SOURCE TRANSFORMATION VIA OPERATOR OVERLOADING FOR ALGORITHMIC DIFFERENTIATION IN MATLAB

By

MATTHEW J. WEINSTEIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2015

Dedicated to my grandfather, Pap, who started me on this journey so many years ago.

ACKNOWLEDGMENTS

I have many people to whom I owe thanks for where I am today and I cannot express enough how grateful I am for the generosity and kindness which has been bestowed upon me throughout my time as a PhD student and my life.

First and foremost, I would like to thank my advisor, Dr. Anil Rao. He has taught me the importance of meticulous research and helped to shape me into the person that I am today. I also owe thanks to the members of my committee, Dr. William Hager, Dr. B.J. Fregly, and Dr. Warren Dixon, for their time and support throughout the doctoral process. I would also like to thank Dr. Daniel Cole for the time he spent mentoring me and opening my eyes to the research world, and the AD community for their willingness to welcome and inform newcomers to the field.

The world of computational research can become rather monotonous and I thank my fellow VDOL members for helping to keep things interesting. Specifically, I would like to thank Michael Patterson for taking me under his wing and making me feel welcome from day one, Begum Cannataro for for her unlimited kindness and friendship, and Kathryn Graham, Joeseph Eide, Fengjin Liu, and Murat Unalm for allowing me to vent when I was frustrated and to bore them with talk of derivatives when I was not. I also could not have done this without the support of Camila Francolin. She has always kept me grounded, taught me not to sweat the little things, and most importantly, shown me the importance of enjoying my time as PhD student. For that, I thank her.

Last, but certainly not least, I give thanks to my family. They have always been there for me and supported me in the good times and the bad. I thank my father for teaching me to always put forth my best efforts, my mother for teaching me the importance of kindness, Curtis for being a big brother that I can look up to, Julie for helping me through the hard times, and Jodi and Daryl for helping to raise me as their own. I could not ask for a better family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SOURCE TRANSFORMATION VIA OPERATOR OVERLOADING FOR ALGORITHMIC DIFFERENTIATION IN MATLAB

By

Matthew J. Weinstein

May 2015

Chair: Anil V. Rao
Major: Mechanical Engineering

The problem of computing accurate and efficient derivatives is one of great importance in the field of numerical analysis. The research of this dissertation is concerned with computing derivatives of mathematical functions described by MATLAB source code by utilizing algorithmic differentiation (AD). Due to the iterative nature of the applications which require 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 times. Algorithmic differentiation is classically performed by using either operator overloading or source transformation. Source transformation techniques tend to produce derivative code with better run-time efficiency than tools which utilize operator overloading at run-time, because optimizations may be performed at compile-time to make the run-time computations more efficient. Source transformation tools are, however, much more complicated to implement. The methods developed in this dissertation perform source transformation via a non-classical operator overloaded approach. While there exist other MATLAB AD tools which utilize source transformation and operator overloading, the source transformation via operator overloading method of this dissertation is unique. Unlike conventional source transformation tools, the optimizations which are performed at compile-time are not those of dead-code and common sub-expression elimination. Instead, derivative sparsity patterns are determined at compile time for each link in the calculus chain rule. These known derivative sparsity

patterns are then exploited to print derivative procedures which typically operate only on vectors of non-zero derivatives. Thus, the method statically exploits sparsity at each link in the chain rule, without requiring any a priori knowledge of Jacobian or Hessian sparsity patterns. Moreover, as the result of the source transformation is a stand-alone MATLAB procedure, the method may be applied recursively to generate n^{th}-order derivative code. Hessian symmetry, however, is not exploited.

The methods of this dissertation have been implemented in the open-source MATLAB algorithmic differentiation package, ADiGator. The ADiGator tool is tested on a variety of examples and is found to produce run-time efficient derivative code. Due to the fact that the implementation determines sparsity patterns at compile-time, the resulting derivative codes are only valid for fixed input sizes. Thus, the cost of generating the derivative files cannot be overlooked. It is shown that while file generation can be expensive for large problem sizes, the compile-times become relatively insignificant for problems which require a large number of derivative evaluations.

CHAPTER 1
INTRODUCTION

Throughout the age of modern computing a great deal of research has been devoted to the field of numerical analysis. A large sub-field of numerical analysis consists of solving nonlinear systems; for example, root finding, optimization, stiff ordinary differential equation integration, etc. The most common numerical schemes used to solve such nonlinear problems are based upon Newton methods which require the use of first and sometimes second order derivatives in order to obtain a solution to the problem under consideration. As problems must generally be solved iteratively, it is often the case that the same derivative must be calculated at many different points. Thus, the efficiency of the method depends heavily upon the accuracy of the derivative and the efficiency with which the derivatives can be computed. This dissertation is concerned with such accurate and efficient derivative computations.

The first, and perhaps most obvious, choice of computing required derivatives is to do so by hand. In doing so, one can derive an exact expression of the derivative, perform hand simplifications, and then transcribe the derivative expression to a computer code. The issue with computing derivatives by hand, however, is that as problems increase in size and complexity, the task of manual differentiation becomes daunting and is inherently error prone. In order to save time and eliminate some aspects of human error, an alternative to computing derivatives by hand is to use a symbolic differentiation tool such as Maple [40], Mathematica [61] or the MATLAB Symbolic Toolbox [37] and then transcribe the resulting expression to a computer code. While symbolic differentiation is appealing, it is known to suffer from expression explosion as the derivative expressions become long and unwieldy. Moreover, symbolic differentiation tools cannot be applied directly to a user function that contains branching statements.

By far the most common approach for computing derivatives computationally is to employ some kind of finite-difference approximation. In a finite-differencing approach, the

derivative is approximated by evaluating the function at two or more neighboring points and dividing the difference of the function by the difference of the points. The use of finite differences is appealing due to the fact that only evaluations of the user function are required. The accuracy of the derivative, however, depends completely upon the spacing of the neighboring points. If the spacing between the neighboring points is too small, then the approximation becomes inaccurate due to machine roundoff error. If, on the other hand, the spacing is too large, then the approximation becomes inaccurate due to truncation error. Moreover, even if the spacing is chosen well, one can only expect the derivative to be accurate to roughly one-half to two-thirds the significant digits of the original function. The number of significant figures of accuracy for second and higher-order derivative approximations is even smaller than it is for a first derivative approximation.

The desire for a method that accurately and efficiently computes derivatives automatically has lead 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 [28]. 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 errors.

Algorithmic differentiation may be performed in either the forward or reverse mode. In both modes, 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 derivative calculations are performed. 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 algorithmic differentiation methods are classically implemented using either operator overloading or source transformation. In an operator overloaded approach, a custom class is constructed and all standard arithmetic operations and mathematical functions are defined to operate on objects of the class. Any object of the custom class typically contains properties that include the function value and derivatives of the object at a particular numerical value of the input. Furthermore, when any operation is performed on an object of the class, both function and derivative calculations are executed from within the overloaded operation. In a source transformation approach, typically a compiler-type software is required to read the source code of the function program and create a new derivative program, 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. Well known implementations of forward and reverse mode AD that utilize operator overloading include MXYZPTLK [39], ADOL-C [29], COSY INFINITY [6], ADOL-F [56], FADBAD [4], IMAS [50], AD01 [48], ADMIT-1 [15], ADMAT [13], INTLAB [53], FAD [2], MAD [21], and CADA [44], while well known implementations which utilize source transformation include DAFOR [5], GRESS [31], PADRE2 [35], Odysée [52], TAF [27], ADIFOR [11, 12], PCOMP [19], ADiMat [10], TAPENADE [30], ELIAD [58] and MSAD [34].

In recent years, MATLAB [37] 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, also make source transformation (in the

classical sense) quite difficult. In this dissertation a method is described for performing source transformation via the non-classical methods of operator overloading and source reading for the algorithmic differentiation of MATLAB programs. The described methods have been implemented in the MATLAB open source AD package, ADiGator.

The research of this dissertation is divided into two parts: the development of an overloaded class which transforms basic blocks of function code into basic blocks of efficient derivative code, and the development of a method which utilizes the overloaded class to transform user programs into efficient derivative programs. The developed 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 overloaded class allows for the developed algorithm to evaluate basic blocks of code on overloaded objects and print the corresponding derivative procedures to file. The derivative procedures are printed to be as efficient as possible, given the information stored within the overloaded objects.

A source transformation method is then developed which utilizes the overloaded class to transform function codes containing flow control statements (for example, conditional `if` and `for` loop statements) into derivative codes containing the same flow control constructs. The source transformation method is based upon two key concepts: the pseudo-overloading of flow control statements and the overmapping of overloaded objects. In order to handle flow control statements, the algorithm must first have a way of identifying their existence. As flow control statements cannot simply be overloaded, a method is developed which allows for their pseudo-overloading by performing an initial user source to intermediate source transformation. The other difficulty which arises when using the overloaded class in the presence of flow control is that the derivative procedures printed by the overloaded evaluations are only valid for fixed sparsity patterns.

Moreover, in the presence of flow control, it is the case that a variable's derivative sparsity pattern may not be fixed at compile-time. In order to deal with these issues, a method is developed for determining an "overmapped object" for each variable in the originating program, where the overloaded evaluation of the intermediate program on overmapped objects results in derivative computations being printed which are valid for any of the given flow control constructs.

This dissertation is organized as follows. In Chapter 2, background information is given on algorithmic differentiation and the difficulties which arise when performing AD in MATLAB. Key topics such as static sparsity exploitation and the run-time overheads inherent to the MATLAB language are discussed. In Chapter 3, an overloaded class is presented which may be utilized to transform basic blocks of function code into basic blocks of derivative code. In Chapter 4, a method is developed which utilizes the developed overloaded class to transform function codes containing flow control statements into derivative codes containing the same flow control constructs. The ADiGator tool is then tested against against other well-known MATLAB AD tools in Chapters 5 and 6. In Chapter 5, a discussion is given on the efficient use of the ADiGator tool for solving optimal control problems via direct collocation, and two test examples are solved using the MATLAB optimal control software GPOPS-II [46] in conjunction with the developed ADiGator tool. In Chapter 6, the algorithm is further tested on three different problem types: stiff ordinary-differential equation integration, non-linear root finding, and non-linear unconstrained minimization. Finally, in Chapter 7, conclusions and future work are presented.

CHAPTER 2
BACKGROUND

In this chapter the background information which is necessary in understanding the scope of the research is given. First, a set of notations and conventions to be used throughout the length of the dissertation are presented. The basics of algorithmic differentiation are then introduced. Topics include the forward and reverse modes of AD, static versus dynamics sparsity exploitation, and the traditional implementations of source transformation and operator overloading. The various existing MATLAB AD tools are then reviewed and a discussion is given on the run-time overheads inherent to the MATLAB language.

2.1 Notations and Conventions

In this dissertation the following notation is employed. First, consider the matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, such that

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

where $X_{i,j},\ (i = 1, \ldots, m),\ (j = 1, \ldots, n)$ are the elements of the $m \times n$ matrix $\mathbf{X}$. Consequently, if $\mathbf{F} : \mathbb{R}^{m \times n} \longrightarrow \mathbb{R}^{p \times q}$ is a matrix function of the matrix variable $\mathbf{X}$, then $\mathbf{F}(\mathbf{X})$ has the form

$$\mathbf{F}(\mathbf{X}) = \begin{bmatrix} F_{1,1}(\mathbf{X}) & F_{1,2}(\mathbf{X}) & \cdots & F_{1,q}(\mathbf{X}) \\ F_{2,1}(\mathbf{X}) & F_{2,2}(\mathbf{X}) & \cdots & F_{2,q}(\mathbf{X}) \\ \vdots & \vdots & \ddots & \vdots \\ F_{p,1}(\mathbf{X}) & F_{p,2}(\mathbf{X}) & \cdots & F_{p,q}(\mathbf{X}) \end{bmatrix} \in \mathbb{R}^{p \times q},$$

where $F_{k,l}(\mathbf{X}),\ (k = 1, \ldots, p),\ (l = 1, \ldots, q)$ are the elements of the $p \times q$ matrix function $\mathbf{F}(\mathbf{X})$. The Jacobian of the matrix function $\mathbf{F}(\mathbf{X})$, denoted $\nabla_{\mathbf{X}}\mathbf{F}(\mathbf{X})$, is then

a four-dimensional array of size $p \times q \times m \times n$ that consists of $pqmn$ elements. This multi-dimensional array will be referred to generically as the rolled representation of the derivative of $\mathbf{F}(\mathbf{X})$ with respect to $\mathbf{X}$ (where the term "rolled" is similar to the term "external" as used in [21]). In order to provide a more tractable form for the Jacobian of $\mathbf{F}(\mathbf{X})$, the matrix variable $\mathbf{X}$ and matrix function $\mathbf{F}(\mathbf{X})$ are transformed into the following so called unrolled form (where, again, the term "unrolled" is similar to the term "internal" [21]). First, $\mathbf{X} \in \mathbb{R}^{m \times n}$, is mapped isomorphically to a column vector $\mathbf{X}^\dagger \in \mathbb{R}^{mn}$, such that

$$X_k^\dagger = X_{i,j}, \quad \text{where} \quad k = i + m(j-1), \quad \forall \quad \begin{array}{rcl} i & = & 1, \ldots m \\ j & = & 1, \ldots n \\ k & = & 1, \ldots mn \end{array} .$$

Similarly, a one-dimensional representation of the function $\mathbf{F}(\mathbf{X}) \in \mathbb{R}^{p \times q}$ is given by letting $\mathbf{F}^\dagger(\mathbf{X}^\dagger) \in \mathbb{R}^{pq}$ be the corresponding one-dimensional transformation, such that

$$F_k^\dagger(\mathbf{X}) = F_{i,j}(\mathbf{X}), \quad \text{where} \quad k = i + q(j-1), \quad \forall \quad \begin{array}{rcl} i & = & 1, \ldots q \\ j & = & 1, \ldots p \\ k & = & 1, \ldots qp \end{array} .$$

Using the one-dimensional representations, $\mathbf{X}\dagger$ and $\mathbf{F}^\dagger(\mathbf{X})$, the four-dimensional Jacobian, $\nabla_{\mathbf{x}}\mathbf{F}(\mathbf{X})$, may be represented in two-dimensional form as

$$\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger(\mathbf{X}^\dagger) = \begin{bmatrix} \frac{\partial F_1^\dagger}{\partial X_1^\dagger} & \frac{\partial F_1^\dagger}{\partial X_2^\dagger} & \cdots & \frac{\partial F_1^\dagger}{\partial X_{mn}^\dagger} \\ \frac{\partial F_2^\dagger}{\partial X_1^\dagger} & \frac{\partial F_2^\dagger}{\partial X_2^\dagger} & \cdots & \frac{\partial F_2^\dagger}{\partial X_{mn}^\dagger} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_{pq}^\dagger}{\partial X_1^\dagger} & \frac{\partial F_{pq}^\dagger}{\partial X_2^\dagger} & \cdots & \frac{\partial F_{pq}^\dagger}{\partial X_{mn}^\dagger} \end{bmatrix} \in \mathbb{R}^{pq \times mn}. \tag{2–1}$$

Eq. (2–1) provides what is referred to as the "unrolled Jacobian".

2.2 Algorithmic Differentiation

The methods for algorithmic differentiation are based upon the fact that any differentiable function may be broken into a sequence of elementary operations, e.g. addition,

multiplication, trigonometric functions, catenations, etc. As a result, one may calculate the derivative of the entire function through the calculation of each elementary operation's derivative together with a systematic application of the calculus chain rule. It should be stressed that algorithmic differentiation is not the same as symbolic differentiation. In a symbolic differentiation approach, an expression for the derivative is built, where the expression is built by breaking the entire problem into its elementary operations and applying the rules of differential calculus. In an algorithmic differentiation approach, however, the derivative is calculated at each intermediate step. In this section further clarity on how this may be achieved is given by introducing the two modes of AD as well as the classical implementation methods of operator overloading and source transformation.

2.2.1 The Forward and Reverse Modes of AD

There are two modes of applying algorithmic differentiation, the forward or the reverse. In order to illustrate the differences between the two modes the function $\mathbf{y} = \mathbf{f}(\mathbf{g}(\mathbf{h}(\mathbf{x})))$ is considered, where $\mathbf{f} : \mathbb{R}^p \to \mathbb{R}^q$, $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^p$, and $\mathbf{h} : \mathbb{R}^m \to \mathbb{R}^n$ are differentiable elementary operations. If this function were to be coded and evaluated on a computer, $\mathbf{y}$ would be calculated by the sequence: $\mathbf{v}_0 = \mathbf{x}$, $\mathbf{v}_1 = \mathbf{h}(\mathbf{v}_0)$, $\mathbf{v}_2 = \mathbf{g}(\mathbf{v}_1)$, $\mathbf{v}_3 = \mathbf{f}(\mathbf{v}_2)$, $\mathbf{y} = \mathbf{v}_3$. The manner in which the derivative $\nabla_{\mathbf{x}}\mathbf{y}$ would be calculated using both the forward and reverse modes of AD is shown in Figure 2-1. Within this figure it is seen that, in the forward mode, the derivative matrices $\nabla_{\mathbf{x}}\mathbf{v}_i$ $(i = 0, 1, 2, 3)$ are propagated forward through the program, where the chain rule is applied in parallel with the function calculations. In the reverse mode, two sweeps of the program are required. In the forward sweep, all function calculations are performed, each intermediate variable is stored, together with a record of the operations which take place. In the reverse sweep, the stored information is used to propagate and compute the adjoint matrices $\nabla_{\mathbf{y}}\mathbf{v}_i$ $(i = 3, 2, 1, 0)$. From this example it is seen that, fundamentally, the same derivative calculations are performed, but the order in which the chain rule is applied is opposite.

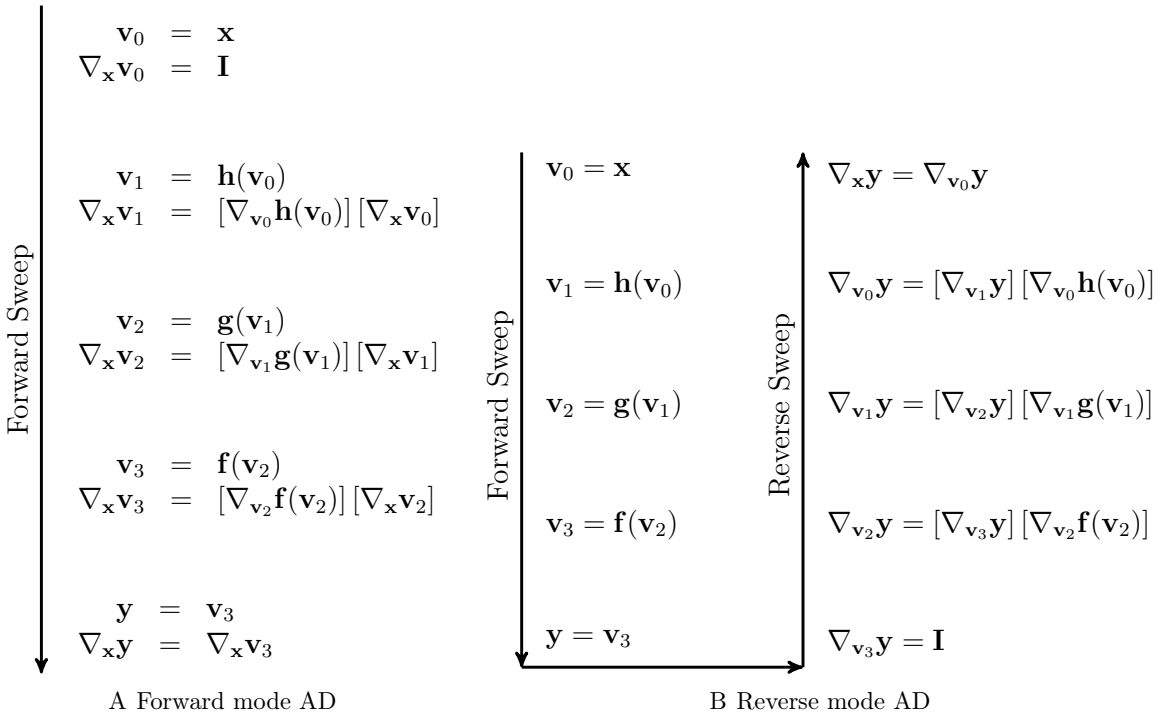


Figure 2-1. Forward versus reverse mode AD.

In order to investigate the advantages and disadvantages of both the forward and reverse mode, the same example is considered. First, consider that the derivative matrices propagated in the forward mode, $\nabla_{\mathbf{x}}\mathbf{v}_i$ $(i = 0, 1, 2, 3)$, all have a fixed column dimension of m and a row dimension of m, n, and p, respectively. Now, consider that the adjoint matrices propagated in the reverse mode, $\nabla_{\mathbf{v}_i}\mathbf{y}$, $(i = 3, 2, 1, 0)$, all have a fixed row dimension of q and a column dimension of p, n, m, respectively. Thus, if $m = p$, the cost of the evaluations shown in Figure 2-1 for the forward and reverse modes would be the same. If, however, $m >> p$, then the cost of the reverse mode calculations would be less than those of the forward mode. Conversely, if $m << p$, then the forward mode calculations would become more efficient.

From an evaluation efficiency standpoint, it should be clear that if the input is of lesser dimension than the output, then the reverse mode is favorable over the forward mode. Moreover, if the output is of a lesser dimension than the input, then the forward mode is more favorable over the reverse mode. From an implementation standpoint,

however, the reverse mode is much more difficult to implement than the forward mode. That is, in the forward mode, the derivatives are evaluated in the same order in which the program is evaluated. In the reverse mode, a great deal of effort must be spent in order to record the trace of the program, store variables, and to then traverse the program in the reverse order. As the research in this thesis is based off of the forward mode, the remaining discussion on algorithmic differentiation will be presented assuming the forward mode is being used.

2.2.2 Sparsity Exploitation

As seen in Section 2.2.1, Jacobians may be computed via AD by propagating and operating upon derivative matrices. As the number of dependent and/or independent variables increases, these matrices too can become quite large. Often times, however, such matrices tend to contain many zero elements. Moreover, the exploitation of such sparse derivative structures can result in substantial computational savings. In this section, a discussion is given on the exploitation of derivative sparsity in order to faster complete the chain rule in AD.

Given a function program, derivative sparsity may be exploited either dynamically or statically. When using a dynamic approach, dynamic data structures are used to propagate both derivative sparsity patterns and derivative non-zeros at run-time. Typically, this is achieved by storing the row number, column number, and value of all non-zero entries of the derivative. Each time an operation is to be performed on the derivative matrix, only the non-zero entries are operated upon and the row number, column number, and value of the resulting matrix are calculated. Performing such sparse operations allows one to exploit sparsity at every link in the chain rule, however, an inherent cost is associated with propagating sparsity patterns. Moreover, as the sparsity propagation is performed at run-time, the aforementioned cost is incurred each time the derivative program is evaluated.

To statically exploit derivative sparsity is to detect sparse structures and use these structures to optimize the derivative program at compile-time. Thus, any time penalty associated with the detection and exploitation of sparsity patterns must only be incurred a single time. Matrix compression techniques are the most common form of static sparsity exploitation. In a column compression approach, used with the forward mode, a seed matrix $\mathbf{S} \in \mathbb{R}^{n \times p}$ is found such that the Jacobian $\nabla_{\mathbf{x}}\mathbf{y} \in \mathbb{R}^{m \times n}$ is fully recoverable from the matrix

$$\mathbf{B} = \nabla_{\mathbf{x}}\mathbf{y}\mathbf{S} \in \mathbb{R}^{m \times p}.$$

Given such a seed matrix, $\mathbf{B}$ may be computed in the forward mode by propagating and operating upon matrices of column dimension $p \leq n$.[1] The most commonly used matrix compression technique is the Curtis-Powell-Reid (CPR) approach of Ref. [16], which has its roots in sparse-finite differencing. The CPR approach is based upon the fact that if there exist two elements of the input, x_j and x_k, such that $\frac{\partial y_i}{\partial x_j} = 0$ or $\frac{\partial y_i}{\partial x_k} = 0$ $\forall i = 1, \ldots, m$, then $\mathbf{f}(\mathbf{x} + \delta\mathbf{e}^{(j)})$ and $\mathbf{f}(\mathbf{x} + \delta\mathbf{e}^{(k)})$ may be computed simultaneously by evaluating $\mathbf{f}(\mathbf{x} + \delta\mathbf{e}^{(j)} + \delta\mathbf{e}^{(k)})$, where $\mathbf{e}^{(j)} \in \mathbb{R}^n$ and $\mathbf{e}^{(k)} \in \mathbb{R}^n$ are the j^{th} and k^{th} Cartesian basis vectors. In such a case, columns j and k would be said to share the same color. Moreover, the CPR seed matrix is formed by compressing the columns of the identity matrix which share the same color. When choosing a seed matrix, it is desired to minimize the column dimension p, equivalent to minimizing the number of colors of the CPR seed matrix. While it is known that the optimal value of p is equal to the maximum number of non-zeros across all rows $\nabla_{\mathbf{x}} y_i$, $i = 1, \ldots, m$, the determination of the optimal seed matrix is NP-hard. As such, heuristic algorithms are used to compute $\mathbf{S}$ such that p is typically only slightly larger than the minimal value. A comprehensive introduction of

[1] When using the reverse mode, a similar approach is used in order to reduce the number of adjoint rows.

such algorithms together with a more in-depth discussion on matrix compression may be found in [26].

When comparing static versus dynamic sparsity exploitation, it would seem that a static approach is more favorable, assuming that the required overhead is not outrageously large. That being said, the most widely used static technique, matrix compression, is not without its flaws. Namely, matrix compression requires a priori knowledge of the output Jacobian structure. Moreover, only the structure of the program as a whole may be exploited. Pending the output Jacobian is in-compressible (contains a full row of non-zeros), then the problem must be partially separated. When using the standard dynamic approach, however, sparsity may be exploited at each step in the chain rule, at the expense of a (sometimes significant) run-time cost.

