1.35	1.95	7.82	5.80	**21.0**
Full-Newton	# jac eval:	-	41	14	17	17	18	18	20	20	23	**188**
	# hes eval:	-	40	13	16	16	17	17	19	19	22	**179**
	Initialization Time (s)											
	non-vect:	-	5.37	5.38	5.43	5.47	5.64	6.06	7.40	11.09	24.04	**75.9**
	vect:	4.59	0.00	0.00	0.00	0.00	0.01	0.03	0.13	0.46	1.78	**7.0**
	NLP Solve Time (s)											
	non-vect:	-	0.86	0.40	0.48	0.52	0.66	0.89	1.48	2.50	5.73	**13.5**
	vect:	-	0.85	0.23	0.30	0.32	0.41	0.54	0.90	1.51	3.33	**8.4**

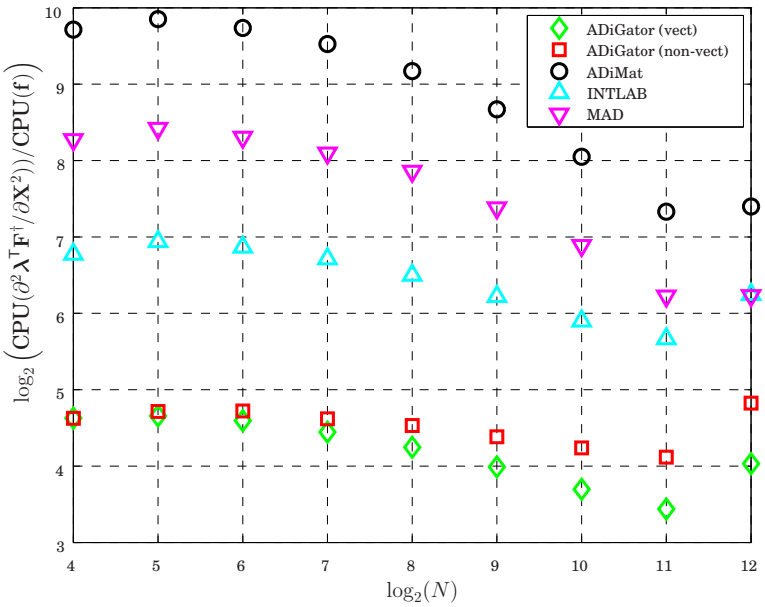
tools. At the first-derivative level, the ADiGator tool was used in the vectorized and non-vectorized modes, ADiMat was used in the compressed scalar forward mode, INTLAB was used in the sparse forward mode, and MAD was used in the compressed forward mode. At the second-derivative level, the ADiGator tool was again used in the vectorized and non-vectorized modes, ADiMat was used in the compressed forward over scalar reverse mode (with operator overloading for the forward computation), INTLAB was used in the sparse second-order forward mode, and MAD was used in the compressed forward over compressed forward mode. Results are shown in Fig. 2. At the second-derivaitve level, the ADiMat tool was used to compute $\partial^2\boldsymbol{\lambda}^\mathsf{T}\mathbf{F}^\dagger/\partial\mathbf{X}^2$, where $\boldsymbol{\lambda} \in \mathbb{R}^{3N}$ and $\mathbf{F}^\dagger \in \mathbb{R}^{3N}$ is the one-dimensional transformation of $\mathbf{F}$. All other tools were used entirely in the forward mode, and thus were simply used to compute $\partial^2\mathbf{F}/\partial\mathbf{X}^2$. The computation time CPU($\partial^2\boldsymbol{\lambda}^\mathsf{T}\mathbf{F}^\dagger/\partial\mathbf{X}^2$) was then computed as the time required to compute $\partial^2\mathbf{F}/\partial\mathbf{X}^2$ plus the time required to pre-multiply $\partial^2\mathbf{F}^\dagger/\partial\mathbf{X}^2$ by $\boldsymbol{\lambda}^\mathsf{T}$. As seen in Fig. 2, the ratios for all tested tools tend to decrease as the value of N is increased. This increase in efficiency is due to a reduction in the relevance of run-time overheads (this is very apparent for the operator overloaded INTLAB and MAD tools at the first derivative) together with the fact that the derivatives become sparser as N is increased. When comparing the results obtained from using ADiGator in the vectorized mode versus the non-vectorized mode, it is seen that the ratios diverge as the dimension of the problem is increased. These differences in computation times are due strictly to run-time overheads associated with reference and assignment indexing.

10. DISCUSSION

The presented algorithm has been designed to perform a great deal of analysis at compile-time in order to generate the most efficient derivative files possible. The results of Section 9 show these ADiGator generated derivative files to be quite efficient at run-time when compared to other well-known MATLAB AD tools. The presented results also show, however, that compile times can become quite large as problem dimen-



(a) $\log_2\left(\text{CPU}(\partial\mathbf{F}/\partial\mathbf{X})/\text{CPU}(\mathbf{F})\right)$ vs. $\log_2(N)$



(b) $\log_2\left(\text{CPU}(\partial^2\boldsymbol{\lambda}^\mathsf{T}\mathbf{F}^\dagger/\partial\mathbf{X}^2))/\text{CPU}(\mathbf{f})\right)$ vs. $\log_2(N)$

Fig. 2: Jacobian and Hessian to Function CPU Ratios for Minimum Time to Climb Vectorized Function.

sions increase. This is particularly the case when dealing with functions containing explicit loops (for example, Burgers' ODE) or those which perform matrix operations (for example, the GL2 minimization problem). Even so, the ADiGator algorithm is well suited for applications requiring many repeated derivative computations, where the cost of file generation becomes less significant as the number of required derivative computations is increased.