2.2.3 Classical Implementation Methods of AD

Algorithmic differentiation is classically implemented using one of two methods: operator overloading or source transformation. The operator overloading approach may be used in any programming language which allows the user to define custom overloaded classes. When using operator overloading, a custom class is built such that, when any intrinsic operation (`+`, `*`, etc.) or library operator (`sin`, `log`, etc.) is used on an object of the custom class, a user written routine is called rather than performing the default operation. Typically, when using operator overloading to implement AD, the custom class is built such that each object contains the numeric value of the corresponding variable as well as the numeric value of the derivative. Then, prior to performing AD, a routine must be written for each operation to be used, where the routine must compute both the function and derivative values corresponding to the operation which it represents. Thus, given a program consisting of only operations which have been overloaded, together with an overloaded instance of the class as the input, the program may be evaluated in order to compute the derivative of the output. By implementing AD in this manner, the rules of differentiation are determined at the time of which the program is evaluated, where the

program is broken down into its elementary operations by simply evaluating the program itself.

In a source transformation approach, a function source code is transformed into a derivative source code, where, when evaluated, the derivative source code computes a desired derivative. An AD tool based on source transformation may be thought of as an AD preprocessor, consisting of both a compiler and a library of differentiation rules. As with any preprocessor, source transformation is achieved via four fundamental steps: parsing of the original source code, transformation of the program, optimization of the new program, and the printing of the optimized program. In the parsing phase, the original source code is read and transformed into a set of data structures which define the procedures and variable dependencies of the code. This information may then be used to determine which operations require a derivative computation, and the specific derivative computation may be found by means of the mentioned library. In doing so, the data representing the original program is augmented to include information on new derivative variables and the procedures required to compute them. This transformed information then represents a new derivative program, which, after an optimization phase, may be printed to a new derivative source code.

Analogous to the forward and reverse modes of AD, the implementation methods of operator overloading and source transformation may both be used to compute accurate derivatives, but are fundamentally different. Consequently, each method has its advantages and disadvantages. In an operator overloaded approach, the AD tool is unaware of the big picture that is the entire user program. It is this isolation that makes overloaded tools both extremely robust as well as relatively simple to implement. Moreover, due to the fact that all numerical values are known at the time of differentiation, issues pertaining to flow control in an operator overloaded approach are a non-factor. In contrast, an AD transformation tool is aware of the big picture and can thus perform optimizations such as eliminating unnecessary derivative calculations. In order to do so, however, a

transformation tool must first be able to extract complex variable dependencies at the time of compilation and make conservative assumptions when the dependencies are uncertain. Thus, full exploitation of sparsity can be difficult in a source transformation approach.

In addition to the performance gains resulting from preprocessor optimizations, source transformation has other advantages. In an application where a derivative calculation is to be performed repeatedly, the overhead associated with operator overloading is incurred at each iteration, whereas the overhead associated with source transformation is only incurred a single time. Furthermore, as a source transformation tool produces source code, it can, in theory, be applied recursively to produce a second- or higher-order derivative file (though symmetry may not be exploited). To produce a second- or higher-order derivative by means of operator overloading, however, generally requires one to code the higher-order rules of differentiation; a much more difficult task than simply coding the first-order rules. While source transformation has many advantages from a run-time efficiency standpoint, it is unfortunately extremely complex to implement.

2.2.4 Existing MATLAB AD Tools

The first comprehensive AD tool written for MATLAB was the operator overloaded tool ADMAT of Coleman and Verma [13, 14]. The ADMAT implementation may be used to perform both the forward and reverse modes to compute gradients, Jacobians and Hessians. Later, the ADMAT tool was interfaced with the ADMIT tool [15], 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 [53], 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 package MAD [21] has been developed. While MAD also employs operator overloading, unlike previously developed MATLAB AD tools, MAD utlizes the derivvec class to store directional derivatives within instances of the fmad class. By utilizing a special class in

order to store directional derivatives, the MAD toolbox is able to be used to compute n^{th} order derivatives by stacking overloaded objects within one another. While the stacking of overloaded objects allows for one to compute n^{th} order derivatives with only a first order method, it requires that the penalties associated with operator overloading be incurred at multiple levels. 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 transfomation and operator overloaded package ADiMat [9] has been developed. ADiMat employs source transformation to create a derivative source code. The derivative code may then be evaluated in a few different ways. If only a single directional derivative is desired, then the generated derivative code may be evaluated independently on numeric inputs in order to compute the derivative; this is referred to as the scalar mode. Thus, a Jacobian may be computed by a process known as strip mining, where each column of the Jacobian matrix is computed separately. In order to compute the entire Jacobian in a single evaluation of the derivative file, it is required to use either an overloaded derivative class or a collection of ADiMat specific run-time functions. Here it is noted that the derivative code used for both the scalar and overloaded modes is the same, but the generated code required to evaluate the entire Jacobian without overloading is slightly different as it requires that different ADiMat function calls be printed. The most recent MATLAB source transformation AD tool to be developed is MSAD, which was designed to test the benefits of using source transformation together with MAD's efficient data structures. The first implementation of MSAD [34] 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 [33], however, does not depend upon operator overloading but still maintains the efficiencies of the derivvec class.

2.3 Run-Time Overheads in MATLAB

The interpreted nature of the MATLAB language provides users with a great deal of flexibility when writing code, a large portion of which results from the fact that no variable declarations must ever be made. Thus, MATLAB variables must not be declared to be of any discerning type or shape. Moreover, after a variable is instantiated as an object of a certain type or shape, it may be instantiated as a completely different type or shape later in the program. The allowance of such flexibilities, however, comes at the price of inherent run-time overheads. A comprehensive discussion on the minimization of such overhead costs is given in [38], and re-emphasized in the context of AD in [33]. In this section, the various causes of the run-time overheads are revisited and used to construct generalizations on how one may improve run-times of MATLAB code.

1. Type and shape checking/dispatch: The first overhead results from the fact that MATLAB operators are ambiguous, where the determination of the meaning of the operator is dependent upon the type(s) and shape(s) of the operand(s). As MATLAB variables are not declared to have a fixed type or shape, the interpreter must use the types and shapes of operands to check for compatibility and dispatch to the proper routine.

2. Dynamic resizing: Due to the fact that matrices are not required to be declared by the programmer nor maintain a fixed size, the allocation of matrix memory is performed on demand. Thus, if an attempt is made to write to the i^{th} element of an object of dimension $m < i$, MATLAB must allocate new storage for the larger object of dimension i. Moreover, in addition to the assignment of the i^{th} element, the elements of the smaller object must be copied over to the first m elements of the larger object. In contrast, if an attempt is made to write to the i^{th} element of an object of dimension $p \geq i$, then the i^{th} element of the object is simply replaced.

3. Array bounds checking: The final notable overhead occurs when using an index to refer to an element of an array, on the left or right of the equal sign. When used on the right hand side (as a reference index), an index which exceeds the variable bounds produces an error, and when used on the left hand side (as an assignment index), an index which exceeds the variable bounds invokes the previously discussed dynamic variable resizing. This overhead becomes particularly noteworthy when index vectors are used to refer to multiple elements of an array, as each element of the index vector must be checked against the variable bounds.

Given the presented overheads of note, the following generalizations may be made on how to reduce run-time penalties associated with overhead:

1. Minimization of Operations: Due to the type and shape checking/dispatch overhead, it is the case that the invocation of all operations has in inherent run-time penalty. Thus, if the same mathematical procedure may be carried out in a variety of ways, it is often the case that the method which utilizes the least number of operations is most efficient. The first most common way of reducing the number of MATLAB operations is that of loop vectorization. When performing vectorization, one makes use of MATLAB's high-level array and matrix operations to carry out procedures which otherwise require a loop or many repeated operations. For instance, the fragment of code

```
b = 0;
for i = 1:length(a)
   b = b+a(i);
end
```

   is more efficiently replaced by the single MATLAB `sum` operation, `b = sum(a)`. The other common code optimization technique which results in a reduction of operations is that of common subexpression elimination. Common subexpression elimination (CSE) is a common compiler optimization technique which replaces identical expressions with single holding variables. CSE can be applied directly by the programmer to both reduce penalties associated with type and size checks as well as the computational costs of the operations themselves.

2. Array Preallocation: The dynamic resizing of arrays results in significant time penalties due to repeated reallocation and copying. In order to avoid such penalties, one should always preallocate arrays. In the event that array sizes are indeterminable, arrays may be over-allocated to at least reduce the number of dynamic resizes. Once the final array is built, the unnecessary elements may then be reduced.

3. Reduction of Reference/Assignment Index Elements: As previously discussed, MATLAB must perform a check on each index element used to refer to an element of an array. This check can prove to be quite costly, particularly when using vectors of indices to refer to multiple elements of an array at a time. Though it is often not possible to completely eliminate the use of reference and assignment indices, one may improve their code by keeping the cost of references and assignment in mind. One way in which the number of index references may be reduced is by the previously discussed code vectorization techniques. Another way is to make use of the `':'` MATLAB reference character. For instance, one may refer to the i^{th} row of a matrix $\mathbf{A} \in \mathbb{R}^{r \times q}$ in a number of ways. First, by use of an index variable, one may perform

```
index = 1:q;
Ai = A(i,index);
```

which would require MATLAB to perform a check on all q elements of the variable `index`, together with a check on the variable `i`. In order to decrease run-time, one may instead perform

```
Ai = A(i,1:q);
```

which would require MATLAB to perform a check on only `i`, `1`, and `q`. The most efficient way of referencing the i^{th} row, however, is given by

```
Ai = A(i,:);
```

which would only require MATLAB to perform a single check on the variable `i`.

Here it is important to note that the above code optimization techniques are merely generalizations on how to reduce the discussed run-time overheads. These generalizations do not, however, take into account the underlying mathematical procedures which operations must perform. Thus, they may be used as a set of guidelines for producing efficient code, rather than rules. For instance, just because a procedure may be written as a single MATLAB operation, it does not always mean that it is most efficient to perform the procedure using the single operation. This is particularly the case whenever information is available pertaining to the sparseness of variables. For example, consider again the summation of all elements of a vector $\mathbf{a} \in \mathbb{R}^m$, $m > 3$, but add in the fact that it is known only the i, j, k elements of $\mathbf{a}$ are non-zero. The summation of the i, j, k elements of $\mathbf{a}$ may be carried out by use of the single MATLAB `b = sum(a)` operation. In doing so, $m - 3$ redundant computations would be performed within the `sum` routine. Knowing that only the elements i, j, k of $\mathbf{a}$ are non-zero, however, one may much more efficiently perform

```
b = a(i) + a(j) + a(k);
```

even though the procedure requires multiple operations as well as multiple index references. Thus, the determination of the most efficient way to perform a procedure in MATLAB is largely procedure dependent, yet it is important to be aware of the presented run-time overheads.

CHAPTER 3
DIFFERENTIATING BASIC BLOCKS VIA THE CADA CLASS

The objective of the presented research is to develop a method of computing accurate and efficient first- and second-order derivatives of mathematical functions coded in MATLAB. Moreover, it is always assumed that the same derivatives are to be evaluated many times, at different numeric input values. While there already exist MATLAB AD tools capable of computing first and second derivatives, it is believed that there is a more efficient way of obtaining them, at least in terms of evaluation time. In this chapter, a new overloaded CADA class is presented. Unlike conventional operator overloaded objects used for AD, CADA objects do not, in general, store numeric function or derivative values. Rather than evaluating a section of code and simultaneously computing derivative values, the CADA class is used to evaluate a section of code and simultaneously print derivative procedures to a file. The new printed file may then be evaluated numerically to compute the derivatives. Thus, the result of evaluating a section of code on CADA objects is similar to the result of using a source transformation tool, however, compiler type software is not required. In this dissertation the discussion on the CADA class is restricted to vector functions of vectors, however, it should be noted that the methods discussed are directly applicable to matrix functions of matrices. In particular, this is achieved by using the unrolled form of Eq. (2–1).

3.1 Motivation

When comparing AD tools which utilize a pure source transformation approach to those which utilize a pure operator overloaded approach, it is usually the case that the source transformation tool will perform better at run-time. That is, the generated derivative source code will typically evaluate faster than evaluating the original source code on overloaded objects. When dealing with MATLAB code, the differences in time are due largely to the type and size checking/dispatch discussed in Section 2.3. Thus, not only is a penalty incurred at the call to each overloaded operation, but additionally at

each operation contained within the overloaded procedure itself. In regards to the target applications of this dissertation, it is the case that the derivatives of the problem must be computed repeatedly during the solution of the problem. Thus, if an operator overloaded AD tool is used, the penalties associated with operator overloading would need to be incurred each time a derivative is computed. If a source transformation AD tool is used, the penalty associated with source transformation (i.e. the file generation) must only be incurred a single time prior to solving the problem. Thus, if first and second derivative source codes can be generated in a reasonable amount of time, it will likely be the case that source transformation is the more favorable approach.

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

the dimensions of `a` and `b` directly within the derivative code, where each branch of the conditional statement contains the proper differentiation rule given the dimension information. In contrast, when using a conventional operator overloaded AD tool, all function sizes/values are known at the time of differentiation, and thus uncertainties such as the meaning of the `*` operator are a non-issue.

As discussed in Section 2.2.2, derivative computation times can be greatly reduced by taking advantage of derivative sparsity. In order to fully exploit sparsity, one would like a method which statically exploits sparsity at the operation level. That being said, the classical approaches to sparsity exploitation are limited by the fact that they may either dynamically exploit sparsity at the operation level (by performing sparse arithmetic at run-time) or statically exploit sparsity of the program as a whole (by utilizing matrix compression techniques). Both approaches may be used with either an operator overloaded tool or a source transformation tool, though an operator overloaded approach is inherently dynamic and thus lends itself well to dynamic sparsity exploitation. That is, one may write the overloaded routines to perform sparse arithmetic and store derivative objects in a row/column/value format. In contrast, it is often difficult for a source transformation tool to extract complex dependency relationships at compile time and thus the resulting derivative code must be made to depend upon an exterior sparse arithmetic library in order to exploit sparsity at each link in the chain rule.

The AD tool developed in this dissertation attempts to capture the strengths of both operator overloading and source transformation while eliminating some of the drawbacks. The method gains robustness from the fact that it simply evaluates a section of code on overloaded objects, yet the result of the overloaded evaluations is a printed procedure which may be evaluated numerically to compute derivatives. Thus, the overhead from the operator overloading must only be incurred a single time for any problem of interest. Additionally, by propagating derivative sparsity patterns within the overloaded objects, sparsity may be exploited at compile-time, for each link in the chain rule. Thus, the

generated derivative code neither relies upon dynamic sparsity propagation, nor matrix compression.

3.2 Sparse Notations

Without loss of generality, consider a vector function of a vector $\mathbf{f}(\mathbf{x})$ where $\mathbf{f} : \mathbb{R}^{n_x} \longrightarrow \mathbb{R}^{m_f}$. The Jacobian of the vector function $\mathbf{f}(\mathbf{x})$, denoted $\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x})$, is then an $m_f \times n_x$ matrix

$$\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_{n_x}} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_{n_x}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{m_f}}{\partial x_1} & \frac{\partial f_{m_f}}{\partial x_2} & \cdots & \frac{\partial f_{m_f}}{\partial x_{n_x}} \end{bmatrix} \in \mathbb{R}^{m_f \times n_x}. \tag{3–1}$$

Assuming the first derivative matrix $\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x})$ 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 $\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x})$ are denoted by the index vector pairs $(\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{f}}) \in \mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f}$, where $\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}$ correspond to the row locations and $\mathbf{j}_{\mathbf{x}}^{\mathbf{f}}$ correspond to the column locations. In order to ensure uniqueness of the row/column pairs $\left(i_{\mathbf{x}}^{\mathbf{f}}(k), j_{\mathbf{x}}^{\mathbf{f}}(k)\right)$, (where $i_{\mathbf{x}}^{\mathbf{f}}(k)$ and $j_{\mathbf{x}}^{\mathbf{f}}(k)$ refer to the k^{th} elements of the vectors $\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}$ and $\mathbf{j}_{\mathbf{x}}^{\mathbf{f}}$, respectively, $k = 1, \ldots, p_x^f$), the following restriction is placed upon the order of the index vectors:

$$i_{\mathbf{x}}^{\mathbf{f}}(1) + n(j_{\mathbf{x}}^{\mathbf{f}}(1) - 1) < i_{\mathbf{x}}^{\mathbf{f}}(2) + n(j_{\mathbf{x}}^{\mathbf{f}}(2) - 1) < \cdots < i_{\mathbf{x}}^{\mathbf{f}}(p_x^f) + n(j_{\mathbf{x}}^{\mathbf{f}}(p_x^f) - 1). \tag{3–2}$$

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

$$d_{\mathbf{x}}^{\mathbf{f}}(k) = \frac{\partial f_{[i_{\mathbf{x}}^{\mathbf{f}}(k)]}}{\partial x_{[j_{\mathbf{x}}^{\mathbf{f}}(k)]}}, \quad (k = 1, \ldots, p_x^f), \tag{3–3}$$

where $d_{\mathbf{x}}^{\mathbf{f}}(k)$ refers to the k^{th} element of the vector $\mathbf{d}_{\mathbf{x}}^{\mathbf{f}}$. Using this sparse notation, the Jacobian $\nabla_{\mathbf{x}}\mathbf{f}(\mathbf{x})$ may be fully defined given the row/column/value triplet $(\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{d}_{\mathbf{x}}^{\mathbf{f}}) \in$

$\mathbb{Z}_+^{p_x^f} \times \mathbb{Z}_+^{p_x^f} \times \mathbb{R}^{p_x^f}$ and the dimension pair (m_f, n_x). This notation may also be extended to matrix functions of matrices by using the unrolled Jacobian form of Eq. (2–1).

In the developed method a large amount of effort is used in order to track possible non-zero locations of vectors (i.e. functions) and matrices (i.e. Jacobians). In order to propagate sparsity patterns it is often the case that operations such as unions and intersections must be performed by operating on index vectors. First, the operator $\mathbf{S}_1 : \mathbb{I}_1 \to 2^{\mathbb{Z}_+}$ is defined to map the set of all ordered positive index vectors

$$\mathbb{I}_1 := \bigcup_{n=1}^{\infty} \left\{ \mathbf{i} \in \mathbb{Z}_+^n \ : \ i_h < i_{h+1}, \ h = 1, \ldots, n-1 \right\} \tag{3–4}$$

into the power set of $\mathbb{Z}_+$, $2^{\mathbb{Z}_+}$, such that, given $\mathbf{i} \in \mathbb{Z}_+^n$, $i_1 < i_2 < \cdots < i_n$,

$$\mathbf{S}_1 [\mathbf{i}] := \{ i_h \ : \ h = 1, \ldots, n \}, \tag{3–5}$$

where, due to the restriction of the domain $\mathbb{I}_1$ in Eq. (3–4), $\mathbf{S}_1 [\cdot]$ has the unique inverse $\mathbf{S}_1^{-1} [\mathbf{S}_1 [\mathbf{i}]] = \mathbf{i}$. Next, the operator $\mathbf{S}_2 : \mathbb{I}_2 \to 2^{\mathbb{Z}_+ \times \mathbb{Z}_+}$ is defined to map the set of all ordered positive index vector pairs

$$\mathbb{I}_2 := \bigcup_{n=1}^{\infty} \left\{ (\mathbf{j}, \mathbf{k}) \in \mathbb{Z}_+^n \times \mathbb{Z}_+^n \ : \ l_h < l_{h+1}, \ l_h = j_h + \max(\mathbf{j})(k_h - 1), h = 1, \ldots, n-1 \right\} \tag{3–6}$$

into the power set of $\mathbb{Z}_+ \times \mathbb{Z}_+$, $2^{\mathbb{Z}_+ \times \mathbb{Z}_+}$, such that, given $\mathbf{j} \in \mathbb{Z}_+^n$, $\mathbf{k} \in \mathbb{Z}_+^n$, $(\mathbf{j}, \mathbf{k}) \in \mathbb{I}_2$,

$$\mathbf{S}_2 [(\mathbf{j}, \mathbf{k})] := \{ (j_h, k_h) \ : \ h = 1, \ldots, n \}, \tag{3–7}$$

where, due to the restriction of the domain $\mathbb{I}_2$ in Eq. (3–6), $\mathbf{S}_2 [\cdot]$ has the unique inverse $\mathbf{S}_2^{-1} [\mathbf{S}_2 [(\mathbf{j}, \mathbf{k})]] = (\mathbf{j}, \mathbf{k})$. Thus, rather than defining operations such as unions and intersections of sparsity patterns, more tractable operations may instead be performed on sets of the form of Eqs. (3–5) and (3–7). For example, the common possible non-zero locations of two Jacobians $\nabla_{\mathbf{x}} \mathbf{g}$ and $\nabla_{\mathbf{x}} \mathbf{f}$ may be defined by the set of integer pairs $\mathbf{S}_2 [(\mathbf{i}_{\mathbf{x}}^{\mathbf{g}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{g}})] \cap \mathbf{S}_2 \left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{f}}) \right]$, where the $(\mathbf{i}_{\mathbf{x}}^{\mathbf{g}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{g}})$ and $(\mathbf{i}_{\mathbf{x}}^{\mathbf{f}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{f}})$ denote the possible non-zero locations of the Jacobians $\nabla_{\mathbf{x}} \mathbf{g}$ and $\nabla_{\mathbf{x}} \mathbf{f}$, respectively.

3.3 The CADA Class

The CADA class is used to print a section of derivative code by evaluating a section of function code on instances of the class. During the evaluation of a section of code on CADA objects, a derivative file is simultaneously generated by storing only sizes, symbolic identifiers, and sparsity patterns within the objects themselves and writing the computational steps required to compute derivatives to a file. In order to exploit derivative sparsity, only the possible non-zero derivatives are propagated within the printed derivative file, while the locations of the possible non-zero derivatives are propagated within the overloaded objects. In order to propagate sparsity patterns in this manner, however, it must be stressed that all function sizes and derivative sparsity patterns must be fixed. As a direct consequence, any derivative procedures generated by overloaded operations are only valid for fixed input function sizes and derivative sparsity patterns. In this section, the properties of CADA objects are given.

Without loss of generality, an overloaded CADA object $\mathcal{Y}$ is considered, where it is assumed that there is a single independent variable of differentiation, $\mathbf{x}$. The properties of such an object are given in Table 3-1. From this table, it is seen that information is stored on both the variable, $\mathbf{y}$, and the derivative, $\nabla_{\mathbf{x}}\mathbf{y}$. Assuming that the overloaded object $\mathcal{Y}$ has been created, then the proper procedures will have been printed to a file to compute both the value of $\mathbf{y}$ and the non-zero derivatives $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$ (if any) of the Jacobian $\nabla_{\mathbf{x}}\mathbf{y}$. In this dissertation, the printed variable which is assigned the value of $\mathbf{y}$ is referred to as a function variable and the printed variable which is assigned the value of $\mathbf{d}_{\mathbf{x}}^{\mathbf{y}}$ is referred to as a derivative variable. As noted in Table 3-1, there is a distinction made between strictly symbolic and not strictly symbolic function variables, where, if a function variable is strictly symbolic, at least one element of the function variable is unknown at the time of code generation. Thus, a function variable which is not strictly symbolic is considered to be a constant with no associated derivative (where constants are propagated within overloaded operations and calculations are still printed to file). Furthermore, it is noted

that the value assigned to any derivative variable may be mapped into a two-dimensional Jacobian using the index vector pairs (e.g. $(\mathbf{i}^{\mathbf{y}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{y}}_{\mathbf{x}})$) stored in the `deriv.nzlocs` field of the corresponding overloaded object.

Table 3-1. Properties of an overloaded object $\mathcal{Y}$. Each overloaded object has the fields `id`, `func` and `deriv`. The `func` field contains information of the function variable which is assigned the value of $\mathbf{y}$ the generated program, and the `deriv` field contains information on the derivative variable which is assigned the value of $\mathbf{d}^{\mathbf{y}}_{\mathbf{x}}$ in the generated program (where $\mathbf{x}$ is the independent variable of differentiation).

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

3.4 Overloaded CADA Operations

The CADA class utilizes forward-mode algorithmic differentiation to propagate derivative sparsity patterns while performing overloaded evaluations on a section of user written code. All standard unary array operations (for example, trigonometric, exponential, etc.), binary array operations (for example, plus, minus, times, etc.), organizational operations (for example, reference, assignment, concatenation, transpose, etc.) and matrix operations (for example, matrix multiplication, matrix division, matrix summations, etc.) are overloaded. These overloaded operations result in the appropriate function and non-zero derivative procedures being printed to a file while

simultaneously determining the associated sparsity pattern of the derivative function. Given the properties of Table 3-1, together with the definitions of function variables and derivative variables, the method of differentiation is then fairly straightforward. Each time an overloaded operation is performed, the input object's properties together with the properties of the corresponding operation must be used to assign the properties of the resulting object. Additionally, the proper procedures must be printed to the derivative file such that, upon numeric evaluation, both the output's function variable and derivative variable will be calculated as a function of the input's function variable and derivative variable.

Due to the nature of the target applications, it is expected that the derivative file will be evaluated many times while the file must only be generated a single time. Thus, when overloaded operations are performed, the efficiency of the printed derivative procedures is of a higher priority than the efficiency of the overloaded operations. Compilation time, however, is also of importance. As discussed in Section 2.3, the determination of how to most efficiently perform MATLAB routines is not always clear. This is particularly the case when sparsity information is known (as is the case in this section) as there is a constant trade-off between performing sparse arithmetic and the number of operations/references/assignments required to perform the sparse arithmetic. The printed procedures which result from the overloaded operations of this section attempt to find the most efficient combination of sparsity exploitation and run-time overhead minimization.

Examples of how CADA operations are used to both propagate derivative sparsity patterns and print efficient derivative calculation procedures are now presented. For all operation types it is assumed that $\mathbf{x} \in \mathbb{R}^{n_x}$ is the variable of differentiation.

3.4.1 Array Operations

The MATLAB language has a class of array operations which perform real valued scalar operations on each element of an input array. For the purposes of this section it is allowed that, if $\mathbf{g}(\cdot)$ is an array operation, then $\mathbf{g}(\cdot)$ performs the operation $g(\cdot)$ on each

element of the input array(s), where $g(\cdot)$ is a scalar real valued function. For instance, if $\mathbf{g}(\mathbf{v})$ is a unary array operation and $\mathbf{v} \in \mathbb{R}^{m_v}$, then

$$\mathbf{g}(\mathbf{v}) = \begin{bmatrix} g(v_1) & g(v_2) & \cdots & g(v_{m_v}) \end{bmatrix}^{\mathsf{T}} \in \mathbb{R}^{m_v}, \tag{3–8}$$

where $g : \mathbb{R} \to \mathbb{R}$. The Jacobian $\nabla_{\mathbf{v}}\mathbf{g}(\mathbf{v})$ is then given by the diagonal matrix

$$\nabla_{\mathbf{v}}\mathbf{g}(\mathbf{v}) = \begin{bmatrix} g_v(v_1) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & g_v(v_2) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & g_v(v_{m_v}) \end{bmatrix} \in \mathbb{R}^{m_v \times m_v}, \tag{3–9}$$

where $g_v(\cdot)$ is the derivative of $g(\cdot)$ with respect to its argument. If $\mathbf{f}(\mathbf{v}, \mathbf{u})$ is a binary array operation (e.g. `plus`, `times`), $(\mathbf{u}, \mathbf{v}) \in \mathbb{R}^{m_v} \times \mathbb{R}^{m_v}$, then

$$\mathbf{f}(\mathbf{u}, \mathbf{v}) = \begin{bmatrix} f(u_1, v_1) & f(u_2, v_2) & \cdots & f(u_{m_v}, v_{m_v}) \end{bmatrix}^{\mathsf{T}} \in \mathbb{R}^{m_v}, \tag{3–10}$$

where $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Moreover, the derivative objects $\nabla_{\mathbf{u}}\mathbf{f}(\mathbf{u}, \mathbf{v})$ and $\nabla_{\mathbf{v}}\mathbf{f}(\mathbf{u}, \mathbf{v})$ are given by the diagonal matrices

$$\nabla_{\mathbf{u}}\mathbf{f}(\mathbf{u}, \mathbf{v}) = \begin{bmatrix} f_u(u_1, v_1) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & f_u(u_2, v_2) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & f_u(u_{m_v}, v_{m_v}) \end{bmatrix} \in \mathbb{R}^{m_v \times m_v}, \tag{3–11}$$

and

$$\nabla_{\mathbf{v}}\mathbf{f}(\mathbf{u}, \mathbf{v}) = \begin{bmatrix} f_v(u_1, v_1) & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & f_v(u_2, v_2) & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & f_v(u_{m_v}, v_{m_v}) \end{bmatrix} \in \mathbb{R}^{m_v \times m_v}, \tag{3–12}$$

where $f_u(\cdot,\cdot)$ is the derivative of $f(\cdot,\cdot)$ with respect to its first argument and $f_v(\cdot,\cdot)$ is the derivative of $f(\cdot,\cdot)$ with respect to its second argument. The overloaded CADA operations corresponding to such array operations are now presented.

3.4.1.1 Unary array operations

The differentiation of unary array operations is fairly straightforward due to the fact that they are only a function of a single input. Without loss of generality, the manner in which all overloaded unary array operations are performed is now demonstrated. First consider a variable $\mathbf{v}(\mathbf{x}) \in \mathbb{R}^{m_v}$ whose Jacobian, $\nabla_{\mathbf{x}}\mathbf{v}$, is defined by the row/column/value triplet $(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{d}_{\mathbf{x}}^{\mathbf{v}}) \in \mathbb{Z}_+^{p_x^v} \times \mathbb{Z}_+^{p_x^v} \times \mathbb{R}^{p_x^v}$, $p_x^v \leq m_v n_x$. Assuming no zero function locations are known in $\mathbf{v}$, the CADA instance, $\mathcal{V}$, would posses the following relevant properties: (i) `func.name`=`'v.f'`; (ii) `func.size`=$(m_v, 1)$; (iii) `deriv.name`=`'v.dx'`; and (iv) `deriv.nzlocs`=$(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})$. It is assumed that, since the object $\mathcal{V}$ has been created, the function variable, `v.f`, and the derivative variable, `v.dx`, have been printed to a file in such a manner that, upon evaluation, `v.f` and `v.dx` will be assigned the numeric values of $\mathbf{v}$ and $\mathbf{d}_{\mathbf{x}}^{\mathbf{v}}$, respectively. Suppose now that the unary array operation

$$\mathbf{w} = \mathbf{g}(\mathbf{v}(\mathbf{x})) \tag{3–13}$$

(as defined by Eq. (3–8)) is encountered during the evaluation of the MATLAB function code (e.g. `w = cos(v)`, `w = sqrt(v)`, etc.). Due to the diagonal structure of the Jacobian $\nabla_{\mathbf{v}}\mathbf{g}(\mathbf{v}(\mathbf{x}))$, it is then the case that $\nabla_{\mathbf{x}}\mathbf{w}(\mathbf{x}) = [\nabla_{\mathbf{v}}\mathbf{g}(\mathbf{v}(\mathbf{x}))]\,[\nabla_{\mathbf{x}}\mathbf{v}(\mathbf{x})]$ contains possible non-zero derivatives in the same entries as the possible non-zeros of $\nabla_{\mathbf{x}}\mathbf{v}(\mathbf{x})$ (i.e. $(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}}) = (\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})$). Additionally, the non-zero derivative values $\mathbf{d}_{\mathbf{x}}^{\mathbf{w}}$ may be calculated as

$$d_{\mathbf{x}}^{\mathbf{w}}(k) = g_v(v_{[i_{\mathbf{x}}^{\mathbf{v}}(k)]})\, d_{\mathbf{x}}^{\mathbf{v}}(k), \quad (k = 1 \ldots p_x^v), \tag{3–14}$$

where $g_v(\cdot)$ is the derivative of $g(\cdot)$ with respect to the argument v. In the file that is being created, the derivative variable, `w.dx`, would be written as follows. Assume that the index $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}$ is printed to a file such that it will be assigned to the variable `i_vx` within the

derivative program. The derivative procedure would then be printed as

$$\texttt{w.dx = g\_v(v.f(i\_vx)).*v.dx}, \tag{3–15}$$

and the function procedure would simply be written as

$$\texttt{w.f = g(v.f)}, \tag{3–16}$$

where `g(·)` represents the MATLAB unary array operation corresponding to $\mathbf{g}(\cdot)$ (e.g. `sin(·)`, `sqrt(·)`, etc.) and `g_v(·)` represents the MATLAB array operation or sequence of array operations corresponding to $\mathbf{g}_v(\cdot)$ (e.g. `cos(·)`, `(1/2)./sqrt(·)`, etc.) The resulting overloaded object, $\mathcal{W}$, would then have the same properties as $\mathcal{V}$ with the exception of `id`, `func.name`, and `deriv.name`. Furthermore, if $g : 0 \to 0$ and $\mathcal{V}$`.func.zerolocs` is not empty, then $\mathcal{W}$`.func.zerolocs` is assigned the values of $\mathcal{V}$`.func.zerolocs`, otherwise it is assumed that there exist no known zero values of the function variable `w.f`.

3.4.1.2 Binary array operations

Without loss of generality the manner in which all overloaded binary array operations are performed is now demonstrated. While it is possible that only one of the two inputs to a binary array operation contains derivative information, the case is trivial given the explanation of Section 3.4.1.1. As such, the case where both inputs contain derivative information is considered. For this demonstration the variable $\mathbf{v}(\mathbf{x})$ and corresponding overloaded object $\mathcal{V}$ of Section 3.4.1.1 are again considered. Additionally, a new variable $\mathbf{u}(\mathbf{x}) \in \mathbb{R}^{m_u}$, $m_u = m_v$, is considered where the Jacobian $\nabla_{\mathbf{x}}\mathbf{u}(\mathbf{x})$ is defined by the row/column/value triplet $(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{d}^{\mathbf{u}}_{\mathbf{x}}) \in \mathbb{Z}_+^{p_x^u} \times \mathbb{Z}_+^{p_x^u} \times \mathbb{R}^{p_x^u}$, $p_x^u \leq m_u n_x$. Assuming no zero function locations are known in $\mathbf{u}$, the CADA instance, $\mathcal{U}$, would posses the following relevant properties: (i) `func.name`=`'u.f'`; (ii) `func.size`=$(m_u, 1)$; (iii) `deriv.name`=`'u.dx'`; and (iv) `deriv.nzlocs`=$(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}})$. It is again assumed that since the object $\mathcal{U}$ has been created, the function variable, `u.f`, and the derivative variable,

`u.dx`, have been printed to a file in such a manner that, upon evaluation, `u.f` and `u.dx` will be assigned the numeric values of $\mathbf{u}$ and $\mathbf{d}_{\mathbf{x}}^{\mathbf{u}}$, respectively. It is now assumed that the binary array operation

$$\mathbf{w} = \mathbf{f}(\mathbf{u}(\mathbf{x}), \mathbf{v}(\mathbf{x})) \tag{3–17}$$

(as defined by Eq. (3–10)) is encountered (e.g. `+`, `.*`, `./`, etc.). Given the diagonal structure of the derivative matrices of Eqs.(3–11) and (3–12), the possible non-zero entries of $\nabla_{\mathbf{x}}\mathbf{w}(\mathbf{x})$ are then defined by the index vector pairs

$$(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}}) = \mathbf{S}_2^{-1}\left[\mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{u}})\right] \cup \mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})\right]\right] \in \mathbb{Z}_+^{p_x^w} \times \mathbb{Z}_+^{p_x^w}, \tag{3–18}$$

where $\max(p_x^u, p_x^v) \leq p_x^w \leq \max(p_x^u + p_x^v, m_u n_x)$. Thus the relevant properties of the object $\mathcal{W}$ would be built as (i) `func.name`=`'w.f'`; (ii) `func.size`=$(m_v, 1)$; (iii) `deriv.name`=`'w.dx'`; and (iv) `deriv.nzlocs`=$(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}})$.[1] The possible non-zero entries of the derivative matrices $[\nabla_{\mathbf{u}}\mathbf{f}(\mathbf{u}, \mathbf{v})]\,[\nabla_{\mathbf{x}}\mathbf{u}]$ and $[\nabla_{\mathbf{v}}\mathbf{f}(\mathbf{u}, \mathbf{v})]\,[\nabla_{\mathbf{x}}\mathbf{v}]$ are now denoted by ${}^{\mathbf{u}}\mathbf{d}_{\mathbf{x}}^{\mathbf{w}} \in \mathbb{R}^{p_x^u}$ and ${}^{\mathbf{v}}\mathbf{d}_{\mathbf{x}}^{\mathbf{w}} \in \mathbb{R}^{p_x^v}$. Without loss of generality, these entries may then be calculated by

$${}^{\mathbf{u}}d_{\mathbf{x}}^{\mathbf{w}}(k) = f_u(u_{[i_{\mathbf{x}}^{\mathbf{u}}(k)]}, v_{[i_{\mathbf{x}}^{\mathbf{u}}(k)]})d_{\mathbf{x}}^{\mathbf{u}}(k), \quad (k = 1, \ldots, p_x^u), \tag{3–19}$$

and

$${}^{\mathbf{v}}d_{\mathbf{x}}^{\mathbf{w}}(k) = f_v(u_{[i_{\mathbf{x}}^{\mathbf{v}}(k)]}, v_{[i_{\mathbf{x}}^{\mathbf{v}}(k)]})d_{\mathbf{x}}^{\mathbf{v}}(k), \quad (k = 1, \ldots, p_x^v), \tag{3–20}$$

[1] Similar to the unary case, if any function values of $\mathbf{u}$ and/or $\mathbf{v}$ are known to be zero these are used in order to determine if any entries of $\mathbf{w}$ are also known to be zero. Additionally, assuming that $u_i = 0$ and/or $v_i = 0$ implies that $\nabla_{\mathbf{x}} w_i = \mathbf{0}$, known zero entries of $\mathbf{u}$ and/or $\mathbf{v}$ are used to cancel elements of $\mathbf{d}_{\mathbf{x}}^{\mathbf{w}}$. Performing such cancellations is imperative in maintaining sparsity, however these cases are not presented for the sake of conciseness.

respectively, where $f_u(u,v)$ and $f_v(u,v)$ are the derivatives of $f(u,v)$ with respect the arguments u and v, respectively.[2] In order to sparsely complete the chain rule, the following three mutually exclusive sets are found

$$\begin{aligned}\mathscr{A} &= \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}})\right] \setminus \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}})\right] \\ \mathscr{B} &= \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}})\right] \setminus \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}})\right] \\ \mathscr{C} &= \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}})\right] \cap \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{v}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{v}}_{\mathbf{x}})\right]\end{aligned},$$

where $\mathscr{A} \cup \mathscr{B} \cup \mathscr{C} = \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{w}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{w}}_{\mathbf{x}})\right]$. Assuming the sets $\mathscr{A}$, $\mathscr{B}$, and $\mathscr{C}$ are not empty, the possible non-zero values of $\nabla_{\mathbf{x}}\mathbf{w}$ are then given by

$$d^{\mathbf{w}}_{\mathbf{x}}(k) = {}^{\mathbf{u}}d^{\mathbf{w}}_{\mathbf{x}}(r), \quad \forall\big(k,r\big) \in \Big\{\big(k,r\big) : \big(i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(k)\big) = (i^{\mathbf{u}}_{\mathbf{x}}(r), j^{\mathbf{u}}_{\mathbf{x}}(r)),\ (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(k)) \in \mathscr{A}\Big\}, \tag{3–21}$$

$$d^{\mathbf{w}}_{\mathbf{x}}(k) = {}^{\mathbf{v}}d^{\mathbf{w}}_{\mathbf{x}}(s), \quad \forall(a,s) \in \{(k,s) : (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(a)) = (i^{\mathbf{v}}_{\mathbf{x}}(s), j^{\mathbf{v}}_{\mathbf{x}}(s)),\ (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(a)) \in \mathscr{B}\}, \tag{3–22}$$

and

$$\begin{aligned} d^{\mathbf{w}}_{\mathbf{x}}(k) = {}^{\mathbf{u}}d^{\mathbf{w}}_{\mathbf{x}}(r) + {}^{\mathbf{v}}d^{\mathbf{w}}_{\mathbf{x}}(s), & \\ \forall(k,r,s) \in \{(k,r,s) : (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(k)) = (i^{\mathbf{v}}_{\mathbf{x}}(r), j^{\mathbf{v}}_{\mathbf{x}}(r)) = (i^{\mathbf{v}}_{\mathbf{x}}(s), j^{\mathbf{v}}_{\mathbf{x}}(s)), & \\ (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(k)) \in \mathscr{C}\}. & \end{aligned} \tag{3–23}$$

This procedure, however, is somewhat complex as multiple sets of reference and assignment indices must be found. Moreover, as discussed in Section 2.3, references and assignments via index vectors can be expensive. Thus, in order to reduce the required number of references and assignments, the two assignment index vectors

$${}^{\mathbf{u}}\mathbf{k}^{\mathbf{w}}_{\mathbf{x}} = \mathbf{S}_2^{-1}\left[\{k : (i^{\mathbf{w}}_{\mathbf{x}}(k), j^{\mathbf{w}}_{\mathbf{x}}(k)) \in \mathbf{S}_2\left[(\mathbf{i}^{\mathbf{u}}_{\mathbf{x}}, \mathbf{j}^{\mathbf{u}}_{\mathbf{x}})\right]\}\right] \in \mathbb{Z}_+^{p^u_x} \tag{3–24}$$

[2] The functions f_u and f_v are typically not a function of both u and v (and neither u nor v in the cases of $+$ and $-$) and this is always checked in order to generate efficient code, however the more general case has been presented.

and

$$ {}^{\mathbf{v}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}} = \mathbf{S}_2^{-1}\left[\{k : (i_{\mathbf{x}}^{\mathbf{w}}(k), j_{\mathbf{x}}^{\mathbf{w}}(k)) \in \mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})\right]\}\right] \in \mathbb{Z}_+^{p_x^v} \tag{3–25} $$

are instead determined. Utilizing these assignment index vectors, the derivative procedures required to compute `w.dx` would be written to the derivative file as follows. First the function reference indices $\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}$ and $\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}$ would be written to variables `i_ux` and `i_vx` and the assignment indices ${}^{\mathbf{u}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}}$ and ${}^{\mathbf{v}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}}$ would be written to variables `k_uwx` and `k_vwx`. Allowing `f_u(·,·)` and `f_v(·,·)` to represent the MATLAB array operation or sequence of array operations corresponding to $\mathbf{f}_u(\cdot,\cdot)$ and $\mathbf{f}_v(\cdot,\cdot)$, respectively, the procedures corresponding to Eqs. (3–19)–(3–23) would then compactly be printed as

```
w.dx         = zeros(p_wx,1),
f_utmp       = f_u(u.f,v.f);
w.dx(k_uwx)  = f_utmp(i_ux).*u.dx,                    (3–26)
f_vtmp       = f_v(u.f,v.f);
w.dx(k_vwx)  = w.dx(k_vwx) + f_vtmp(i_vx).*v.dx,
```

where it is noted that the temporary variables `f_utmp` and `f_vtmp` are used in order to reduce the number of required index references. The function procedure would simply be written as

```
w.f = f(u.f,v.f).                                     (3–27)
```

where `(·,·)` represents the MATLAB binary array operation corresponding to $\mathbf{f}(\cdot,\cdot)$.

3.4.2 Organizational Operations

The MATLAB language has a wide variety of organizational operations ranging from references and assignments to concatenations and reshapes, all of which may be written as one or more references and/or assignments. For instance, `transpose`, `reshape` and `repmat` may all be replaced by a single call to `subsref` (the MATLAB subscript array reference function), and `horzcat` and `vertcat` may be replaced by multiple calls `subsasgn` (the MATLAB subscript array assignment function). Thus, in this section the overloaded

CADA operations corresponding to both array subscript references and assignments are presented. All other organizational operation rules may be directly inferred from these two operations.

3.4.2.1 Subscript array reference operation

In this section the reference operation $\mathbf{r} : \mathbb{R}^{m_u} \times \mathbb{Z}_+^{m_w} \to \mathbb{R}^{m_w}$, is considered such that, if $\mathbf{w} = \mathbf{r}(\mathbf{u}(\mathbf{x}), \mathbf{k})$, then

$$w_h = u_{[k(h)]}, \quad h = 1, \ldots, m_w \tag{3–28}$$

where, for the sake of mathematical convenience,the index vector $\mathbf{k}$ is restricted to be strictly increasing. (In MATLAB this would correspond to the operation `w = u(k)`). It is now assumed that $(\mathbf{i}_\mathbf{x}^\mathbf{u}, \mathbf{j}_\mathbf{x}^\mathbf{u}, \mathbf{d}_\mathbf{x}^\mathbf{u}) \in \mathbb{Z}_+^{p_x^u} \times \mathbb{Z}_+^{p_x^u} \times \mathbb{R}^{p_x^u}$ are the row/column/value triplets of $\nabla_\mathbf{x}\mathbf{u}$, thus, the row/column index pair of $\nabla_\mathbf{x}\mathbf{w}$ is given by

$$(\mathbf{i}_\mathbf{x}^\mathbf{w}, \mathbf{j}_\mathbf{x}^\mathbf{w}) = \mathbf{S}_2^{-1}\left[\{(i,j) : (k(i), j) \in \mathbf{S}_2\left[(\mathbf{i}_\mathbf{x}^\mathbf{u}, \mathbf{j}_\mathbf{x}^\mathbf{u})\right]\}\right] \in \mathbb{Z}_+^{p_x^w} \times \mathbb{Z}_+^{p_x^w}. \tag{3–29}$$

In order to print the derivative rule, one must determine the proper elements of $\mathbf{d}_\mathbf{x}^\mathbf{u}$ to reference. Due to the strictly increasing restriction placed upon $\mathbf{k}$, these indices are given by

$$\mathbf{k}_\mathbf{x}^\mathbf{w} = \mathbf{S}_1^{-1}\left[\{h : (i_\mathbf{x}^\mathbf{u}(k(h)), j_\mathbf{x}^\mathbf{u}(h)) \in \mathbf{S}_2\left[(\mathbf{i}_\mathbf{x}^\mathbf{w}, \mathbf{j}_\mathbf{x}^\mathbf{x})\right]\}\right] \in \mathbb{Z}^{p_x^w}, \tag{3–30}$$

where

$$d_\mathbf{x}^\mathbf{w}(h) = d_\mathbf{x}^\mathbf{u}(k_\mathbf{x}^\mathbf{w}(h)), \quad h = 1, \ldots, p_x^w. \tag{3–31}$$

Thus, if the overloaded operation $\mathcal{W} = \mathcal{R}(\mathcal{U}, \mathbf{k})$ were encountered, the values of $(\mathbf{i}_\mathbf{x}^\mathbf{w}, \mathbf{j}_\mathbf{x}^\mathbf{w}, \mathbf{k}_\mathbf{x}^\mathbf{w})$ would be determined as previously described and the properties of $\mathcal{W}$ would be built accordingly. In the printed derivative file, the index vectors $\mathbf{k}_\mathbf{x}^\mathbf{w}$ and $\mathbf{k}$ would be written to the variables `k_wx` and `k`, respectively. The derivative procedure would then be printed as

```
w.dx = u.dx(k_wx)
```  (3–32)

and the function procedure would be printed as

$$\texttt{w.f = u.f(k).} \tag{3–33}$$

3.4.2.2 Subscript array assignment operation

In this section an assignment operation is considered such that all elements of the vector $\mathbf{b}(\mathbf{x}) \in \mathbb{R}^{m_b}$ are assigned to the $\mathbf{k} \in \mathbb{Z}_+^{m_b}$ entries of a vector $\mathbf{u}(\mathbf{x}) \in \mathbb{R}^{m_u}$, where $m_u > m_b$ and the index vector $\mathbf{k}$ contains m_b unique elements. This operation will be denoted by $\mathbf{w} = \mathbf{a}(\mathbf{u}(\mathbf{x}), \mathbf{b}(\mathbf{x}), \mathbf{k})$ which corresponds to the MATLAB `subsasgn` operation `u(k) = b` followed by the assignment `w = u;`. For mathematical convenience, it is also assumed that the index vector $\mathbf{k}$ is strictly increasing. Given this definition, all elements of the variable $\mathbf{w}$ are then given by

$$\begin{array}{rcll} w_{[k(h)]} & = & b_h, & h = 1, \ldots, m_b, \\ w_l & = & u_l, & \forall\ l \in \{l : l \leq m_u, l \notin \mathbf{S}_1[\mathbf{k}]\} \end{array} . \tag{3–34}$$

It is now assumed that $(\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{d}_{\mathbf{x}}^{\mathbf{u}}) \in \mathbb{Z}_+^{p_x^u} \times \mathbb{Z}_+^{p_x^u} \times \mathbb{R}^{p_x^u}$ and $(\mathbf{i}_{\mathbf{x}}^{\mathbf{b}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{b}}, \mathbf{d}_{\mathbf{x}}^{\mathbf{b}}) \in \mathbb{Z}_+^{p_x^b} \times \mathbb{Z}_+^{p_x^b} \times \mathbb{R}^{p_x^b}$ are the row/column/value triplets of $\nabla_{\mathbf{x}}\mathbf{u}$ and $\nabla_{\mathbf{x}}\mathbf{b}$, respectively. In order to identify the possible non-zero derivative locations of $\nabla_{\mathbf{x}}\mathbf{w}$, an index vector pair $({}^{\mathbf{b}}\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, {}^{\mathbf{b}}\mathbf{j}^{\mathbf{w}}) \in \mathbb{Z}_+^{p_x^b} \times \mathbb{Z}_+^{p_x^b}$ may be found such that

$$({}^{\mathbf{b}}i_{\mathbf{x}}^{\mathbf{w}}(h), {}^{\mathbf{b}}j_{\mathbf{x}}^{\mathbf{w}}(h)) = (k(i_{\mathbf{x}}^{\mathbf{b}}(h)), j_{\mathbf{x}}^{\mathbf{b}}(h)), \quad h = 1, \ldots, p_x^b. \tag{3–35}$$

Similarly, it is allowed that

$$({}^{\mathbf{u}}\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, {}^{\mathbf{u}}\mathbf{j}_{\mathbf{x}}^{\mathbf{w}}) = \mathbf{S}_2^{-1}\left[\mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{u}})\right] \setminus \{(i,j) : i \in \mathbf{S}_1[\mathbf{k}]\}\right] \tag{3–36}$$

and thus the possible non-zero derivative locations of $\nabla_{\mathbf{x}}\mathbf{w}$ are given by

$$(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}}) = \mathbf{S}_2^{-1}\left[\mathbf{S}_2\left[({}^{\mathbf{b}}\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, {}^{\mathbf{b}}\mathbf{j}_{\mathbf{x}}^{\mathbf{w}})\right] \cup \mathbf{S}_2\left[({}^{\mathbf{u}}\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, {}^{\mathbf{u}}\mathbf{j}^{\mathbf{w}})\right]\right]. \tag{3–37}$$