Next, the fact that the method has been implemented in the MATLAB language is both an advantage and a hindrance. Due to MATLAB's high level array and matrix operations, the corresponding overloaded operations are granted a great deal of information at compile-time. The overloaded operations can thus print derivative procedures that are optimal for the sequence of elementary operations which the high-level operation performs, rather than printing derivative procedures that are optimal for each of the individual elementary operations. In order to exploit derivative sparsity, however, the overloaded operations print derivative procedures which typically only operate on vectors of non-zero derivatives. Moreover, the derivative procedures rely heavily on index array reference and assignment operations (for example, `a(ind)`, where `ind` is a vector of integers). Due to the interpreted nature of the MATLAB language, such reference and assignment operations are penalized at run-time by MATLAB's array bounds checking mechanism, where the penalty is proportional to the length of the index vector (for example, `ind`) [Menon and Pingali 1999]. Moreover, the length of the used index vectors are proportional to the number of non-zero derivatives at each link in the chain rule. Thus, as problem sizes increase, so do the derivative run-time penalties associated with the array bounds checking mechanism. The increase in run-time overheads is manifested in the the results of Fig. 1 and Table V of Sections 9.1 and 9.3, respectively. For both problems, the Jacobians become relatively more sparse as the problem dimensions are increased. One would thus expect the Jacobian to function CPU ratios to decrease as problem dimensions increase. The witnessed behavior of the ADiGator tool is, however, the opposite and is attributed to the relative increase in run-time overhead due to indexing operations. When studying the vectorized problem of Section 9.4, the indexing run-time overheads may actually be quantified as the difference in run times between the vectorized and non-vectorized generated files. From the results of Fig. 2, it is seen that, at small values of N, the differences in run times are negligible. At $N = 4096$, however, the non-vectorized ADiGator generated first and second derivative files spent *at least* 32% and 42%, respectively, of the computation time performing array bounds checks from indexing operations.

11. CONCLUSIONS

A toolbox called *ADiGator* has been described for algorithmically differentiating mathematical functions in MATLAB. ADiGator statically exploits sparsity at each link in the chain rule in order to produce run-time efficient derivative files, and does not require any a priori knowledge of derivative sparsity, but instead determines derivative sparsity as a direct result of the transformation process. The algorithm is described in detail and is applied to four examples of varying complexity. It is found that the derivative files produced by ADiGator are quite efficient at run-time when compared to other well-known AD tools. The generated derivative files are, however, valid only for fixed dimensional inputs and thus the cost of file generation cannot be overlooked. Finally, it is noted that the ADiGator tool is well suited for applications requiring many repeated derivative computations.

REFERENCES

ASCHER, U., MATTHEIJ, R., AND RUSSELL, R. 1995. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.

AVERICK, B. M., CARTER, R. G., AND MORÃL', J. J. 1991. The minpack-2 test problem collection.

BETTS, J. T. 2009. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, 2 ed. SIAM Press, Philadelphia.

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

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. The efficient computation of sparse jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing 19,* 4, 1210–1233.

COLEMAN, T. F. AND VERMA, A. 2000. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Transactions on Mathematical Software 26,* 1, 150–175.

CURTIS, A. R., POWELL, M. J. D., AND REID, J. K. 1974. On the estimation of sparse jacobian matrices. *IMA Journal of Applied Mathematics 13,* 1, 117–119.

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.

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.

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

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.

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

HUANG, W., REN, Y., AND RUSSELL, R. D. 1994. Moving mesh methods based on moving mesh partial differential equations. *J. Comput. Phys 113*, 279–290.

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

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

LENTON, K. 2005. An efficient, validated implementation of the MINPACK-2 test problem collection in MATLAB. M.S. thesis, Cranfield University (Shrivenham Campus), Applied Mathematics & Operational Research Group, Engineering Systems Department, RMCS Shrivenham, Swindon SN6 8LA, UK.

MATHWORKS. 2014. *MATLAB Version R2014b*. The MathWorks Inc., Natick, Massachusetts.

MENON, V. AND PINGALI, K. 1999. A case for source-level transformations in matlab. *SIGPLAN Not. 35,* 1 (Dec.), 53–65.

NOAA. 1976. *U. S. standard atmosphere, 1976*. National Oceanic and Amospheric Administration, Washington, D.C.

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.

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

SHAMPINE, L. F. AND REICHELT, M. W. 1997. The matlab ode suite. *SIAM journal on scientific computing 18,* 1, 1–22.

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, Berlin Heidelberg, 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.

WEINSTEIN, M. J. AND RAO, A. V. 2015. A source transformation via operator overloading method for the automatic differentiation of mathematical functions in MATLAB. *ACM Transactions on Mathematical Software 42,* 1, 2:1–2:46.