In order to concisely build the derivative vector $\mathbf{d}_{\mathbf{x}}^{\mathbf{w}} \in \mathbb{R}^{p_x^w}$ it is then required to determine the three index vectors:

$$\begin{aligned} {}^{\mathbf{b}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}} &= \mathbf{S}_1^{-1}\left[\left\{h : ({}^{\mathbf{b}}i_{\mathbf{x}}^{\mathbf{w}}(h), {}^{\mathbf{b}}j_{\mathbf{x}}^{\mathbf{w}}(h)) \in \mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}})\right]\right\}\right] &&\in \mathbb{Z}_+^{p_x^b} \times \mathbb{Z}_+^{p_x^b} \\ {}^{\mathbf{u}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}} &= \mathbf{S}_1^{-1}\left[\{h : ({}^{\mathbf{u}}i_{\mathbf{x}}^{\mathbf{w}}(h), {}^{\mathbf{u}}j_{\mathbf{x}}^{\mathbf{w}}(h)) \in \mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}})\right]\}\right] &&\in \mathbb{Z}_+^{p_x^w - p_x^b} \times \mathbb{Z}_+^{p_x^w - p_x^b} \quad , \\ {}^{\mathbf{u}}\mathbf{r}_{\mathbf{x}}^{\mathbf{w}} &= \mathbf{S}_1^{-1}\left[\{h : (i_{\mathbf{x}}^{\mathbf{u}}(h), j_{\mathbf{x}}^{\mathbf{u}}(h)) \in \mathbf{S}_2\left[({}^{\mathbf{u}}\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, {}^{\mathbf{u}}\mathbf{j}_{\mathbf{x}}^{\mathbf{w}})\right]\}\right] &&\in \mathbb{Z}_+^{p_x^w - p_x^b} \times \mathbb{Z}_+^{p_x^w - p_x^b} \end{aligned} \tag{3–38}$$

where it is noted that $\mathbf{S}_1\left[{}^{\mathbf{b}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}}\right] \cup \mathbf{S}_1\left[{}^{\mathbf{u}}\mathbf{k}_{\mathbf{x}}^{\mathbf{w}}\right] = \emptyset$. The possible non-zero values of $\nabla_{\mathbf{x}}\mathbf{w}$ are then given by

$$d_{\mathbf{x}}^{\mathbf{w}}({}^{\mathbf{b}}k_{\mathbf{x}}^{\mathbf{w}}(h)) = d_{\mathbf{x}}^{\mathbf{b}}(h), \quad h = 1, \ldots, p_x^b, \tag{3–39}$$

and

$$d_{\mathbf{x}}^{\mathbf{w}}({}^{\mathbf{u}}k_{\mathbf{x}}^{\mathbf{w}}(h)) = d_{\mathbf{x}}^{\mathbf{u}}({}^{\mathbf{u}}r_{\mathbf{x}}^{\mathbf{w}}(h)), \quad h = 1, \ldots, p_x^w - p_x^b. \tag{3–40}$$

Thus, if the overloaded operation $\mathcal{W} = \mathcal{A}(\mathcal{U}, \mathcal{B}, \mathbf{k})$ were encountered, the index vector pair $(\mathbf{i}_{\mathbf{x}}^{\mathbf{w}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{w}})$ would be computed via Eq. (3–37) and the properties of $\mathcal{W}$ would be built accordingly. Additionally, the index vectors of Eq. (3–38) would be computed and written to the variables `k_bwx`, `k_uwx`, and `r_uwx`, respectively. The derivative procedure would then be printed as

```
w.dx         = zeros(p_wx);
w.dx(k_bwx)  = b.dx;
w.dx(k_uwx)  = u.dx(r_uwx);
```
(3–41)

3.4.3 Matrix Operations

In this section a distinction must be made between array operations and matrix operations. While the array operations of Section 3.4.1 execute element by element operations on input arrays, the matrix operations discussed in this section follow the rules of linear algebra. Of consideration are the operations of matrix multiplication, matrix summation, matrix inversion, and matrix division, where it is noted that the derivative rule for all such operations may be written as a sequence of addition and matrix

multiplications. For instance, the derivative of the operation $\mathbf{A}(\mathbf{x})\mathbf{B}(\mathbf{x})$ is defined by

$$\frac{\partial \mathbf{A}(\mathbf{x})\mathbf{B}(\mathbf{x})}{\partial x_k} = \frac{\partial \mathbf{A}(\mathbf{x})}{\partial x_k}\mathbf{B}(\mathbf{x}) + \mathbf{A}(\mathbf{x})\frac{\partial \mathbf{B}(\mathbf{x})}{\partial x_k}, \quad k = 1, \ldots, n_x, \tag{3–42}$$

the derivative of the operation $\mathbf{A}^{-1}(\mathbf{x})$ (assuming $\mathbf{A}(\mathbf{x})$ is square) is defined by

$$\frac{\partial \mathbf{A}^{-1}(\mathbf{x})}{\partial x_k} = \mathbf{A}^{-1}(\mathbf{x})\frac{\partial \mathbf{A}(\mathbf{x})}{\partial x_k}\mathbf{A}^{-1}(\mathbf{x}), \quad k = 1, \ldots, n_x, \tag{3–43}$$

etc. In order to explore the methods used to handle such operations, the case of a non-constant matrix being pre-multiplied by a constant matrix is now explored. The manner in which all other matrix operations are carried out may then be inferred given the following explanations together with the transpose property, $(\mathbf{AB})^\mathsf{T} = \mathbf{B}^\mathsf{T}\mathbf{A}^\mathsf{T}$ and the discussion of Section 3.4.1.2.

Consider a matrix variable $\mathbf{A}(\mathbf{x}) \in \mathbb{R}^{q_a \times r_a}$ whose unrolled Jacobian, $\nabla_\mathbf{x}\mathbf{A}^\dagger(\mathbf{x})$, is defined by the row/column/value triplet $(\mathbf{i}^\mathbf{a}_\mathbf{x}, \mathbf{j}^\mathbf{a}_\mathbf{x}, \mathbf{d}^\mathbf{a}_\mathbf{x}) \in \mathbb{Z}_+^{p_x^a} \times \mathbb{Z}_+^{p_x^a} \times \mathbb{R}^{p_x^a}$, and a constant matrix variable $\mathbf{B} \in \mathbb{R}^{q_c \times q_a}$, where the elements $\bar{\mathbf{k}}^\mathbf{b} \in \mathbb{Z}_+^{\bar{p}^b}$ of $\mathbf{B}^\dagger$ are known to be zero. Given this information, the relevant properties of the corresponding overloaded objects are given in Table 3-2. The the matrix operation $\mathbf{C}(\mathbf{x}) = \mathbf{B}\mathbf{A}(\mathbf{x})$, $\mathbf{B} \in \mathbb{R}^{q_c \times q_a}$, is

Table 3-2. Relevant properties of $\mathcal{A}$ and $\mathcal{B}$.

	`func.name`	`func.size`	`func.zerolocs`	`deriv.name`	`deriv.nzlocs`
$\mathcal{A}$:	`'A.f'`	(q_a, r_a)	-	`'A.dx'`	$(\mathbf{i}^\mathbf{a}_\mathbf{x}, \mathbf{j}^\mathbf{a}_\mathbf{x})$
$\mathcal{B}$:	`'B.f'`	(q_c, q_a)	$\bar{\mathbf{k}}^\mathbf{b}$	-	-

now considered. It is then the goal of the corresponding overloaded operation to first determine the possible non-zero derivative locations of the unrolled Jacobian $\nabla_\mathbf{x}\mathbf{C}^\dagger(\mathbf{x})$ and to concisely print efficient calculations to a MATLAB file which compute the non-zero derivatives of $\nabla_\mathbf{x}\mathbf{C}^\dagger(\mathbf{x})$ as a function of $\mathbf{d}^\mathbf{a}_\mathbf{x}$ and $\mathbf{B}$. In is now noted that the derivative of $\mathbf{C}(\mathbf{x})$ is given by the chain rule

$$\nabla_{x_k}\mathbf{C}(\mathbf{x}) = \mathbf{B}\nabla_{x_k}\mathbf{A}(\mathbf{x}), \quad k = 1, \ldots, n_x. \tag{3–44}$$

Moreover, if the matrix

$$\mathbf{D}(\mathbf{x}) = \left[\begin{array}{cccc} \nabla_{x_1}\mathbf{A}(\mathbf{x}) & \nabla_{x_2}\mathbf{A}(\mathbf{x}) & \cdots & \nabla_{x_{n_x}}\mathbf{A}(\mathbf{x}) \end{array} \right] \in \mathbb{R}^{q_a \times r_a n_x} \tag{3–45}$$

is built, then the n_k derivative matrices of Eq. (3–44) may be obtained by the pre-multiplication of $\mathbf{D}(\mathbf{x})$ by $\mathbf{B}$ and collected in the matrix $\mathbf{E}(\mathbf{x})$ such that

$$\begin{array}{rcl} \mathbf{E}(\mathbf{x}) & = & \mathbf{B}\mathbf{D}(\mathbf{x}) \\ & = & \left[\begin{array}{ccc} \mathbf{A}\nabla_{x_1}\mathbf{B}(\mathbf{x}) & \cdots & \mathbf{A}\nabla_{x_{n_x}}\mathbf{B}(\mathbf{x}) \end{array} \right] \\ & = & \left[\begin{array}{ccc} \nabla_{x_1}\mathbf{C}(\mathbf{x}) & \cdots & \nabla_{x_{n_x}}\mathbf{C}(\mathbf{x}) \end{array} \right] \in \mathbb{R}^{q_c \times r_c n_x}. \end{array} \tag{3–46}$$

Thus, one may obtain the locations of the possible non-zeros of $\nabla_{\mathbf{x}}\mathbf{C}^{\dagger}(\mathbf{x})$ as follows. First, the non-zeros of $\mathbf{D}(\mathbf{x})$ may be identified by the row/column/value triplet $(\mathbf{i}^{\mathbf{d}}, \mathbf{j}^{\mathbf{d}}, \mathbf{d}^{\mathbf{a}}_{\mathbf{x}}) \in \mathbb{Z}_+^{p_x^a} \times \mathbb{Z}_+^{p_x^a} \times \mathbb{R}^{p_x^a}$ where $(\mathbf{i}^{\mathbf{d}}, \mathbf{j}^{\mathbf{d}})$ may be obtained via the affine transformations

$$\begin{array}{rcl} i^{\mathbf{d}}(h) & = & i^{\mathbf{a}}_{\mathbf{x}}(h) - q_a \lfloor \frac{i^{\mathbf{a}}_{\mathbf{x}}(h)-1}{q_a} \rfloor, \\ j^{\mathbf{d}}(h) & = & r_a \left(j^{\mathbf{a}}_{\mathbf{x}}(h) - 1 \right) + \lfloor \frac{i^{\mathbf{a}}_{\mathbf{x}}(h)-1}{q_a} \rfloor + 1, \end{array} \qquad h = 1, \ldots, p_x^a. \tag{3–47}$$

Similarly, the row/column locations of the possible non-zero elements of the matrix $\mathbf{B}$, $(\mathbf{i}^{\mathbf{b}}, \mathbf{j}^{\mathbf{b}}) \in \mathbb{Z}_+^{p^a} \times \mathbb{Z}_+^{p^a}$, $(p^a = q_c q_a - \bar{p}^a)$ are given by the transformations

$$\begin{array}{rcl} i^{\mathbf{b}}(h) & = & k^{\mathbf{a}}(h) - q_c \lfloor \frac{k^{\mathbf{a}}(h)-1}{q_c} \rfloor \\ j^{\mathbf{b}}(h) & = & \lfloor \frac{k^{\mathbf{a}}(h)-1}{q_c} \rfloor + 1 \end{array}, h = 1, \ldots, p_a, \tag{3–48}$$

where

$$\mathbf{k}^{\mathbf{a}} = \mathbf{S}_1^{-1} \left[\{1, \ldots, q_c q_a\} \setminus \mathbf{S}_1 \left[\bar{\mathbf{k}}^{\mathbf{b}} \right] \right] \in \mathbb{Z}_+^{p_a}. \tag{3–49}$$

The possible non-zero elements of the matrix $\mathbf{E}(\mathbf{x})$ may then be identified by the index vector pair

$$(\mathbf{i}^{\mathbf{e}}, \mathbf{j}^{\mathbf{e}}) = \mathbf{S}_2^{-1} \left[\left\{ (i,j) : \left\{ k : (i,k) \in \mathbf{S}_2 \left[(\mathbf{i}^{\mathbf{b}}, \mathbf{j}^{\mathbf{b}}) \right] \right\} \cap \left\{ k : (k,j) \in \mathbf{S}_2 \left[(\mathbf{i}^{\mathbf{d}}, \mathbf{j}^{\mathbf{d}}) \right] \right\} \neq \emptyset \right\} \right], \tag{3–50}$$

where $(\mathbf{i^e}, \mathbf{j^e}) \in \mathbb{Z}_+^{p_x^c} \times \mathbb{Z}_+^{p_x^c}$, $p_x^c \leq q_c r_c n_x$. Finally, the locations of the possible non-zero elements of $\nabla_{\mathbf{x}} \mathbf{C}^\dagger(\mathbf{x})$, $(\mathbf{i_x^c}, \mathbf{j_x^c}) \in \mathbb{Z}_+^{p_x^c} \times \mathbb{Z}_+^{p_x^c}$, are obtained via the affine transformations

$$\begin{aligned} i_{\mathbf{x}}^{\mathbf{c}}(h) &= q_c \left(j^{\mathbf{e}}(h) - r_c \lfloor \tfrac{j^{\mathbf{e}}(h)-1}{r_c} \rfloor - 1 \right) + i^{\mathbf{e}}(h), \\ j_{\mathbf{x}}^{\mathbf{c}}(h) &= \lfloor \tfrac{j^{\mathbf{e}}(h)-1}{r_c} \rfloor + 1, \end{aligned} \qquad h = 1, \ldots, p_x^c. \tag{3–51}$$

Now, given that an expression for the non-zero locations of $\nabla_{\mathbf{x}} \mathbf{C}^\dagger(\mathbf{x})$ has been derived, it is then required to print the procedures which compute the non-zero derivatives $\mathbf{d_x^c} \in \mathbb{R}_+^{p_x^c}$ as a function of $\mathbf{d_x^b}$ and $\mathbf{A}$. This procedure amounts to performing the matrix multiplication of Eq. (3–46). Now, knowing the sparsity patterns of both $\mathbf{B}$ and $\mathbf{D}$, it is possible to print a procedure which only operates on the non-zero elements of $\mathbf{A}$ and $\mathbf{D}$ to compute the non-zero elements of $\mathbf{E}$. That being said, such a procedure would require many operations and many references/assignments, both of which result in run-time overhead penalties. Thus, unlike the previously presented overloaded operations, the overloaded matrix operations write derivative rules in terms of matrix multiplications. This does not mean, however, that sparsity may not be exploited. Another way in which sparsity could be exploited is by utilizing matrix compression techniques presented in Section 2.2.2. That is, knowing $(\mathbf{i^e}, \mathbf{j^e})$, one could determine a seed matrix $\mathbf{S}$ such that $\mathbf{E}$ is fully recoverable from the result of $\mathbf{ES} = \mathbf{ADS}$. Thus, assuming $\mathbf{S}$ is constructed via a Curtis-Powell-Reid approach [16], a matrix $\bar{\mathbf{D}} = \mathbf{DS}$ may be constructed directly from the non-zero derivatives of $\mathbf{A}$, $\mathbf{d_x^a}$. The computation $\bar{\mathbf{E}} = \mathbf{B}\bar{\mathbf{D}}$ could then be printed and the non-zero derivatives of $\mathbf{C}$ extracted directly from $\bar{\mathbf{E}}$. While such an approach could greatly reduce the number of columns in the printed matrix multiplication, the computation of such a seed matrix is relatively expensive, so much so that the computation of such a seed matrix would dominate compilation times.

As an alternative to the expensive seed matrix computation, sparsity is used to simply eliminate known-zero columns of the $\mathbf{D}$ matrix of Eq. (3–45). In order to do so, the index

$\bar{\mathbf{k}}^{\mathbf{d}} \in \mathbb{Z}_+^{\bar{r}_d}$ is used to identify the columns of $\mathbf{D}$ which are possibly non-zero,

$$\bar{\mathbf{k}}^{\mathbf{d}} = \left\{ j : (i,j) \in \mathbf{S}_2 \left[(\mathbf{i}^{\mathbf{d}}, \mathbf{j}^{\mathbf{d}}) \right] \right\} \in \mathbb{Z}_+^{\bar{r}_d}. \tag{3–52}$$

Allowing $\bar{\mathbf{D}} \in \mathbb{R}^{q_a \times \bar{r}_d}$ to be a matrix consisting of the non-zero columns of $\mathbf{D}$, it is then desired to print the procedures which project the values of $\mathbf{d}_{\mathbf{x}}^{\mathbf{a}}$ into the proper elements of the matrix $\bar{\mathbf{D}}$, perform the matrix multiplication $\bar{\mathbf{E}} = \mathbf{A}\bar{\mathbf{D}}$, and extract the values of $\mathbf{d}_{\mathbf{x}}^{\mathbf{c}}$ from the proper elements of $\bar{\mathbf{E}}$. To this end, the index vector pair $(\bar{\mathbf{i}}^{\mathbf{d}}, \bar{\mathbf{j}}^{\mathbf{d}}) \in \mathbb{Z}_+^{p_x^a} \times \mathbb{Z}_+^{p_x^a}$ is found such that

$$(\bar{\mathbf{i}}^{\mathbf{d}}, \bar{\mathbf{j}}^{\mathbf{d}}) = \mathbf{S}_2^{-1} \left[\left\{ (i,j) : (i, \bar{k}^{\mathbf{d}}(j)) \in \mathbf{S}_2 \left[(\mathbf{i}^{\mathbf{d}}, \mathbf{j}^{\mathbf{d}}) \right] \right\} \right]. \tag{3–53}$$

Next, the non-zero locations of the matrix $\bar{\mathbf{E}}$ are likewise determined such that

$$(\bar{\mathbf{i}}^{\mathbf{e}}, \bar{\mathbf{j}}^{\mathbf{e}}) = \mathbf{S}_2^{-1} \left[\left\{ (i,j) : (i, \bar{k}^{\mathbf{d}}(j)) \in \mathbf{S}_2 \left[(\mathbf{i}^{\mathbf{e}}, \mathbf{j}^{\mathbf{e}}) \right] \right\} \right] \in \mathbb{Z}_+^{p_x^c} \times \mathbb{Z}_+^{p_x^c}, \tag{3–54}$$

with linear mapping index $\bar{\mathbf{h}}^{\mathbf{e}} \in \mathbb{Z}_+^{p_x^c}$,

$$\bar{h}^{\mathbf{e}}(l) = (\bar{j}^{\mathbf{e}}(l) - 1)\, q_c + \bar{i}^{\mathbf{e}}(l), \quad l = 1, \ldots, p_x^c. \tag{3–55}$$

Finally, a check is determined as to whether $\bar{\mathbf{D}}$ should be built as a `sparse` MATLAB object or not, where the check is based upon the sparsity of the $\bar{\mathbf{D}}$ matrix. Assuming that it is determined the $\bar{\mathbf{D}}$ matrix should be built as a `sparse` object[3] , the derivative procedure would then be printed as

```
Dbar  =  sparse(ibar_d,jbar_d,a.dx,q_a,rbar_d);
Ebar  =  A*Dbar;
c.dx  =  full(Ebar(hbar_e));
```

[3] Under the condition that $\mathbf{D}$ is either a small or dense matrix, it is not built as an object of the `sparse` class. This case is not, however, presented.

where the values of $\bar{\mathbf{i}}^{\mathbf{d}}$, $\bar{\mathbf{j}}^{\mathbf{d}}$, and $\bar{\mathbf{h}}^{\mathbf{e}}$ would be written to the variables `ibar_d`, `jbar_d`, and `hbar_e`, respectively. Thus, a small amount of work is performed in order to eliminate redundant operations on known-zero columns, but the MATLAB `sparse` library is utilized in order to carry out the matrix multiplication.

3.5 Storage of Indices Used in Generated Code

As may be seen from Section 3.4, the derivative procedures printed by overloaded operations can be highly dependent upon reference and assignment index vectors being printed to variables in the 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 strings which build the index vectors to the file (i.e. `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.

3.6 Vectorization of the CADA Class

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

$$\mathbf{F}(\mathbf{X}) = \begin{bmatrix} \mathbf{f}(\mathbf{X}_1) & \mathbf{f}(\mathbf{X}_2) & \cdots & \mathbf{f}(\mathbf{X}_N) \end{bmatrix} \in \mathbb{R}^{n \times N}, \tag{3–56}$$

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

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 & \mathbf{X}_2 & \cdots & \mathbf{X}_N \end{bmatrix} \in \mathbb{R}^{n_x \times N}. \tag{3–57}$$

Such functions are similar to the array operations of Section 3.4.1, however, 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 diagonal Jacobian structure due to the fact that

$$\frac{\partial F_{l,i}(\mathbf{X})}{\partial X_{j,k}} = 0, \quad \forall i \neq k,\ l = 1, \ldots, m_f,\ j = 1, \ldots, n_x. \tag{3–58}$$

Thus $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger(\mathbf{X}^\dagger)$ is given by the block diagonal matrix

$$\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger(\mathbf{X}^\dagger) = \begin{bmatrix} \nabla_{\mathbf{X}_1}\mathbf{F}_1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \nabla_{\mathbf{X}_2}\mathbf{F}_2 & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \nabla_{\mathbf{X}_N}\mathbf{F}_N \end{bmatrix} \in \mathbb{R}^{m_f N \times n_x N}, \tag{3–59}$$

where

$$\nabla_{\mathbf{X}_i}\mathbf{F}_i = \begin{bmatrix} \frac{\partial F_{1,i}}{\partial X_{1,i}} & \frac{\partial F_{1,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{1,i}}{\partial X_{n_x,i}} \\ \frac{\partial F_{2,i}}{\partial X_{1,i}} & \frac{\partial F_{2,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{2,i}}{\partial X_{n_x,i}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_{m_f,i}}{\partial X_{1,i}} & \frac{\partial F_{m_f,i}}{\partial X_{2,i}} & \cdots & \frac{\partial F_{m_f,i}}{\partial X_{n_x,i}} \end{bmatrix} \in \mathbb{R}^{m_f \times n_x}, \quad i = 1, \ldots, N. \tag{3–60}$$

Such functions commonly occur when utilizing collocation methods [1] 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 $\mathbf{X} \in \mathbb{R}^{n_x \times N}$ and $\mathbf{s} \in \mathbb{R}^{n_s}$ which solve the equation

$$\mathbf{c}(\mathbf{F}(\mathbf{X}), \mathbf{X}, \mathbf{s}) = \mathbf{0} \in \mathbb{R}^{m_c}, \tag{3–61}$$

where $\mathbf{F}(\mathbf{X})$ is of the form of Eq. (3–56). Now, one could apply AD directly to Eq. (3–61), however, it is often the case that it is more efficient to instead apply AD to the function $\mathbf{F}(\mathbf{X})$, where the specific structure of Eq. (3–59) may be exploited. The results may then be used to compute the derivatives of Eq. (3–61). It is of the concern of this section to efficiently use the CADA class to compute the derivative of vectorized functions of the form of Eq. (3–56), where the aforementioned vectoried function has been coded as a basic block in MATLAB.

Due to the block diagonal structure of Eq. (3–59), it is the case that the vectorized problem has an inherently compressible Jacobian of column dimension n_x. This compression may be performed via the pre-defined seed matrix

$$\mathbf{S} = \begin{bmatrix} \mathbf{I}_{n_x} \\ \mathbf{I}_{n_x} \\ \vdots \\ \mathbf{I}_{n_x} \end{bmatrix} \in \mathbb{R}^{n_x N \times n_x}, \tag{3–62}$$

where $\mathbf{I}_{n_x}$ is the $n_x \times n_x$ identity matrix. The method of this section, does not, however, utilize matrix compression, but rather utilizes the fact that the structure of the Jacobian of Eq. (3–59) is determined by the structure of the Jacobian of Eq. (3–60). To exhibit this point, the row/column pairs of the derivative of $\mathbf{f}$ 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, $\nabla_{\mathbf{X}_i}\mathbf{F}_i$, 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 the derivative matrix $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$ are then given by

$$\mathbf{D}^{\mathbf{F}}_{\mathbf{X}} = \begin{bmatrix} \mathbf{d}^{\mathbf{F}_1}_{\mathbf{X}_1} & \mathbf{d}^{\mathbf{F}_2}_{\mathbf{X}_2} & \cdots & \mathbf{d}^{\mathbf{F}_N}_{\mathbf{X}_N} \end{bmatrix} \in \mathbb{R}^{p_x^f \times N}. \tag{3–63}$$

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

In order to notify the overloaded operations that vectorized derivative procedures are to be printed, all vectorized CADA instances are made to have a value of ∞ located in the size field corresponding to the vectorized dimension. Thus, if an object $\mathcal{X}$ was instantiated to correspond to a matrix of size $n_x \times N$, then the $\mathcal{X}$`.func.size` property would contain the value (n_x, ∞), thus notifying any operations performed on $\mathcal{X}$ that

the second dimension is vectorized. Moreover, all overloaded operations of Section 3.4 are peformed in a relatively similar manner; where the only difference is that they print procedures to compute vectorized non-zero derivative matrices of the form of $\mathbf{D}_{\mathbf{X}}^{\mathbf{F}} \in \mathbb{R}^{p_x^f \times N}$, rather than non-zero derivative vectors of the form of $\mathbf{d}_{\mathbf{x}}^{\mathbf{f}} \in \mathbb{R}^{p_x^f}$. For instance, if $\mathcal{V}$ of Section 3.4.1.1 was made vectorized such that $\mathcal{U}$.`func.size` $= (m_u, \infty)$, then the printed procedure of Eq. (3–15) would be replaced by

```
w.dx = g_v(v.f(i_vx,:)).*v.dx,                                    (3–64)
```

and the function procedure would stay the same. Similarly, if $\mathcal{U}$ and $\mathcal{V}$ of Section 3.4.1.2 were made vectorized such that their size fields were (m_u, ∞), then the printed procedure of Eq. (3–26) would be replaced by

```
w.dx           =  zeros(p_wx,size(u.f,2)),
f_utmp         =  f_u(u.f,v.f);
w.dx(a_uwx,:)  =  f_utmp(i_ux,:).*u.dx,                           (3–65)
f_vtmp         =  f_v(u.f,v.f);
w.dx(a_vwx,:)  =  w.dx(a_vwx,:)  + f_vtmp(i_vx,:).*v.dx,
```

and again the function calculation would remain the same. Furthermore, in both cases the propagation of row/column index vector pairs is identical to the non-vectorized case.

Here is is noted that, given a fixed value of N, the methods previously described in Section 3.4 could easily be used to print the procedures required to compute the non-zero derivatives of the Jacobian $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$. Moreover, the underlying mathematical operations of the derivative procedures printed using the methods of either Sections 3.4 or 3.6 would be nearly identical. The advantages of differentiating vectorized functions using the methods of this section over those previously described in Section 3.4 are now discussed:

1. Derivative Files are Vectorized: Typically functions of the form of Eq. (3–56) are coded such that the value of N may be any positive integer. By utilizing the CADA class in the manner presented in this section, it is the case that the derivative files are generated such that N may be any positive integer. In contrast, any

files generated via the methods of Section 3.4 are only valid for fixed input sizes. The allowance of the dimension N to change is particularly helpful when using collocation methods with a process known as mesh refinement [7]. In such instances, the problem of Eq. (3–61) must often be re-solved with different values of N.

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

3. Reduction in Run-Time Overhead: As has been seen in this chapter, the CADA class relies heavily on reference and assignment indexes in order to print sparse derivative procedures. As was discussed in Section 2.3, such references and assignments come at a large run-time overhead cost. By printing derivative procedures in the manner proposed in this section, however, such run-time overheads are effectively reduced by an order of N. This run-time overhead reduction is achieved by storing non-zero derivatives as matrices of the form of Eq. (3–63) in the generated derivative files. Thus, when performing references/assignments such as those of Eqs. (3–64) and (3–65), the ':' character may be used. The dimensions of the required reference/assignment indices are thus reduced by an order of N.

CHAPTER 4
SOURCE TRANSFORMATION VIA OPERATOR OVERLOADING FOR AD IN MATLAB

Chapter 3 identifies the fact that the overloaded CADA class may be used to transform a basic block of function code into a basic block of derivative code. Moreover, the generated derivative code may be evaluated on numeric function and derivative input values in order to sparsely compute the non-zero derivatives of the outputs of the basic blocks. The method may also be applied to programs containing unrollable loops, where, if evaluated on CADA objects, the resulting derivative code would contain an unrolled representation of the loop. Function programs containing indeterminable conditional statements (that is, conditional statements which are dependent upon strictly symbolic objects), however, cannot be evaluated on instances of the CADA class as multiple branches may be possible. This chapter is focused on developing the methods to allow for the application of the CADA class to differentiate function programs containing indeterminable conditional statements and loops, where the flow control of the originating program is preserved in the derivative program. For the sake of conciseness the discussion of this chapter is restricted to vector functions of vectors. The methods of this chapter, however, are directly applicable to matrix functions of matrices and to the vectorized functions presented in Section 3.6.

4.1 Overmaps and Unions

Before proceeding to the description of the method, it is important to describe an overmap and a union, both which are integral parts of the method itself. Specifically, due to the nature of conditional blocks and loop statements, it is often the case that different assignment objects may be written to the same variable, where the determination of which object is assigned depends upon which conditional branch is taken or which iteration of the loop is being evaluated. The issue is that any overloaded operation performed on such a variable may print different derivative calculations depending upon which object is assigned to the variable. In order to print calculations that are valid for all objects which

may be assigned to the variable (that is, all of the variable's immediate predecessors), a CADA overmap is assigned to the variable, where an overmap has the following properties:

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

Furthermore, the overmap is defined as the union of all possible objects that may be assigned to the variable. Without loss of generality, the concept of a union of two overloaded objects is presented in Table 4-1, where $\mathcal{U}$ and $\mathcal{V}$ represent different overloaded objects which may be assigned to the same variable and $\mathcal{W} = \mathcal{U} \cup \mathcal{V}$ is the union of the objects $\mathcal{U}$ and $\mathcal{V}$.

Table 4-1. Overloaded union example: $\mathcal{W} = \mathcal{U} \cup \mathcal{V}$.

Property	$\mathcal{U}$	$\mathcal{V}$	$\mathcal{W} = \mathcal{U} \cup \mathcal{V}$
`func.size:`	$(m_u, 1)$	$(m_v, 1)$	$(\max(m_u, m_v), 1)$
`func.zerolocs:`	$\bar{\mathbf{i}}^{\mathbf{u}}$	$\bar{\mathbf{i}}^{\mathbf{v}}$	$\mathbf{S}_1^{-1}\left[\mathbf{S}_1\left[\bar{\mathbf{i}}^{\mathbf{u}}\right] \cap \mathbf{S}_2\left[\bar{\mathbf{i}}^{\mathbf{v}}\right]\right]$
`deriv.nzlocs:`	$(\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{u}})$	$(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})$	$\mathbf{S}_2^{-1}\left[\mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{u}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{u}})\right] \cup \mathbf{S}_2\left[(\mathbf{i}_{\mathbf{x}}^{\mathbf{v}}, \mathbf{j}_{\mathbf{x}}^{\mathbf{v}})\right]\right]$

4.2 Source Transformation via the Overloaded CADA Class

A new method is now described for generating derivative files of mathematical functions implemented in MATLAB, where the function source code may contain flow control statements. Source transformation on such function programs is performed using the overloaded CADA class together with unions and overmaps as described in Section 4.1. That is, all function/derivative computations are printed to the derivative file as described in Chapter 3 while flow control is preserved by performing overloaded unions where code fragments join, e.g. on the output of conditional fragments, on the input of loops, etc. The method thus has the feature that the resulting derivative code depends solely on the functions from the native MATLAB library (that is, derivative source code generated by the method does not depend upon overloaded statements or separate run-time functions). Furthermore, the structure of the flow control statements is transcribed to the derivative

source code. In this section, the various processes that are used to carry out the source transformation are described in detail. An outline of the source transformation method is shown in Fig. 4-1.

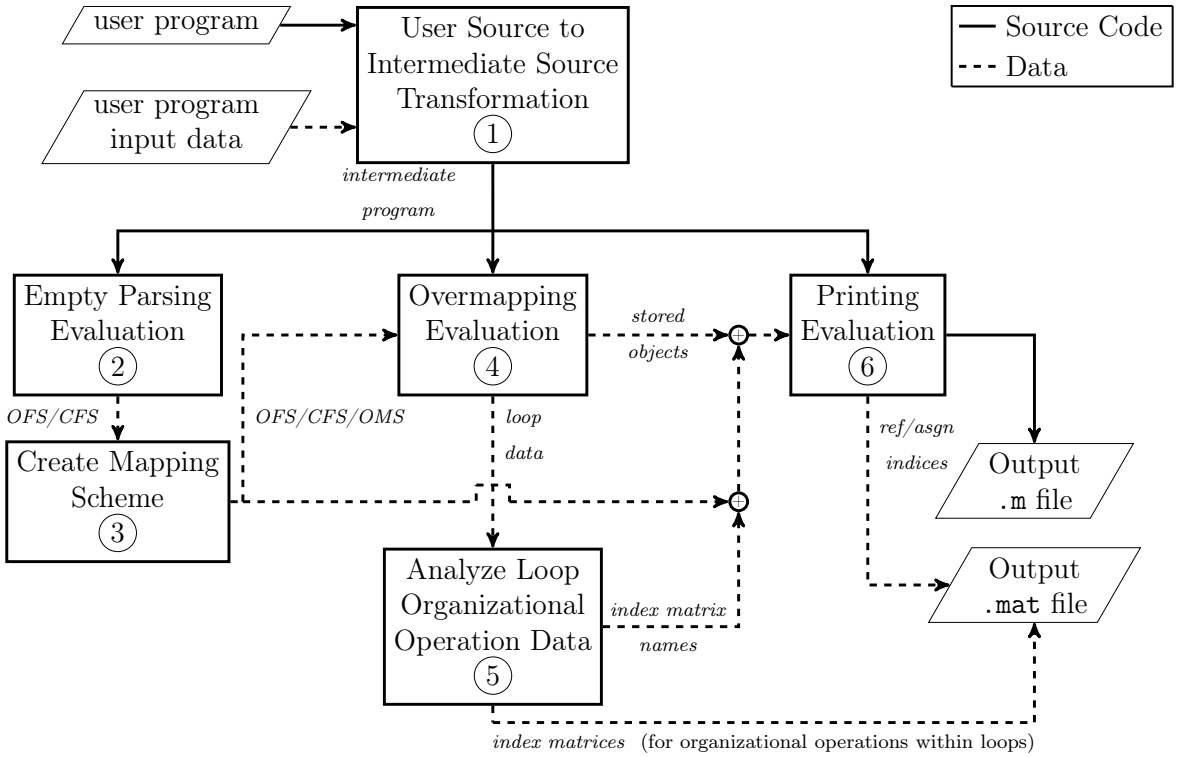


Figure 4-1. Source transformation via operator overloading process.

From Fig. 4-1 it is seen that the inputs to the transformation process are the user program to be differentiated together with the information required to create CADA instances of the inputs to the user program. Using Fig. 4-1 as a guide, the source transformation process starts by performing a transformation of the original source code to an intermediate source code (process ①). This initial transformation then results in a source code on which an overloaded analysis may be performed. Algorithmic differentiation is then effected by performing three overloaded evaluations of the intermediate source code. During the first evaluation (process ②), a record of all objects and locations relative to flow control statements is built to form an object flow structure (OFS) and a control flow structure (CFS), but no derivative calculations are

performed. Next, an object mapping scheme (OMS) is created (process ③) using the DFS and CFS obtained from the first evaluation, where the OMS defines where overloaded unions must be performed and where overloaded objects must be saved. During the second evaluation (process ④), the intermediate source code is evaluated on CADA instances, where each overloaded CADA operation does not print any calculations to a file. During this second evaluation, overmaps are built and are stored in global memory (shown as stored objects output of process ④) while data is collected regarding any organizational operations contained within loops (shown as loop data output of process ④). The organizational operation data is then analyzed to produce special derivative mapping rules for each operation (process ⑤). During the third evaluation of the intermediate source program (process ⑥), all of the data produced from processes ②–⑤ is used to print the final derivative program to a file. In addition, a MATLAB binary file is written that contains the reference and assignment indices required for use in the derivative source code. The details of the steps required to transform a function program containing no function/sub-function calls into a derivative program are given in Sections 4.2.1–4.2.6 and correspond to the processes ①–⑥, respectively, shown in Fig. 4-1. Section 4.2.7 then shows how the methods developed in Sections 4.2.1–4.2.6 are applied to programs containing multiple function calls.

4.2.1 User Source-to-Intermediate-Source Transformation

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

`end`). The code is then read again, line by line, and the intermediate program is printed by copying sections of the user code and applying different augmentations at the statement and flow control levels.

Figure 4-2 shows the transformation of a basic block in the original program, A, into the basic block of the intermediate program, A'. At the basic block level, it is seen that each user assignment is copied exactly from the user program to the intermediate program, but is followed by a call to the transformation routine `VarAnalyzer`. It is also seen that, after any object is written to a variable, the variable analyzer is provided the assignment object, the string of code whose evaluation resulted in the assignment object, the name of the variable to which the object was written, and a flag stating whether or not the assignment was an array subscript assignment. After each assignment object is sent to the `VarAnalyzer`, the corresponding variable is immediately rewritten on the output of the `VarAnalyzer` routine. By sending all assigned variables to the variable analyzer it is possible to distinguish between assignment objects and intermediate objects and determine the names of the variables to which each assignment object is written. Additionally, by rewriting the variables on the output of the variable analyzer, full control over the overloaded workspace (i.e. the collection of all variables active in the intermediate program) is given to the source transformation algorithm. Consequently, all variables (and any operations performed on them) are forced to be overloaded.

Figure 4-3 shows the transformation of the k^{th} conditional fragment encountered in the original program to a transformed conditional fragment in the intermediate program. In the original conditional fragment, C_k, each branch contains the code fragment $B_{k,i}$ ($i = 1, \ldots, N_k$), where the determination of which branch is evaluated depends upon the logical values of the statements `s1,s2,`$\cdots$`,sNk-1`. In the transformed conditional fragment, it is seen that no flow control surrounds any of the branches. Instead, all conditional variables are evaluated and sent to the `IfInitialize` transformation routine. Then the transformed branch fragments $B'_{k,1}, \cdots, B'_{k,N_k}$ are evaluated in a linear manner,

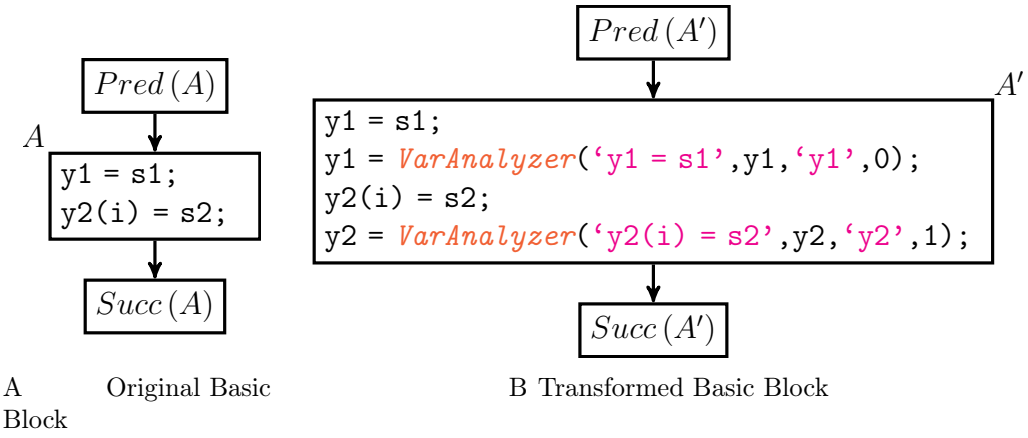
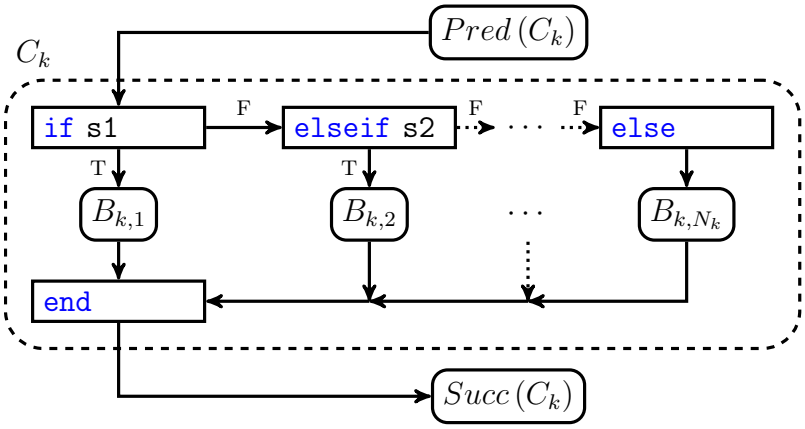


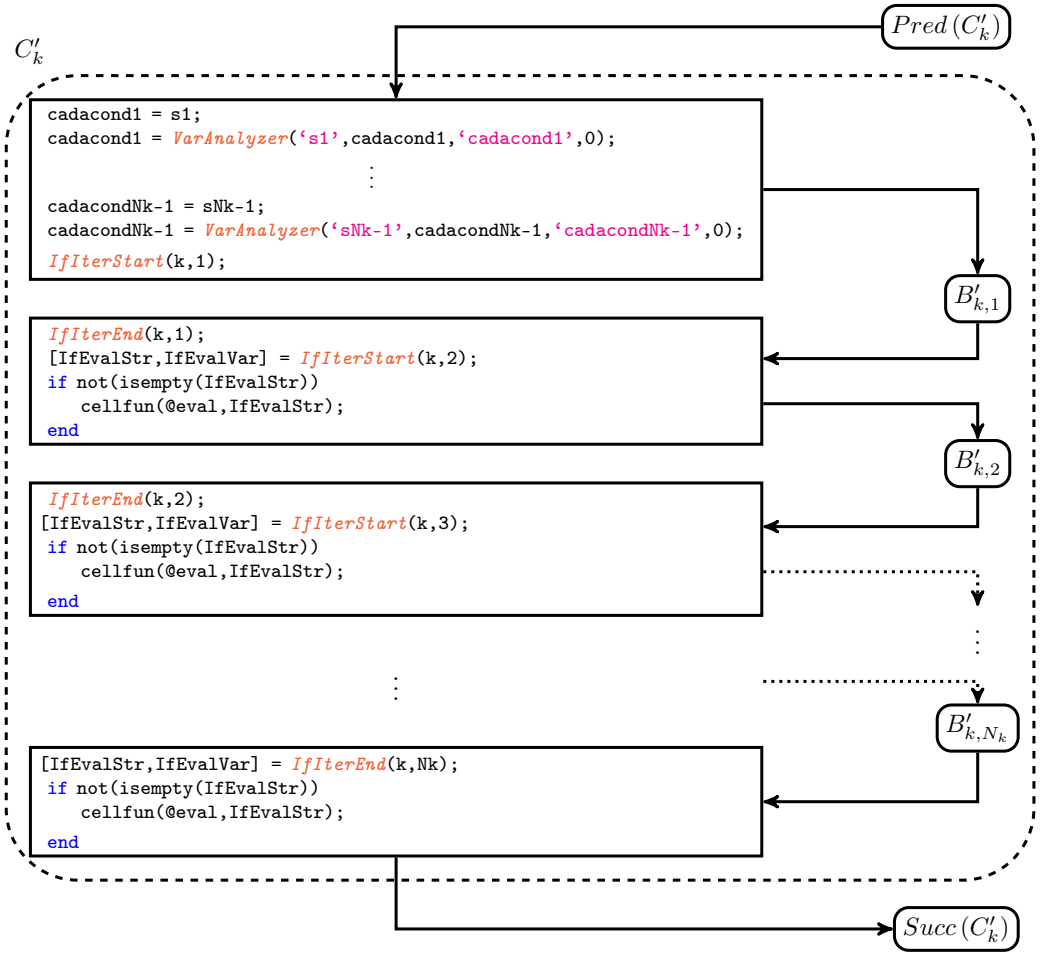
Figure 4-2. Transformation of user source basic block A to intermediate source basic block A'. The quantities `s1` and `s2` represent generic MATLAB expressions.

with a call to the transformation routines `IfIterStart` and `IfIterEnd` before and after the evaluation of each branch fragment. By replacing all conditional statements with transformation routines, the control of the flow is given to the transformation routines. When the intermediate program is evaluated, the `IfInitialize` routine can determine whether each conditional variable returns true, false, or indeterminate, and then the `IfIterStart` routine can set different global flags prior to the evaluation of each branch in order to emulate the conditional statements. As the overloaded analysis of each branch $B'_{k,i}$ is performed in a linear manner, it is important to ensure that each branch is analyzed independent of the others. Thus, prior to any `elseif/else` branch the overloaded workspace may be modified using the outputs, `IfEvalStr` and `IfEvalVar`, of the `IfIterStart` transformation routine. Furthermore, following the overloaded evaluation and analysis of the conditional fragment C'_k, it is often the case that overmapped outputs must be brought into the overloaded workspace. Thus, the final `IfIterEnd` routine has the ability to modify the workspace via its outputs, `IfEvalStr` and `IfEvalVar`.

Figure 4-4 shows an original loop fragment, L_k, and the corresponding transformed loop fragment L'_k, where k is the k^{th} loop encountered in the original program. Similar to the approach used for conditional statements, Fig. 4-4 shows that control over the flow of the loop is given to the transformation routines. Transferring control to the transformation



A Original Conditional Fragment



B Transformed Conditional Fragment

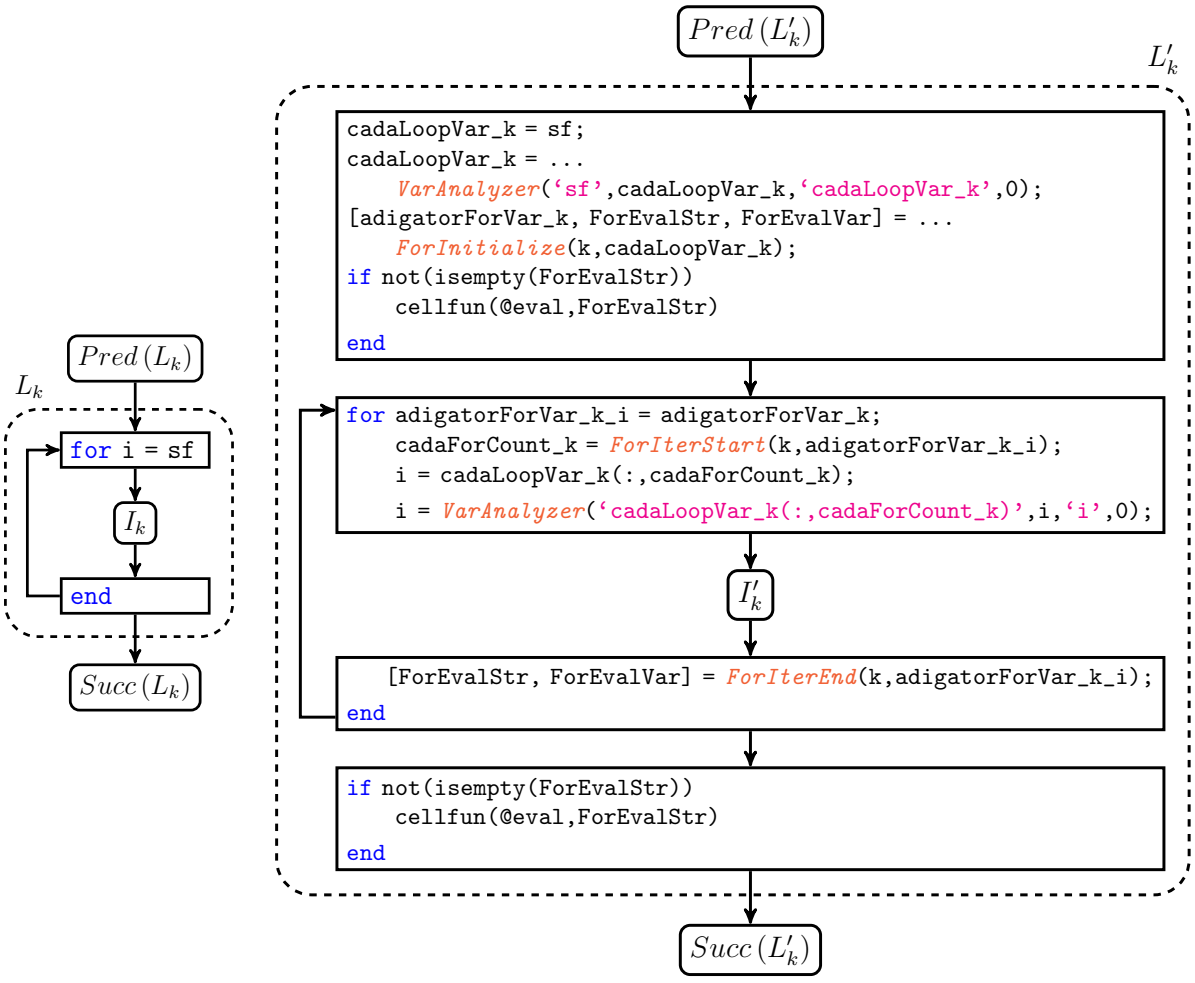
Figure 4-3. Transformation of user source conditional fragment C_k to intermediate source conditional fragment C'_k. Here, C_k is the k^{th} conditional fragment encountered in the user program, $B_{k,i}$ $(i = 1, \ldots, N_k)$ are the fragments of code contained within each branch of the conditional fragment C_k, and the texts `s1`, `s2`, `sNk-1` represent MATLAB logical expressions.

routines is achieved by first evaluating the loop index expression and feeding the result to the `ForInitialize` routine. The loop is then run on the output, `adigatorForVar_k`, of the `ForInitialize` routine, with the loop variable being calculated as a reference on each iteration. Thus the `ForInitialize` routine has the ability to unroll the loop for the purposes of analysis, or to evaluate the loop for only a single iteration. Furthermore, the source transformation algorithm is given the ability to modify the inputs and outputs of the loop. This modification is achieved via the outputs, `ForEvalStr` and `ForEvalVar`, of the transformation routines *`ForInitialize`* and `ForIterEnd`.

It is important to note that the code fragments $B_{k,i}$ of Fig. 4-3 and I_k of Fig. 4-4 do not necessarily represent basic blocks, but may themselves contain conditional fragments and/or loop fragments. In order to account for nested flow control, the user source to intermediate source transformation is performed in a recursive process. Consequently, if the fragments $B_{k,i}$ and/or I_k contain loop/conditional fragments, then the transformed fragments $B'_{k,i}$ and/or I'_k are made to contain transformed loop/conditional fragments.

4.2.2 Parsing of the Intermediate Program

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



A Original Loop Fragment

B Transformed Loop Fragment

Figure 4-4. Transformation of user source loop fragment L_k to intermediate source loop fragment L'_k. The quantity L_k refers to the k^{th} loop fragment encountered in the user program, I_k is the fragment of code contained within the loop, and the text `sf` denotes an arbitrary loop index expression.

`OBJECTCOUNT` is incremented. As control flow has been removed from the intermediate program, there exists only a single path which it may take and thus if an object takes on the `id` field equal to i on one evaluation of the intermediate program, then that object will take on the `id` field equal to i on any evaluation of the intermediate program. Furthermore, if a loop is to be evaluated for multiple iterations, then the value of `OBJECTCOUNT` prior to the first iteration is saved, and `OBJECTCOUNT` is set to the saved value prior to the evaluation of any iteration of the loop. By resetting `OBJECTCOUNT` at the

start of each loop iteration, it is ensured that the `id` assigned to all objects within a loop are iteration independent.

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

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

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

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

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

4.2.3 Creating an Object Overmapping Scheme

Using the OFS and CFS obtained from the empty evaluation, it is then required that an object mapping scheme (OMS) be built, where the OMS will be used in both the overmapping and printing evaluations. The OMS is used to tell the various transformation routines when an overloaded union must be performed to build an overmap, where the

overmapped object is being stored, and when the overmapped object must be written to a variable in the overloaded workspace. The determination of where unions must be performed is based off of where, in the original user code, the program joins. That is, unions must be performed at the exit of conditional fragments and at the entrance of loops. Similarly, the OMS must also tell the transformation routines when an object must be saved, to where it will be saved, and when the saved object must be written to a variable in the overloaded workspace. The manner in which the OMS must be built in order to deal with both conditional branches and loop statements is now investigated.

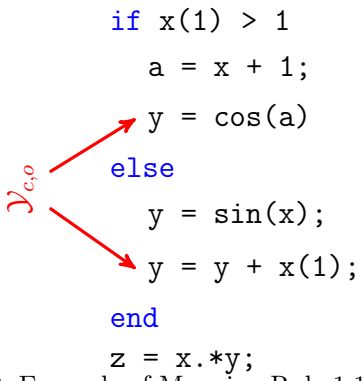
4.2.3.1 Conditional statement mapping scheme

It is required to print conditional fragments to the derivative program such that different branches may be taken depending upon numerical input values. The difficulty that arises using the CADA class is that, given a conditional fragment, different branches can write different assignment objects to the same output variable, and each of these assignment objects can contain differing function and/or derivative properties. To ensure that any operations performed on these variables after the conditional block are valid for any of the conditional branches, a conditional overmap must be assigned to all variables defined within a conditional fragment and used later in the program (that is, the outputs of the conditional fragment). Given a conditional fragment C'_k containing N_k branches, where each branch contains the fragment $B'_{k,i}$ (as shown in Fig. 4-3), the following rules are defined:

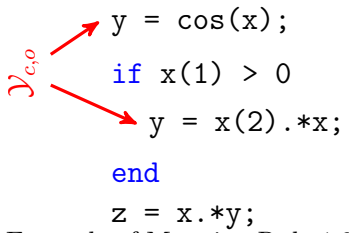
Mapping Rule 1 If a variable y is written within C'_k and read within $Succ\,(C'_k)$, then a single conditional overmap, $\mathcal{Y}_{c,o}$, must be created and assigned to the variable y prior to the execution of $Succ\,(C'_k)$.

Mapping Rule 1.1 For each branch fragment $B'_{k,i}$ $(i = 1, \ldots, N_k)$ within which y is written, the last object assigned to y within $B'_{k,i}$ must belong to $\mathcal{Y}_{c,o}$. See Fig. 4-5A for illustrative example.

Mapping Rule 1.2 If there exists a branch fragment $B'_{k,i}$ within which `y` is not written, or there exists no `else` branch, then the last object which is written to `y` within $Pred\,(C'_k)$ must belong to $\mathcal{Y}_{c,o}$. See Fig. 4-5B for illustrative example.



A Example of Mapping Rule 1.1.



B Example of Mapping Rule 1.2.

Figure 4-5. Illustrative examples of Mapping Rule 1. A) The marked variables belong to the conditional overmap $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.1. Since the unmarked variables are not outputs of the conditional fragment, they do not belong to a conditional overmap. B) The marked variables belong to the conditional overmap $\mathcal{Y}_{c,o}$ as a result of Mapping Rule 1.2.

Using Mapping Rule 1, it is determined for each conditional fragment which variables require a conditional overmap and which assignment objects belong to which conditional overmaps. By adhering to Mapping Rule 1, it is the case that each conditional fragment is analyzed independently of the others. For instance, if a program contains two successive conditional fragments C_1 and C_2, $C_2 \not\subset C_1$, containing N_1 and N_2 branches, respectively, then the program contains N_1N_2 possible branches (assuming both C_1 and C_2 contain `else` branches). Rather than analyzing all N_1N_2 possible branches, the proposed mapping rule states to first analyze the N_1 branches of C_1 and to then use the overmapped outputs to analyze the N_2 branches of C_2. Similarly, if a program contains a nested conditional fragment $C_2 \subset C_1$, then the inner fragment C_2 is first analyzed and the overmapped outputs are used for the analysis of the outer fragment C_1.

The second issue with evaluating transformed conditional fragments of the form shown in Fig. 4-3 stems from the fact that flow control is removed in the intermediate

program. Thus, each branch is analyzed via overloaded evaluation in a successive order. To properly emulate the flow control and force each transformed conditional branch to be analyzed independently, the following mapping rule is defined for each transformed conditional fragment C'_k:

Mapping Rule 2 If a variable `y` is written within $Pred\,(C'_k)$, rewritten within a branch fragment $B'_{k,i}$ and read within a branch fragment $B'_{k,j}$ $(i < j)$, then the last object written to `y` within $Pred\,(C'_k)$ must be saved. Furthermore, the saved object must be assigned to the variable `y` prior to the evaluation of the fragment $B'_{k,j}$. See Fig. 4-6 for illustrative example.

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

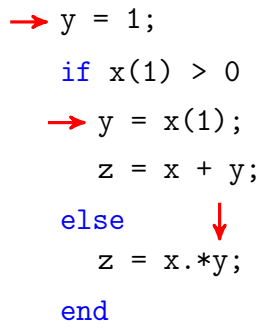


Figure 4-6. Illustrative example of Mapping Rule 2. The variable `y` is an input to the `else` branch, but rewritten in the `if` branch. Since both branches are analyzed successively, the input version of `y` (`y = 1`) must be saved prior to the overloaded evaluation of the `if` branch and brought back into the overloaded workspace prior to the overloaded evaluation of the `else` branch.

4.2.3.2 Loop mapping scheme

In order to print derivative calculations within loops in the derivative program, a single loop iteration must be evaluated on a set of overloaded inputs to the loop, where the input function sizes and/or derivative sparsity patterns may change with each loop iteration. In order to print calculations which are valid for all possible loop inputs, a set of loop overmapped inputs are found by analyzing all possible loop iterations (i.e. unrolling

for the purpose of analysis) in the overmapping evaluation phase. The loop may then be printed as a rolled loop to the derivative program by evaluating on the set of overmapped loop inputs. The first mapping rule is now given for a transformed loop L'_k of the form shown in Fig. 4-4.

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

Mapping Rule 3.1 If a variable `y` is written within $Pred\,(L'_k)$, read within I'_k, and then rewritten within I'_k (that is, `y` is an iteration dependent input to I'_k), then the last objects written to `y` within $Pred\,(L'_k)$ and I'_k must share the same loop overmap. Furthermore, during the printing evaluation, the loop overmap must be written to the variable `y` prior to the evaluation of I'_k. See Fig. 4-7A for illustrative example.

Mapping Rule 3.2 Any assignment object which results from an array subscript assignment belongs to the same loop overmap as the last object which was written to the same variable. See Fig. 4-7B for illustrative example.

Mapping Rule 3.3 If an assignment object is created within multiple nested loops, then it still only belongs to one loop overmap. See Fig. 4-7C for illustrative example.

Mapping Rule 3.4 Any object belonging to a conditional overmap within a loop must share the same loop overmap as all objects which belong to the conditional overmap. See Fig. 4-7D for illustrative example.

Using this set of rules together with the developed OFS and CFS, it is determined, for each loop L'_k, which assignment objects belong to which loop overmaps. Unlike the handling of conditional fragments, outer loops are made to dominate the overloaded analysis as a result of Mapping Rule 3. While this rule may result in the collection of unnecessary data, it simplifies the analysis of any flow control nested within the loop.

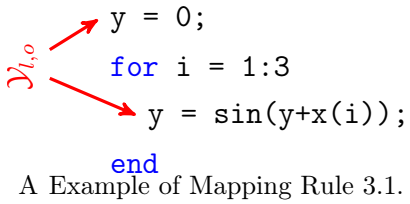
For instance, in the case of a conditional fragment nested within a loop (such as that shown in Fig. 4-7D), the loop overmap is always made to contain the iteration dependent conditional overmap. In the overmapping evaluation of such a code fragment, the iteration dependent conditional overmaps would be built on each iteration in order to properly propagate sparsity patterns. In the printing evaluation, however, conditional overmaps are unnecessary as it is ensured that all possible conditional overmaps belong to the loop overmap. Additionally, this provides a measure of safety for the printing evaluation by ensuring that all assignment objects created within loops are in the overmapped form.

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

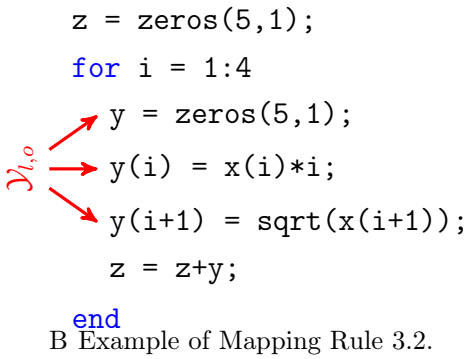
Mapping Rule 4: For any outer loop L'_k (that is, there does not exist L'_j such that $L'_k \subset L'_j$), if a variable `y` is written within I'_k, and read within $Succ\,(L'_k)$, then the last object written to `y` within the loop during the overmapping evaluation must be saved. Furthermore, the saved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

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

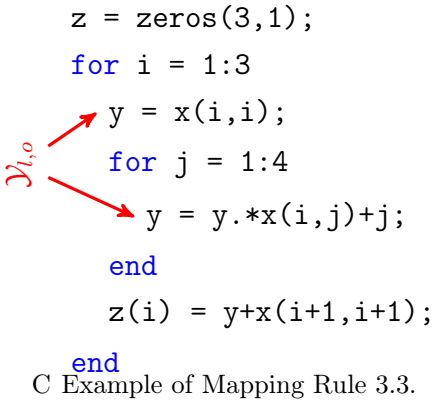
It is noted that the presented mapping rules do not address cases of loops containing `break` or `continue` statements. The manner in which the proposed method handles such cases is now briefly introduced. For the sake of conciseness, however, they will neither be discussed in detail nor addressed in the remaining sections. In the presence of `break/continue` statements, the true outputs of the loop (as addressed in Mapping Rule



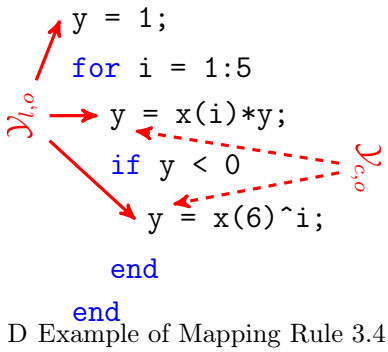
A Example of Mapping Rule 3.1.



B Example of Mapping Rule 3.2.



C Example of Mapping Rule 3.3.



D Example of Mapping Rule 3.4.

Figure 4-7. Illustrative examples of Mapping Rule 3. A) The variable `y` is an iteration dependent input to the loop, thus the objects assigned to the marked variables must be joined to create the corresponding overmapped input. During the overmapping evaluation phase, this is achieved by joining the four different objects assigned to `y`: the original input, together with the result of `sin(y + x(i))` for the three values of `i`. B) Since the variable `y` is initialized to zero then assigned to via subscript index assignment twice, the objects which result from all three assignments are made to belong to the loop overmap $\mathcal{Y}_{l,o}$. Thus, the loop overmap $\mathcal{Y}_{l,o}$ is built by joining 12 overloaded objects, three for each loop iteration. C) In this example, the variable `y` is written within a nested loop and is also an iteration dependent input to the nested loop. As a result of Mapping Rules 3.1 and 3.3, the corresponding loop overmap is made to contain the 15 different objects written to `y`: three from the outer loop assignment and 12 from the nested loop assignment. D) As a result of Mapping Rule 1.2, the conditional overmap $\mathcal{Y}_{c,o}$ is made to contain the objects assigned to `y` at both places within the loop, where a new conditional overmap is built on each iteration of the loop in the overmapping evaluation. As a result of Mapping Rule 3.4, the loop overmap $\mathcal{Y}_{l,o}$ is made to contain the object assigned to `y` prior to the loop, as well as the objects assigned to `y` across all overloaded evaluations of the loop.

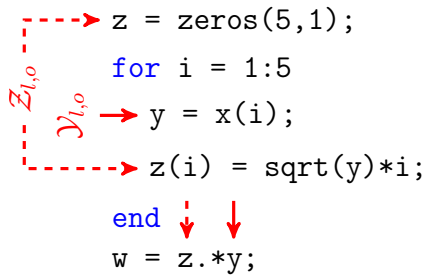


Figure 4-8. Illustrative example of Mapping Rule 4. In this example, both the variables `y` and `z` are outputs of the loop. Thus, the objects assigned to the variables `y` and `z` on the fifth and final iteration of the loop must be stored in the overmapping evaluation phase and returned as outputs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

4) are considered to be the union between all possible outputs of the loop (i.e. the outputs which result from any `break` statement firing across all iterations, the outputs which result from any `continue` statement firing on the final iteration, and the outputs which result from no `break` statements firing on any iteration and no `continue` statements firing on the final iteration). In the presence of `continue` statements, the overmapped loop inputs (as addressed in Mapping Rule 3.1) must be made to contain all possible inputs to the loop (i.e. any initial inputs together with any iteration dependent inputs which result from a `continue` statement firing, or no `continue` statements firing).

4.2.4 Overmapping Evaluation

The purpose of the overmapping evaluation is to build the aforementioned conditional overmaps and loop overmaps, as well as to collect data regarding organizational operations within loop statements. This overmapping evaluation is performed by evaluating the intermediate program on overloaded CADA objects, where no calculations are printed to file, but rather only data is collected. Recall now from Mapping Rules 1 and 3 that any object belonging to a conditional and/or loop overmap must be an assignment object. Additionally, all assignment objects are sent to the `VarAnalyzer` routine immediately after

they are created. Thus, building the conditional and loop overmaps may be achieved by performing overloaded unions within the variable analyzer routine immediately after the assignment objects are created. The remaining transformation routines must then control the flow of the program and manipulate the overloaded workspace such that the proper overmaps are built. The various tasks performed by the transformation routines during the overmapping evaluation of conditional and loop fragments is now described.

4.2.4.1 Overmapping evaluation of conditional fragments

During the overmapping evaluation of a conditional fragment, it is required to emulate the corresponding conditional statements of the original program. First, those branches on which overmapping evaluations are performed must be determined. This determination is made by analyzing the conditional variables given to the `IfInitialize` routine (`cadacond1`,...,`cadacondn-1` of Fig. 4-3), where each of these variables may take on a value of true, false, or indeterminate. Any value of indeterminate implies that the variable may take on the value of either true or false within the derivative program. Using this information, it can be determined if overmapping evaluations are not to be performed within any of the branches. For any such branches within which overmapping evaluations are not to be performed, empty evaluations are performed in a manner similar to those performed in the parsing evaluation. Next, it is required that all branches of the conditional fragment be evaluated independently (that is, Mapping Rule 2 must be adhered to). Thus, for a conditional fragment C'_k, if a variable is written within $Pred\,(C'_k)$, rewritten within $B'_{k,i}$, and then read within $B'_{k,j}$ $(i < j)$, then the variable analyzer will use the developed OMS to save the last assignment object written to the variable within $Pred\,(C'_k)$. The saved object may then be written to the overloaded workspace prior to the evaluation of any dependent branches using the outputs of the `IfIterStart` routines corresponding to the dependent branches. Finally, in order to ensure that any overmaps built after the conditional fragment are valid for all branches of the conditional fragment, all conditional overmaps associated with the conditional fragment must be assigned to

the overloaded workspace. For some conditional overmap, these assignments are achieved ideally within the `VarAnalyzer` at the time which the last assignment object belonging to the conditional overmap is assigned. If, however, such assignments are not possible (due to the variable being read prior to the end of the conditional fragment), then the conditional overmap may be assigned to the overloaded workspace via the outputs of the last `IfIterEnd` routine.

4.2.4.2 Overmapping evaluation of loops

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

4.2.5 Organizational Operations within `For` Loops

Consider that all organizational operations may be written as one or more references or assignments: horizontal or vertical concatenation can be written as multiple subscript index assignments, reshapes can be written as either a reference or a subscript index assignment, etc. Consider now that the derivative operation corresponding to a function reference/assignment is given by performing the same reference/assignment on the first dimension of the Jacobian. In the method of this paper, however, derivative variables are written as vectors of non-zeros. Thus, the derivative procedures corresponding to function references/assignments cannot be written in terms of function reference/assignment

indices. Moreover, when dealing with loops, it is often the case that function reference/assignment indices change on loop iterations, and thus the corresponding derivative procedures must be made to be iteration dependent. In this section a description of how organizational operations are handled within loops is given.

4.2.5.1 Example of an organizational operation within a loop

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

$$\mathbf{w}_i = \mathbf{f}(\mathbf{v}_i, \mathbf{j}_i^{\mathbf{v}}, \mathbf{j}_i^{\mathbf{w}}), \tag{4–1}$$

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

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

$$d^{\bar{\mathbf{w}}}_{\mathbf{x}[k_i^{\bar{\mathbf{w}}}(l)]} = d^{\bar{\mathbf{v}}}_{\mathbf{x}[k_i^{\bar{\mathbf{v}}}(l)]} \quad l = 1, \ldots, m_i \tag{4–2}$$

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

```
w.dx(logical(K(:,i))) = v.dx(nonzeros(K(:,i)));
```
(4–3)

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

4.2.5.2 Collecting and analyzing organizational operation data

As seen in the example above, derivative rules for organizational operations within loops rely on index matrices to print valid derivative variable references and assignments. Here it is emphasized that, given the proper index matrix, valid derivative calculations are printed to file in the manner shown in Eq. 4–3 for any organizational operation contained within a loop. The aforementioned example also demonstrates that the index matrices may be built given the overmapped inputs and outputs (for example, $\bar{\mathcal{W}}$ and $\bar{\mathcal{V}}$), together with the iteration dependent function reference and assignment indices (e.g. $\mathbf{j}_i^{\mathbf{v}}$ and $\mathbf{j}_i^{\mathbf{w}}$). While there exists no single operation in MATLAB which is of the form of Eq. (4–1), it is noted that the function reference and assignment indices may be easily determined for any organizational operation by collecting certain data at each call to the operation within a loop. Namely, for any single call to an organizational operation, any input reference/assignment indices and function sizes of all inputs and outputs are collected for each iteration of a loop within which the operation is contained. Given this information, the function reference and assignment indices are then determined for each iteration to rewrite the operation to one of the form presented in Eq. (4–1). In order to collect this data, each overloaded organizational operation must have a special routine written to store the required data when it is called from within a loop during the overmapping evaluations. Additionally, in the case of multiple nested loops, the data from each child loop must be neatly collected on each iteration of the parent loop's `ForIterEnd` routine. Furthermore, because it is required to obtain the overmapped versions of all inputs and outputs and it is sometimes the case that an input and/or output does not belong to a loop overmap, it is also sometimes the case that unions must be performed within the organizational operations themselves in order to build the required overmaps.

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

```
K2 = reshape(K1(:,i1),n,m2);
```
(4–4)

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

4.2.6 Printing Evaluation

During the printing evaluation, the derivative code is finally printed to a file by evaluating the intermediate program on instances of the CADA class, where each overloaded operation prints calculations to the file in the manner presented in Chapter 3. In order to print `if` statements and `for` loops to the file that contains the derivative

program, the transformation routines must both print flow control statements and ensure the proper overloaded objects exist in the overloaded workspace. Manipulating the overloaded workspace in this manner enables all overloaded operations to print calculations that are valid for the given flow control statements. Unlike in the overmapping evaluations, if the overloaded workspace is changed, then the proper re-mapping calculations must be printed to the derivative file such that all function variables and derivative variables within the printed file reflect the properties of the objects within the active overloaded workspace. In the following section, the concept of an overloaded re-map is first given. The various tasks carried out by the transformation routines to print derivative programs containing conditional and/or loop fragments are then explored.

4.2.6.1 Re-mapping of overloaded objects

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

the derivative file to compute `o.f` and `o.dx`. If it is desired to re-map $\mathcal{O}$ to $\mathcal{P}$ such that the variables `o.f` and `o.dx` are used to create the function variable, `p.f`, and derivative variable, `p.dx`, which reflect the properties of $\mathcal{P}$, the following steps are executed:

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

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

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

4.2.6.2 Printing evaluation of conditional fragments

Printing a valid conditional fragment to the derivative program requires that the following tasks must be performed. First, it is necessary to print the conditional statements, that is, print the statements `if`, `elseif`, `else`, and `end`. Second, it is required that each conditional branch is evaluated independently (as was the case with the

overmapping evaluation). Third, after the conditional fragment has been evaluated in the intermediate program (and printed to the derivative file), all associated conditional overmaps must be assigned to the proper variables in the intermediate program. In addition, all of the proper re-mapping calculations must be printed to the derivative file such that when each branch of the derivative conditional fragment is evaluated it will calculate derivative variables and function variables that reflect the properties of the active conditional overmaps in the intermediate program. These three aforementioned tasks are now explained in further detail.

The printing of the conditional branch headings is fairly straightforward due to the manner in which all expressions following the `if`/`elseif` statements in the user program are written to a conditional variable in the intermediate program (as seen in Fig. 4-3). Thus, each `IfIterStart` routine may simply print the branch heading as: `if cadacond1, elseif cadacond2, else` and so on. As each branch of the conditional fragment is then evaluated the overloaded CADA operations will print derivative and function calculations to the derivative file in the manner described in Chapter 3. As was the case with the overmapping evaluations, each of the branch fragments must be evaluated independently, where this evaluation is performed in the same manner as described in Section 4.2.4.1. Likewise, it is ensured that the overloaded workspace will contain all associated conditional overmaps in the same manner as in the overmapping evaluation. Unlike the overmapping evaluation, however, conditional overmaps are not built during the printing evaluations. Instead, the conditional overmaps are stored within memory and may be accessed by the transformation routines in order to print the proper re-mapping calculations to the derivative file. For some conditional fragment C'_k within which the variable `y` has been written, the following calculations are required to print

the function variable and derivative variable that reflect the properties of the conditional overmap $\mathcal{Y}_{c,o}$[1]

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

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

4.2.6.3 Printing evaluation of loops

In order to print a valid loop fragment to the derivative program an overloaded evaluation of a single iteration of the transformed loop is performed in the intermediate program. To ensure that all printed calculations are valid for each iteration of the loop in the derivative program, these overloaded evaluations are performed only on loop overmaps. The printing evaluation of loops is then completed as follows. When an outermost loop is encountered in the printing evaluation of the intermediate program, the `ForInitialize` routine must first use the OMS to determine which, if any, objects belong to loop overmaps but are created prior to the loop, that is, loop inputs which are loop

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

iteration dependent. Any of these previously created objects will have been saved within the `VarAnalyzer` routine, and the `ForInitialize` routine may then access both the previously saved objects and the loop overmaps to which they belong and then re-map the saved objects to their overmapped form. As it is required that the overmapped inputs to the loop be active in the overloaded workspace, the `ForInitialize` then uses its outputs, `ForEvalStr` and `ForEvalVar`, to assign the overmapped objects to the proper variables in the overloaded workspace. Next, the `ForIterStart` routine must print out the outermost `for` statement. Here it is noted that all outer loops must evaluate for a fixed number of iterations, so if the loop is to be run for 10 iterations, then the `for` statement would be printed as `for cadaforcount = 1:10`. The `cadaforcount` variable will then be used in the derivative program to reference columns off of the index matrices presented in Section 4.2.5 and to reference the user defined loop variable. As the loop is then evaluated on loop overmaps, all overloaded operations will print function and derivative calculations to file in the manner presented in Chapter 3, with the exception being the organizational operations. The organizational operations will instead use stored global data to print derivative references and assignments using the index matrices as described in Section 4.2.5. Moreover, during the evaluation of the loop, only organizational operations are cognizant of the fact that the evaluations are being performed within the loop. Thus, when operations are performed on loop overmaps, the resulting assignment objects can sometimes be computed to be different from their overmapped form. In order to adhere to Mapping Rule 3,after each variable assignment within the loop, the assignment object is sent to the `VarAnalyzer` routine which then re-maps the assigned object to the stored loop overmap to which it belongs.

Nested loops are handled different from non-nested loops. When a nested loop is reached during the printing evaluation, the active overloaded workspace will only contain loop overmaps, thus it is not required to re-map any nested loop inputs to their overmapped form. What is required though, is that the `ForInitialize` routine check

to see if any index matrix references and/or reshapes must be printed, where if this is the case, the proper references and/or reshapes are stored during the organizational operation data analysis. These references/reshapes are then accessed and printed to file. The printing of the actual nested loop statements is then handled by the `ForIterStart` routine, where, unlike outer loops, nested loops may run for a number of iterations which is dependent upon a parent loop. If it is the case that a nested loop changes size depending upon a parent loop's iteration, then the loop statement is printed as: `for cadaforcount2 = 1:K(cadaforcount1)`, where `cadaforcount2` takes on the nested loop iteration, `cadaforcount1` is the parent loop iteration, and `K` is a vector containing the sizes of the nested loop. Any evaluations performed within the nested loop are then handled exactly as if they were within an outer loop, and when the `ForIterEnd` routine of the nested loop is encountered it prints the `end` statement.

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

4.2.7 Multiple User Functions

At this point, the methods have been described that transform a program containing only a single function into a derivative program containing only a single function. It

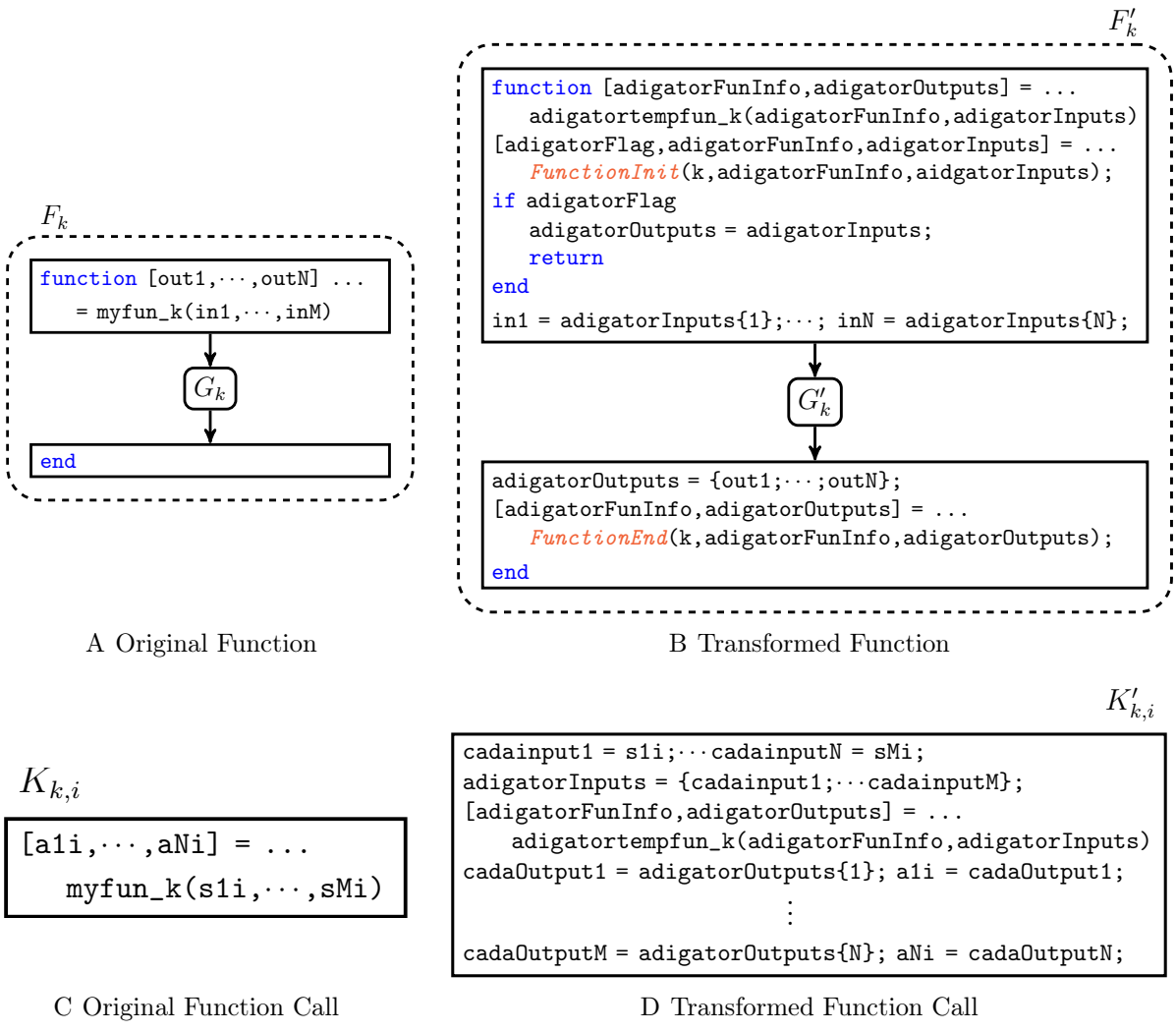


Figure 4-9. Transformation of original function F_k to transformed function F'_k and transformation of the i^{th} call to function F_k, $K_{k,i}$, to the i^{th} call to function F'_k, $K'_{k,i}$.

is often the case, however, that a user program consists of multiple functions and/or sub-functions. The method of this paper handles both the cases of called functions and sub-functions in the same manner. In order to deal with programs containing multiple function calls, the transformations of Fig. 4-9 are applied during the user source to intermediate source transformation phase. In the parsing phase, it is then determined which functions F'_k are called more than a single time. If a function is called only once, then the input and output objects are saved during the overmapping evaluation phase.

Then, in the printing evaluation phase, when the function F'_k is called, the `FunctionInit` routine returns the saved outputs and sets the `adigatorFlag` true, returning the outputs to the calling function. After the calling function has been printed, the function F'_k is then evaluated in the manner discussed in Section 4.2.6.

In the case that a function F'_k is called multiple times, it is required to build a set of overmapped inputs and overmapped outputs of the function. The building of overmapped inputs and outputs is performed during the overmapping evaluation phase by joining the inputs across each function call, evaluating the called function on each unique set of inputs, and joining all possible outputs. Moreover, the outputs of each call are stored in memory for use in the printing evaluation phase. In order to allow for function-call-dependent organizational operations, each call is assigned an iteration count and organizational operations are then handled in the manner presented in Section 4.2.5. During the printing phase, when the function F'_k is called, the `FunctionInit` routine re-maps the given inputs for the particular call to the overmapped inputs, prints the function call, re-maps the overmapped outputs to the stored outputs, and then returns the stored outputs. The function is then not evaluated by setting the `adigatorFlag` to true. After the calling function has finished printing, the function F'_k is then evaluated on the set of stored overmapped inputs and all organizational operations are treated as if they were being called from within a loop.

4.3 Higher-Order Derivatives

An advantage of the method of this dissertation is that, by producing stand-alone derivative source code, the method may be applied recursively to generate n^{th}-order derivative files. The methods of Chapter 3 themselves may be used to do so, however, the resulting n^{th}-order code would contain redundant 1^{st} through $(n-1)^{th}$ derivative computations. To illustrate, consider applying the methods of Chapter 3 to a function which computes `y = sin(x)`, and then again on the resulting derivative code. The

transformation would performed as follows:

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

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 method of this dissertation, 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 a file 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.

4.4 Illustrative Examples

The method described in Section 4.2 (which is implemented in the software ADiGator) is now applied to two illustrative examples. The first example provides a detailed examination of the process that ADiGator uses to handle conditional statements. The second example is the well known Speelpenning problem of Ref. [57] and demonstrates the manner in which the ADiGator handles a `for` loop.

4.4.1 Example 1: Conditional Statement

In this example, the developed method is used to transform a user program containing a conditional statement into a derivative program containing the same conditional statement. The original user program and the transformed intermediate program is shown in Figure 4-10, where it is seen that the variable `y` is written within both the `if` branch and the `else` branch. Furthermore, because the variable `y` is read after the conditional

fragment, a conditional overmap, $\mathcal{Y}_{c,o}$, is required to print derivative calculations that are valid for both the `if` and `else` branches.

Now let $\mathcal{Y}_{\texttt{if}}$ and $\mathcal{Y}_{\texttt{else}}$ be the overloaded objects written to `y` as a result of the overloaded evaluation of the `if` and `else` branches, respectively, in the intermediate program. Using this notation, the required conditional overmap is then given by $\mathcal{Y}_{c,o} = \mathcal{Y}_{\texttt{if}} \cup \mathcal{Y}_{\texttt{else}}$, where $\mathcal{Y}_{c,o}$ is built during the overmapping evaluation of the intermediate program shown in Figure 4-10. Fixing the input to be a vector of length five, the sparsity patterns of the Jacobians defined by $\mathcal{Y}_{\texttt{if}}$, $\mathcal{Y}_{\texttt{else}}$, and $\mathcal{Y}_{c,o}$ are shown in Figure 4-11, and the final transformed derivative program is seen in Figure 4-12. From Figure 4-12, it is seen that re-mapping calculations are required from within both the `if` and `else` branches, where these re-mapping calculations represent the mapping of the non-zero elements of the Jacobians shown in Figures 4-11A and 4-11B into those shown in Figure 4-11C. More precisely, in the `if` branch of the derivative program, the assignment indices `Gator1Indices.Index4` maps the derivative variable of $\mathcal{Y}_{\texttt{if}}$ into elements $[1, 2, 3, 4, 5, 6, 7, 8, 13]$ of the derivative variable of $\mathcal{Y}_{c,o}$. Similarly, in the `else` branch of the derivative program, the assignment index `Gator1Indices.Index8` maps the derivative variable of $\mathcal{Y}_{\texttt{else}}$ into the $[1, 6, 7, 8, 9, 10, 11, 12, 13]$ elements of the derivative variable of $\mathcal{Y}_{c,o}$. Thus, the evaluation of the derivative program shown in Figure 4-12 always produces a 13 element derivative variable, `z.dx`, which is mapped into a Jacobian of the form shown in Figure 4-11C using the mapping index written to `z.dx_location`. Due to the nature of the `if` and `else` branches of the derivative program, at least 4 elements of the derivative variable `z.dx` will always be identically zero, where the locations of the zero elements will depend upon which branch of the conditional block is taken.

4.4.2 Example 2: Loop Statement

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

User Program	Intermediate Program

`function z = myfun(x)`
 	
`N  = 5;`
`x1 = x(1);`
`xN = x(N);`
 	
`if x1 > xN`
`  y = x*x1;`
`else`
`  y = x*xN;`
`end`
 	
`z = sin(y);`
	`function [adigatorFunInfo, adigatorOutputs] = ...`
	`    adigatortempfunc1(adigatorFunInfo,adigatorInputs)`
	`[adigatorFlag, adigatorFunInfo, adigatorInputs] = ...`
	`   FunctionInit(1,adigatorFunInfo,adigatorInputs);`
	`if adigatorFlag; adigatorOutputs = adigatorInputs; return; end;`
	`x = adigatorInputs{1};`
	
	`N  = 5;`
	`N  = VarAnalyzer('N  = 5;',N,'N',0);`
	`x1 = x(1);`
	`x1 = VarAnalyzer('x1 = x(1);',x1,'x1',0);`
	`xN = x(N);`
	`xN = VarAnalyzer('xN = x(N);',xN,'xN',0);`
	
	`% ADiGator IF Statement #1: START`
	`cadacond1 = x1 > xN;`
	`cadacond1 = VarAnalyzer('cadacond1 = x1 > xN',cadacond1,'cadacond1',0);`
	`IfInitialize(1,cadacond1,[]);`
	`IfIterStart(1,1);`
	`    y = x*x1;`
	`    y = VarAnalyzer('y = x*x1;',y,'y',0);`
	`IfIterEnd(1,1);`
	`[IfEvalStr, IfEvalVar] = IfIterStart(1,2);`
	`if not(isempty(IfEvalStr)); cellfun(@eval,IfEvalStr); end`
	`    y = x*xN;`
	`    y = VarAnalyzer('y = x*xN;',y,'y',0);`
	`[IfEvalStr, IfEvalVar] = IfIterEnd(1,2);`
	`if not(isempty(IfEvalStr)); cellfun(@eval,IfEvalStr); end`
	`% ADiGator IF Statement #1: END`
	
	`z = sin(y);`
	`z = VarAnalyzer('z = sin(y);',z,'z',0);`
	
	`adigatorOutputs = {z};`
	`[adigatorFunInfo, adigatorOutputs] = ...`
	`    FunctionEnd(1,adigatorFunInfo,adigatorOutputs);`

Figure 4-10. User source to intermediate source transformation for Example 1.

to be transformed computes the Speelpenning function [57] given as

$$y = \prod_{i=1}^{N} x_i, \tag{4–5}$$

where Eq. (4–5) is implemented using a `for` loop as shown in Fig. 4-13. From Fig. 4-13, two major challenges of transforming the loop are identified. First, it is seen that the variable `y` is an input to the loop, rewritten within the loop, and read after the loop (as the output of the function), thus a loop overmap, $\mathcal{Y}_{c,o}$ must be built to print valid derivative calculations within the loop. Second, it is seen that the reference `x(I)`

$$\underset{\text{A } \mathbf{struct}(\mathcal{Y}_{\texttt{if}}.\texttt{deriv})}{\begin{bmatrix} d_1 & 0 & 0 & 0 & 0 \\ d_2 & d_6 & 0 & 0 & 0 \\ d_3 & 0 & d_7 & 0 & 0 \\ d_4 & 0 & 0 & d_8 & 0 \\ d_5 & 0 & 0 & 0 & d_9 \end{bmatrix}} \quad \underset{\text{B } \mathbf{struct}(\mathcal{Y}_{\texttt{else}}.\texttt{deriv})}{\begin{bmatrix} d_1 & 0 & 0 & 0 & d_5 \\ 0 & d_2 & 0 & 0 & d_6 \\ 0 & 0 & d_3 & 0 & d_7 \\ 0 & 0 & 0 & d_4 & d_8 \\ 0 & 0 & 0 & 0 & d_9 \end{bmatrix}} \quad \underset{\text{C } \mathbf{struct}(\mathcal{Y}_{c,o}.\texttt{deriv})}{\begin{bmatrix} d_1 & 0 & 0 & 0 & d_9 \\ d_2 & d_6 & 0 & 0 & d_{10} \\ d_3 & 0 & d_7 & 0 & d_{11} \\ d_4 & 0 & 0 & d_8 & d_{12} \\ d_5 & 0 & 0 & 0 & d_{13} \end{bmatrix}}$$

Figure 4-11. Derivative sparsity patterns of $\mathcal{Y}_{\texttt{if}}$, $\mathcal{Y}_{\texttt{else}}$, and $\mathcal{Y}_{c,o} = \mathcal{Y}_{\texttt{if}} \cup \mathcal{Y}_{\texttt{else}}$.

depends upon the loop iteration, where the object assigned to `x` has possible non-zero derivatives. Thus, an index matrix, $\mathbf{K}$, and an overmapped `subsref` output, $\mathcal{R}_o$, must be built to allow for the iteration dependent derivative reference corresponding to the function reference `x(I)`. To further investigate, let $\mathcal{Y}_i$ be the object written to `y` after the evaluation of the i^{th} iteration of the loop from within the intermediate program. Furthermore, R_i is allowed to be the intermediate object created as a result of the overloaded evaluation of `x(I)` within the intermediate program. The overmapped objects $\mathcal{Y}_{l,o}$ and $\mathcal{R}_o$ are now defined as

$$\mathcal{Y}_{l,o} = \bigcup_{i=0}^{N} \mathcal{Y}_i \tag{4–6}$$

and

$$\mathcal{R}_o = \bigcup_{i=1}^{N} \mathcal{R}_i, \tag{4–7}$$

where $\mathcal{Y}_0$ represents the object written to `y` prior to the evaluation of the loop in the intermediate program. Now, as the input object, $\mathcal{X}$, has a Jacobian with a diagonal sparsity pattern (which does not change), then each object $\mathcal{R}_i$ will have a $1 \times N$ gradient where the i^{th} element is a possible non-zero. Thus, the union of all such gradients over i results in all N elements being possibly non-zero, that is, a full gradient. Within the derivative program, the reference operation must then result in a derivative variable of length N, where on iteration i, the i^{th} element of the derivative variable corresponding to $\mathcal{X}$ must be assigned to the i^{th} element of the derivative variable corresponding to $\mathcal{R}_o$. This reference and assignment is done as described in Section 4.2.5 using the index matrix

$\mathbf{K} \in \mathbb{Z}^{N \times N}$, where

$$K_{i,j} = \begin{cases} i, & i = j \qquad i = 1, \ldots, N \\ 0, & i \neq j \qquad j = 1, \ldots, N \end{cases}. \tag{4–8}$$

Now, because $\mathcal{R}_i$ contains a possible non-zero derivative in the i^{th} location of the gradient and $\mathcal{Y}_i = \mathcal{Y}_{i-1} * \mathcal{R}_i$, $\mathcal{Y}_i$ will contain i possible non-zero derivatives in the first i locations. Furthermore, the union of $\mathcal{Y}_i$ over $i = 0, \ldots, N$ results in an object $\mathcal{Y}_{l,o}$ containing N possible non-zero derivatives (that is, a full gradient). Thus, in the derivative program, the object $\mathcal{Y}_0$ must be re-mapped to the object $\mathcal{Y}_{l,o}$ prior to the printing evaluation of the loop in the intermediate program. The result of the printing evaluation of the intermediate program for $N = 10$ is seen in Fig. 4-14. In particular, Fig. 4-14 shows the re-mapping of the object $\mathcal{Y}_0$ to $\mathcal{Y}_{l,o}$ immediately prior to the loop. Additionally, it is seen that the derivative variable `cada1f1dx` (which results from the aforementioned reference within the loop) is of length 10, where the reference and assignment of `cada1f1dx` depends upon the loop iteration. This reference and assignment is made possible by the index matrix $\mathbf{K}$ who is assigned to the variable `Gator1Indices.Index1` within the derivative program.

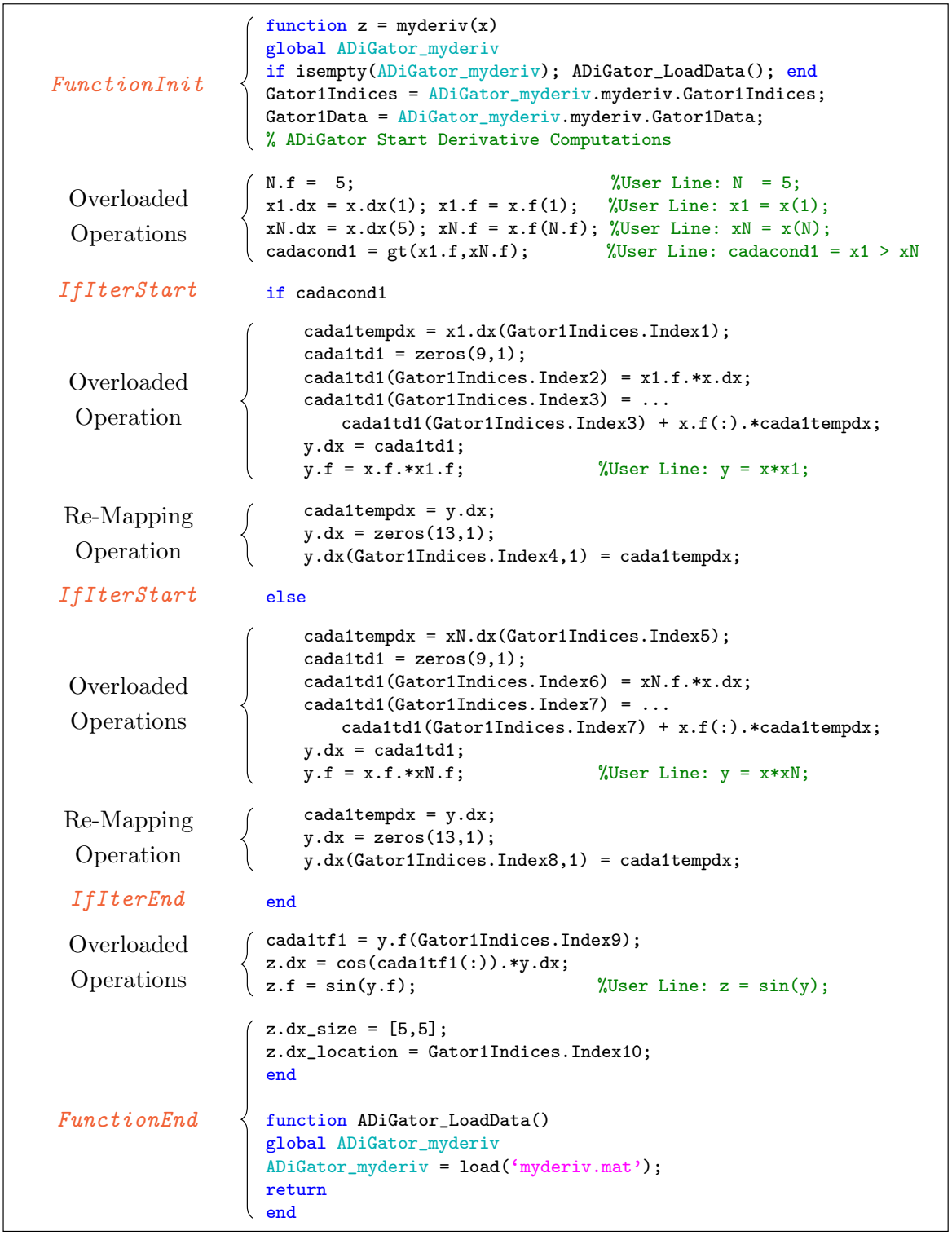


Figure 4-12. Transformed derivative program for Example 1 showing from where each line is printed.

User Program	Intermediate Program

function y = SpeelFun(x)
 	
y = 1;
 	
for I = 1:length(x)
 	
  y = y*x(I);
 	
end
	function [adigatorFunInfo, adigatorOutputs] = ...
	  adigatortempfun_1(adigatorFunInfo,adigatorInputs)
	[flag, adigatorFunInfo, adigatorInputs] = ...
	  FunctionInit(1,adigatorFunInfo,adigatorInputs);
	if flag; adigatorOutputs = adigatorInputs; return; end;
	x = adigatorInputs{1};
	
	y = 1;
	y = *VarAnalyzer*('y = 1;',y,'y',0);
	
	% ADiGator FOR Statement #1: START
	cadaLoopVar_1 = 1:length(x);
	cadaLoopVar_1 = ...
	  VarAnalyzer('cadaLoopVar_k = 1:length(x);',cadaLoopVar_k,'cadaLoopVar_k',0);
	[adigatorForVar_1, ForEvalStr, ForEvalVar] = *ForInitialize*(1,cadaLoopVar_1);
	if not(isempty(ForEvalStr)); cellfun(@eval,ForEvalStr); end
	for adigatorForVar_1_i = adigatorForVar_1
	  cadaForCount_1 = *ForIterStart*(1,adigatorForVar_1_i);
	  I = cadaLoopVar_1(:,cadaForCount_1);
	  I = *VarAnalyzer*('I = cadaLoopVar_1(:,cadaForCount_1);',I,'I',0);
	  y = y*x(I);
	  y = *VarAnalyzer*('y = y*x(I);',y,'y',0);
	  [ForEvalStr, ForEvalVar]= *ForIterEnd*(1,adigatorForVar_1_i);
	end
	if not(isempty(ForEvalStr)); cellfun(@eval,ForEvalStr); end
	% ADiGator FOR Statement #1: END
	
	adigatorOutputs = {y};
	[adigatorFunInfo, adigatorOutputs] = ...
	  FunctionEnd(1,adigatorFunInfo,adigatorOutputs);

Figure 4-13. User source to intermediate source transformation for Speelpenning problem.

FunctionInit

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

Overloaded Operations Re-Mapping Operation

```
y.f =  1;                    %User Line: y = 1;
cada1f1 = length(x.f);
cadaforvar1.f = 1:cada1f1;  %User Line: cadaforvar1 = 1:length(x);
y.dx = zeros(10,1);
```

ForInitialize

```
for cadaforcount1 = 1:10
```

Overloaded Operation

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

Organizational Overloaded Operation

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

Overloaded Operation

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

ForIterEnd

```
end
```

FunctionEnd

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

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

Figure 4-14. Transformed derivative program for Speelpenning problem for $N = 10$ showing origin of each printed line of code.

CHAPTER 5
APPLICATION TO DIRECT COLLOCATION OPTIMAL CONTROL

Over the past two decades the direct collocation method of solving optimal control problems has gained a great deal of popularity. In a direct collocation approach, the state is approximated using a set of trial (basis) functions and the dynamics are collocated at a specified set of points in the time interval. The state and control are discretized and the differential-algebraic equations are enforced at a set of discrete points. The new discretized problem is then solved by means of either a first- or second-derivative NLP solver. In a first-derivative (quasi-Newton) NLP solver, the objective function gradient and constraint Jacobian are used together with a dense quasi-Newton approximation of the Lagrangian Hessian. In a second-derivative (Newton) NLP solver, the objective function gradient and constraint Jacobian are used together with the Hessian of the NLP Lagrangian. While it is known that providing the Lagrangian Hessian can significantly improve the computational performance of an NLP solver, quasi-Newton methods are more commonly used due to the difficulty presented in computing the Lagrangian Hessian. Multiple optimal control algorithms which use direct collocation methods have been developed, among them are the well known programs SOCS [8], DIDO [51], DIRCOL [59], GPOPS [49], and GPOPS-II [46].

In this chapter, the developed AD software of this dissertation, ADiGator, is used in order to compute the first and second derivatives of the NLP which arises from direct collocation methods. While the methods of this chapter may be applied to multiple direct collocation schemes, the discussion is focused upon an hp-adaptive Legendre-Gauss-Radau [17, 18, 23–25, 43] scheme which has been coded in the commercial optimal control software GPOPS-II. The GPOPS-II software provides a good test environment due to the manner in which GPOPS-II requires only the derivatives of the control problem [46], together with the fact that it is implemented in the same language as the ADiGator tool. This chapter is organized as follows. In Section 5.1 a generic transformed Bolza

optimal control problem is introduced. In Section 5.2 the NLP which results from the hp Legendre-Gauss-Radau collocation of the transformed Bolza problem is formulated. In Section 5.3 a discussion is given on the separation of vectorized functions from the NLP of Section 5.2. In Section 5.4 the compressible nature of vectorized functions is explored, and in Section 5.5, three optimal control test problems are solved.

5.1 Transformed Bolza Optimal Control Problem

In this chapter a single phase continuous time Bolza optimal control problem transformed from the domain $t \in [t_0,\ t_f]$ to the fixed domain $\tau \in [-1,\ 1]$ is considered. This transformation is achieved by the affine transformation

$$\tau = \frac{2}{t_f - t_0} t + \frac{t_f + t_0}{t_f - t_0}. \tag{5–1}$$

The continuous time Bolza problem is then defined on the interval $[-1,\ 1]$ as follows. Determine the state, $\mathbf{y}(\tau) \in \mathbb{R}^{n_y}$, control, $\mathbf{u}(\tau) \in \mathbb{R}^{n_u}$, initial time, t_0, and final time, t_f, that minimizes the cost functional

$$J = \Phi(\mathbf{y}(-1), t_0, \mathbf{y}(+1), t_f) + \frac{t_f - t_0}{2} \int_{-1}^{+1} g(\mathbf{y}(\tau), \mathbf{u}(\tau)) d\tau, \tag{5–2}$$

subject to the dynamic constraints

$$\frac{d\mathbf{y}}{d\tau} = \frac{t_f - t_0}{2} \mathbf{a}(\mathbf{y}(\tau), \mathbf{u}(\tau)) \in \mathbb{R}^{n_y}, \tag{5–3}$$

the path constraints

$$\mathbf{c}_{\min} \leq \mathbf{c}(\mathbf{y}(\tau), \mathbf{u}(\tau)) \leq \mathbf{c}_{\max} \in \mathbb{R}^{m_c}, \tag{5–4}$$

and boundary conditions

$$\boldsymbol{\phi}(\mathbf{y}(-1), t_0, \mathbf{y}(+1), t_f) = \mathbf{0} \in \mathbb{R}^{m_\phi}. \tag{5–5}$$

The optimal control problem of Eqs. (5–2)–(5–5) will be referred to as the transformed continuous Bolza problem. Here it is noted that a solution to the transformed continuous Bolza problem can be transformed from the domain $\tau \in [-1,\ +1]$ to the domain

$t \in [t_0,\ t_f]$ via the affine transformation

$$t = \frac{t_f - t_0}{2}\tau + \frac{t_f - t_0}{2}. \tag{5–6}$$

5.2 Formulation of NLP Resulting from Radau Collocation

In this section the NLP which arises from the multiple-interval Legendre-Gauss-Radau (LGR) collocation of the transformed Bolza problem of Section 5.1 is given. In the LGR collocation method, the interval $\tau \in [-1,\ 1]$ is divided into a mesh consisting of K mesh intervals defined by the mesh points $-1 = T_0 < T_1 < \cdots < T_K = +1$. In each mesh interval k, the state is approximated by a $(N_k + 1)^{th}$ degree Lagrange polynomial such that

$$\mathbf{y}^{(k)}(\tau) \approx \mathbf{Y}^{(k)}(\tau) = \sum_{i=1}^{N_k+1} \mathbf{Y}_i^{(k)} l_i^{(k)}(\tau), \quad l_i^{(k)}(\tau) = \prod_{\substack{j=1 \\ j\neq i}}^{N_k} \frac{\tau - \tau_j^{(k)}}{\tau_i^{(k)} - \tau_j^{(k)}}, \tag{5–7}$$

where $\mathbf{Y}^{(k)}(\tau)$ is the approximation of the state $\mathbf{y}^{(k)}(\tau)$ in the k^{th} mesh interval, $\tau \in [-1,\ 1]$, $l_i^{(k)}(\tau)$, $(i = 1, \ldots, N_k + 1)$ is a basis of Lagrange polynomials, $\tau_1^{(k)}, \ldots, \tau_{N_k}^{(k)}$ are the Legendre-Gauss-Radau collocation points defined on the subinterval $\tau^{(k)} \in [T_{k-1},\ T_k)$, and $T_{N_k+1}^{(k)} = T_K$ is a noncollocated point. The dynamic constraints of Eq. (5–3) are enforced by either differentiating or integrating Eq. (5–7) with respect to τ to yield the defect constraints. Moreover, the quadrature rule used to approximate the integral of Eq. (5–2) is obtained by integrating Eq. (5–7) with respect to τ.

Using the variables $\mathbf{Y}_i^{(k)} \in \mathbb{R}^{n_y}$ and $\mathbf{U}_i^{(k)} \in \mathbb{R}^{n_u}$ to represent the discretized state and control corresponding to the point $\tau_i^{(k)}$, the collection of discrete state and control variables across the k^{th} interval are defined by

$$\mathbf{Y}^{(k)} = \begin{bmatrix} \mathbf{Y}_1^{(k)} & \mathbf{Y}_2^{(k)} & \cdots & \mathbf{Y}_{N_k}^{(k)} \end{bmatrix} \in \mathbb{R}^{n_y \times N}, \tag{5–8}$$

and

$$\mathbf{U}^{(k)} = \begin{bmatrix} \mathbf{U}_1^{(k)} & \mathbf{U}_2^{(k)} & \cdots & \mathbf{U}_{N_k}^{(k)} \end{bmatrix} \in \mathbb{R}^{n_u \times N}, \tag{5–9}$$

respectively. It is noted that continuity in the state at the interior mesh points $k \in [1,\ldots,K-1]$ is enforced via the condition $\mathbf{Y}_{N_k+1}^{(k)} = \mathbf{Y}_1^{(k+1)}, \quad (k=1,\ldots,K-1)$, where the same variable is used for both $\mathbf{Y}_{N_k+1}^{(k)}$ and $\mathbf{Y}_1^{(k+1)}$. The values of the state and control corresponding to the $N = \sum_{k=1}^{K} N_k$ LGR points are then collected into the matrices

$$\mathbf{Y} = \left[\begin{array}{cccc} \mathbf{Y}^{(1)} & \mathbf{Y}^{(2)} & \cdots & \mathbf{Y}^{(K)} \end{array} \right] \in \mathbb{R}^{n_y \times N}, \tag{5–10}$$

and

$$\mathbf{U} = \left[\begin{array}{cccc} \mathbf{U}^{(1)} & \mathbf{U}^{(2)} & \cdots & \mathbf{U}^{(K)} \end{array} \right] \in \mathbb{R}^{n_u \times N}, \tag{5–11}$$

respectively. Allowing $\mathbf{Y}_i$ and $\mathbf{U}_i$ to represent the i^{th} columns of $\mathbf{Y}$ and $\mathbf{U}$, the continuous functions $g(\mathbf{y}(\tau),\mathbf{u}(\tau))$, $\mathbf{a}(\mathbf{y}(\tau),\mathbf{u}(\tau))$, and $\mathbf{c}(\mathbf{y}(\tau),\mathbf{u}(\tau))$ of Section 5.1 evaluated at the points $(\mathbf{Y}_i,\mathbf{U}_i),\ (i=1,\ldots,N)$ are defined as

$$\mathbf{G}(\mathbf{Y},\mathbf{U}) = \left[\begin{array}{cccc} g(\mathbf{Y}_1,\mathbf{U}_1) & g(\mathbf{Y}_2,\mathbf{U}_2) & \cdots & g(\mathbf{Y}_N,\mathbf{U}_N) \end{array} \right] \in \mathbb{R}^{1\times N}, \tag{5–12}$$

$$\mathbf{A}(\mathbf{Y},\mathbf{U}) = \left[\begin{array}{cccc} \mathbf{a}(\mathbf{Y}_1,\mathbf{U}_1) & \mathbf{a}(\mathbf{Y}_2,\mathbf{U}_2) & \cdots & \mathbf{a}(\mathbf{Y}_N,\mathbf{U}_N) \end{array} \right] \in \mathbb{R}^{n_y\times N}, \tag{5–13}$$

and

$$\mathbf{C}(\mathbf{Y},\mathbf{U}) = \left[\begin{array}{cccc} \mathbf{c}(\mathbf{Y}_1,\mathbf{U}_1) & \mathbf{c}(\mathbf{Y}_2,\mathbf{U}_2) & \cdots & \mathbf{c}(\mathbf{Y}_N,\mathbf{U}_N) \end{array} \right] \in \mathbb{R}^{m_c\times N}, \tag{5–14}$$

respectively.

Allowing $\mathbf{Y}_{N+1} \in \mathbb{R}^{n_y}$ to be the state discretized at the point $\tau = +1$, the NLP may now be formed as follows. Determine the discrete variables $\mathbf{Y} \in \mathbb{R}^{n_y\times N}$, $\mathbf{Y}_{N+1} \in \mathbb{R}^{n_y}$, $\mathbf{U} \in \mathbb{R}^{n_u\times N}$, $t_0 \in \mathbb{R}$, and $t_f \in \mathbb{R}$ which minimize the cost function

$$J \approx \Phi(\mathbf{Y}_1,t_0,\mathbf{Y}_{N+1},t_f) + \frac{t_f-t_0}{2}\mathbf{G}(\mathbf{Y},\mathbf{U})\mathbf{w}, \tag{5–15}$$

subject to the defect constraints

$$\left[\begin{array}{cc} \mathbf{Y} & \mathbf{Y}_{N+1} \end{array} \right] \mathbf{D}^{\mathsf{T}} - \frac{t_f-t_0}{2}\mathbf{A}(\mathbf{Y},\mathbf{U})\mathbf{B}^{\mathsf{T}} = \mathbf{0} \in \mathbb{R}^{n_y\times N}, \tag{5–16}$$

the path constraints

$$\mathbf{C}_{\min} \leq \mathbf{C}(\mathbf{Y}, \mathbf{U}) \leq \mathbf{C}_{\max} \in \mathbb{R}^{m_c \times N}, \tag{5–17}$$

and boundary conditions

$$\boldsymbol{\phi}(\mathbf{Y}_1, t_0, \mathbf{Y}_{N+1}, t_f) = \mathbf{0} \in \mathbb{R}^{m_\phi}, \tag{5–18}$$

where $\mathbf{w} \in \mathbb{R}^N$ is a column vector of LGR quadrature weights, and the matrices $\mathbf{D} \in \mathbb{R}^{N \times (N+1)}$ and $\mathbf{B} \in \mathbb{R}^{N \times N}$ are constant matrices whose values are determined by the given mesh together with whether a differentiation or integration scheme is being used. If an LGR differentiation scheme is being used, then the matrix $\mathbf{D}$ is the LGR differentiation matrix and $\mathbf{B}$ is the identity matrix. If an LGR integration scheme is being used, then the matrix $\mathbf{D}$ consists of entries of 0, -1, and $+1$, and the matrix $\mathbf{B}$ is the LGR integration matrix.

5.3 Discretization Separability

In order to solve the NLP of Section 5.2 with a second-derivative (Newton) NLP solver, it is required that the objective gradient, constraint Jacobian, and Lagrangian Hessian be iteratively supplied to the NLP solver. While one could simply obtain these derivatives by differentiating the objective and constraints of Section 5.2, it is often more efficient to instead differentiate the optimal control functions and to then build the NLP derivatives. By differentiating the optimal control functions, one may then take advantage of discretization separability [7] and the sparse nature of the vectorized functions discussed in Section 3.6.

That is, because only the endpoint functions $\boldsymbol{\Phi}(\cdot)$ and $\phi(\cdot)$ and the vectorized functions $\mathbf{G}(\cdot)$, $\mathbf{A}(\cdot)$, and $\mathbf{C}(\cdot)$ may change, only the first and second derivatives of the endpoint functions and vectorized functions with respect to their given arguments are required in order to build the first and second derivatives of the NLP. In order to concisely write the required optimal control derivatives, a single endpoint and vectorized function are formed. In order to formulate the endpoint function, the variable $\mathbf{v} \in \mathbb{R}^{n_v}$,

$n_v = 2n_y + 2$, is first constructed such that

$$\mathbf{v} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_{N+1} \\ t_0 \\ t_f \end{bmatrix}. \tag{5–19}$$

The endpoint function, $\boldsymbol{\psi} : \mathbb{R}^{n_v} \to \mathbb{R}^{m_\psi}$, $m_\psi = m_\phi + 1$, is then constructed as

$$\boldsymbol{\psi}(\mathbf{v}) = \begin{bmatrix} \Phi(\mathbf{Y}_1, t_0, \mathbf{Y}_{N+1}, t_f) \\ \boldsymbol{\phi}(\mathbf{Y}_1, t_0, \mathbf{Y}_{N+1}, t_f) \end{bmatrix}. \tag{5–20}$$

In order to formulate a vectorized function, first consider the continuous variable $\mathbf{x}(\tau) \in \mathbb{R}^{n_x}$, $n_x = n_y + n_u$, such that

$$\mathbf{x}(\tau) = \begin{bmatrix} \mathbf{y}(\tau) \\ \mathbf{u}(\tau) \end{bmatrix}, \tag{5–21}$$

and the corresponding discretized variable $\mathbf{X} \in \mathbb{R}^{n_x \times N}$,

$$\mathbf{X} = \begin{bmatrix} \mathbf{Y} \\ \mathbf{U} \end{bmatrix}. \tag{5–22}$$

The vectorized function $\mathbf{F} : \mathbb{R}^{n_x \times N} \to \mathbb{R}^{m_f \times N}$, $m_f = 1 + n_y + m_c$, is then constructed as

$$\begin{aligned} \mathbf{F}(\mathbf{X}) &= \begin{bmatrix} \mathbf{f}(\mathbf{X}_1) & \mathbf{f}(\mathbf{X}_2) & \cdots & \mathbf{f}(\mathbf{X}_N) \end{bmatrix} \\ &= \begin{bmatrix} g(\mathbf{Y}_1, \mathbf{U}_1) & g(\mathbf{Y}_2, \mathbf{U}_2) & \cdots & g(\mathbf{Y}_N, \mathbf{U}_N) \\ \mathbf{a}(\mathbf{Y}_1, \mathbf{U}_1) & \mathbf{a}(\mathbf{Y}_2, \mathbf{U}_2) & \cdots & \mathbf{a}(\mathbf{Y}_N, \mathbf{U}_N) \\ \mathbf{c}(\mathbf{Y}_1, \mathbf{U}_1) & \mathbf{c}(\mathbf{Y}_2, \mathbf{U}_2) & \cdots & \mathbf{c}(\mathbf{Y}_N, \mathbf{U}_N) \end{bmatrix}, \end{aligned} \tag{5–23}$$

which has the corresponding continuous function, $\mathbf{f} : \mathbb{R}^{n_x} \to \mathbb{R}^{m_f}$,

$$\mathbf{f}(\mathbf{x}(\tau)) = \begin{bmatrix} g(\mathbf{y}(\tau), \mathbf{u}(\tau)) \\ \mathbf{a}(\mathbf{y}(\tau), \mathbf{u}(\tau)) \\ \mathbf{c}(\mathbf{y}(\tau), \mathbf{u}(\tau)) \end{bmatrix}. \tag{5–24}$$

Thus, the first and second derivatives of the NLP of Eqs. (5–15)–(5–18) may be computed given the derivatives of the functions $\boldsymbol{\psi}(\mathbf{v})$ and $\mathbf{F}(\mathbf{X})$ together with the values of $\mathbf{D}$, $\mathbf{B}$, and $\mathbf{w}$, as defined by the given collocation scheme. Moreover, by performing the separation, the vectorized nature of the function $\mathbf{F}(\mathbf{X})$ may be fully exploited by performing analysis on the derivatives of the smaller dimensional function $\mathbf{f}(\mathbf{x})$.

5.4 Compressibility of Vectorized Problems

As discussed in Section 3.6, vectorized functions of the form of Eq. (5–23) have inherently sparse derivative structures of which should be taken advantage. The method of this dissertation differentiates vectorized functions (i.e. $\mathbf{F}(\mathbf{X})$) by statically exploiting sparsity of the corresponding continuous functions (i.e. $\mathbf{f}(\mathbf{x})$) and printing vectorized derivative procedures. When computing $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$ via sparse finite differences or an AD tool which utilizes matrix compression, one may take advantage of the compressible nature of $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$ by performing an a priori analysis on the structure of the Jacobian $\nabla_{\mathbf{x}}\mathbf{f}$. To illustrate, suppose a suitable Curtis-Powell-Reid (CPR) seed matrix $\mathbf{S} \in \mathbb{Z}^{n_x \times q_x}$ $(q_x \leq n_x)$ is found such that $\nabla_{\mathbf{x}}\mathbf{f}$ is fully recoverable from $[\nabla_{\mathbf{x}}\mathbf{f}]\,\mathbf{S}$. Given the seed matrix $\mathbf{S}$, one may then construct a seed matrix, $\bar{\mathbf{S}}$, for the vectorized problem such that

$$\bar{\mathbf{S}} = \begin{bmatrix} \mathbf{S} \\ \mathbf{S} \\ \vdots \\ \mathbf{S} \end{bmatrix} \in \mathbb{R}^{n_x N \times q_x} \tag{5–25}$$

where the matrix $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger \in \mathbb{R}^{m_f N \times n_x N}$ is fully recoverable from the matrix

$$\left[\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger\right]\bar{\mathbf{S}} = \begin{bmatrix} [\nabla_{\mathbf{X}_1}\mathbf{F}_1]\,\mathbf{S} \\ [\nabla_{\mathbf{X}_2}\mathbf{F}_2]\,\mathbf{S} \\ \vdots \\ [\nabla_{\mathbf{X}_N}\mathbf{F}_N]\,\mathbf{S} \end{bmatrix} \in \mathbb{R}^{m_f N \times q_x}. \tag{5–26}$$

Thus, if utilizing a sparse forward finite difference and a fixed step-size to approximate $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$, one would only need to perform $q_x + 1$ function evaluations, regardless of the value of N. Similarly, $\nabla_{\mathbf{X}^\dagger}\mathbf{F}^\dagger$ may be found via AD by computing q_x directional derivatives, regardless of the value of N. Comparable exploitations may also be performed at the second derivative level by making use of a known sparse structure of the Hessian $\nabla^2_{\mathbf{xx}}\boldsymbol{\lambda}^\mathsf{T}\mathbf{f}$ ($\boldsymbol{\lambda} \in \mathbb{R}^{m_f}$).

5.5 Examples

In this section the developed method of this dissertation is used to compute the first and second derivatives of the functions $\mathbf{F}(\mathbf{X})$ and $\boldsymbol{\psi}(\mathbf{v})$ in order to solve two optimal control problems using the GPOPS-II optimal control software. The GPOPS-II optimal control software transcribes the continuous time Bolza optimal control problem into an NLP of the form presented in Section 5.2. The NLP is then solved on an initial mesh, integration tolerances are inspected, and if not met, the NLP is solved on a new mesh. This process is known as mesh refinement and is repeated until integration (mesh) tolerances are met. Each new mesh effectively changes the values and dimensions of the matrices $\mathbf{D}$ and $\mathbf{B}$ of Eq. (5–16), the quadrature weights $\mathbf{w}$ of Eq. (5–15), and the number of collocation points, N. Thus, by utilizing the ADiGator tool to differentiate $\boldsymbol{\psi}(\mathbf{v})$ and $\mathbf{F}(\mathbf{X})$ for fixed values of n_v and n_x (but non-fixed value of N), the resulting derivative files are valid for any mesh. The efficiency of the approach is investigated by comparing derivative computation times against other well known MATLAB algorithmic differentiation tools.

All problems were solved using GPOPS-II with the Radau integration collocation scheme and the hp-adaptive mesh refinement method of Ref. [45]. Each NLP constructed by GPOPS-II is solved via the open-source NLP solver IPOPT [60] in conjunction with the linear solver MA57 [20]. Moreover, for each problem, the relative NLP tolerance is set to $\epsilon_{\text{NLP}} = \sqrt{\epsilon_{\text{machine}}}$ (where $\epsilon_{\text{machine}} \approx 2.22 \times 10^{-16}$ is the machine precision), and the relative mesh (integration) tolerance is set to $\epsilon_{\text{mesh}} = 10\epsilon_{\text{NLP}}$. All computations were

performed on an Apple MacBook Pro with Mac OS-X 10.10.2 (Yosemite) and a 2.3 GHz Intel Core 17 processor with 16 GB 1066 MHz DDR3 RAM using MATLAB version R2014b.

5.5.1 Example 1: Low Thrust Orbit Transfer

The example of this section is a low-thrust orbital transfer optimal control problem taken from Ref. [7]. This example is an ideal candidate for applying algorithmic differentiation due to the complex dynamics of the system. This example is divided into two parts; the problem formulation and analysis of results.

5.5.1.1 Problem formulation

The state of the system is given in modified equinoctial elements while the control is given in radial-transverse-normal coordinates. The goal is to determine the state

$$\left[\begin{array}{cc} \mathbf{y}^{\mathsf{T}} & w \end{array}\right] = \left[\begin{array}{ccccccc} p & f & g & h & k & L & w \end{array}\right]^{\mathsf{T}}, \tag{5–27}$$

the control

$$\mathbf{u} = \left[\begin{array}{ccc} u_r & u_\theta & u_h \end{array}\right]^{\mathsf{T}}, \tag{5–28}$$

and the throttle parameter, τ, that transfer the spacecraft from an initial orbit to a final orbit while maximizing the final weight of the spacecraft. The spacecraft starts in a circular low-Earth orbit with inclination $i(t_0) = 28.5\,\text{deg}$ and terminates in a highly elliptic low periapsis orbit with inclination $i(t_f) = 63.4\,\text{deg}$. The continuous-time optimal control problem corresponding to this orbital transfer problem can be stated in Mayer form as follows. Minimize the cost functional

$$J = -w(t_f) \tag{5–29}$$

subject to the dynamic constraints

$$\begin{aligned} \dot{\mathbf{y}} &= \mathbf{A}(\mathbf{y})\boldsymbol{\Delta} + \mathbf{b}, \\ \dot{w} &= -\frac{T(1+0.01\tau)}{I_{sp}}, \end{aligned} \tag{5–30}$$

the path constraint

$$||\mathbf{u}|| = 1, \tag{5–31}$$

the parameter constraint

$$-50 \leq \tau \leq 0, \tag{5–32}$$

and the boundary conditions

$$\begin{aligned}
&p(t_0) = 21837080.052835 \text{ ft}, && p(t_f) = 40007346.015232 \text{ ft}, \\
&f(t_0) = 0, && \sqrt{f^2(t_f) + g^2(t_f)} = 0.73550320568829, \\
&g(t_0) = 0, && \sqrt{h^2(t_f) + k^2(t_f)} = 0.61761258786099, \\
&h(t_0) = -0.25396764647494, && f(t_f)h(t_f) + g(t_f)k(t_f) = 0, \\
&k(t_0) = 0, && g(t_f)h(t_f) - k(t_f)f(t_f) \leq 0, \\
&L(t_0) = \pi \text{ rad}, && w(t_0) = 1 \text{ lbm}, \\
&i(t_0) = 28.5 \text{ deg}, && i(t_f) = 63.4 \text{ deg}.
\end{aligned} \tag{5–33}$$

The matrix $\mathbf{A}(\mathbf{y})$ in Eq. (5–30) is given as

$$\mathbf{A} = \begin{bmatrix}
0 & \frac{2p}{q}\sqrt{\frac{p}{\mu}} & 0 \\
\sqrt{\frac{p}{\mu}}\sin(L) & \sqrt{\frac{p}{\mu}}\frac{1}{q}\Big((q+1)\cos(L) + f\Big) & -\sqrt{\frac{p}{\mu}}\frac{g}{q}\Big(h\sin(L) - k\cos(L)\Big) \\
-\sqrt{\frac{p}{\mu}}\cos(L) & \sqrt{\frac{p}{\mu}}\frac{1}{q}\Big((q+1)\sin(L) + g\Big) & \sqrt{\frac{p}{\mu}}\frac{f}{q}\Big(h\sin(L) - k\cos(L)\Big) \\
0 & 0 & \sqrt{\frac{p}{\mu}}\frac{s^2\cos(L)}{2q} \\
0 & 0 & \sqrt{\frac{p}{\mu}}\frac{s^2\sin(L)}{2q} \\
0 & 0 & \sqrt{\frac{p}{\mu}}\Big(h\sin(L) - k\cos(L)\Big)
\end{bmatrix} \tag{5–34}$$

while the vector is

$$\mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \sqrt{\mu p}(\frac{q}{p})^2 \end{bmatrix}, \tag{5–35}$$

where

$$\begin{aligned} &q = 1 + f\cos(L) + g\sin(L), && r = p/q, \\ &\alpha^2 = h^2 - k^2, && \chi = \sqrt{h^2 + k^2}, \\ &s^2 = 1 + \chi^2. \end{aligned} \tag{5–36}$$

The spacecraft acceleration is modeled as

$$\boldsymbol{\Delta} = \boldsymbol{\Delta}_g + \boldsymbol{\Delta}_T, \tag{5–37}$$

where $\boldsymbol{\Delta}_g$ is the acceleration due to the oblateness of the Earth while $\boldsymbol{\Delta}_T$ is the thrust specific force. The acceleration due to Earth oblateness is expressed in rotating radial coordinates as

$$\boldsymbol{\Delta}_g = \mathbf{Q}_r^{\mathsf{T}} \delta\mathbf{g}, \tag{5–38}$$

where $\mathbf{Q}_r$ is the transformation from rotating radial coordinates to Earth centered inertial coordinates. The matrix $\mathbf{Q}_r$ is given column-wise as

$$\mathbf{Q}_r = \begin{bmatrix} \mathbf{i}_r & \mathbf{i}_\theta & \mathbf{i}_h \end{bmatrix}, \tag{5–39}$$

where the basis vectors $\mathbf{i}_r$, $\mathbf{i}_\theta$, and $\mathbf{i}_h$ are given as

$$\mathbf{i}_r = \frac{\mathbf{r}}{\|\mathbf{r}\|} \quad , \quad \mathbf{i}_h = \frac{\mathbf{r} \times \mathbf{v}}{||\mathbf{r} \times \mathbf{v}||} \quad , \quad \mathbf{i}_\theta = \mathbf{i}_h \times \mathbf{i}_r. \tag{5–40}$$

Furthermore, the vector $\delta\mathbf{g}$ is defined as

$$\delta\mathbf{g} = \delta g_n \mathbf{i}_n - \delta g_r \mathbf{i}_r, \tag{5–41}$$

where $\mathbf{i}_n$ is the local North direction and is defined as

$$\mathbf{i}_n = \frac{\mathbf{e}_n - (\mathbf{e}_n^\mathsf{T}\mathbf{i}_r)\mathbf{i}_r}{||\mathbf{e}_n - (\mathbf{e}_n^\mathsf{T}\mathbf{i}_r)\mathbf{i}_r||} \tag{5–42}$$

and $\mathbf{e}_n = (0, 0, 1)$. The oblate earth perturbations are then expressed as

$$\delta g_r = -\frac{\mu}{r^2}\sum_{k=2}^{4}(k+1)\left(\frac{R_e}{r}\right)^k P_k(s)J_k, \tag{5–43}$$

$$\delta g_n = -\frac{\mu\cos(\phi)}{r^2}\sum_{k=2}^{4}\left(\frac{R_e}{r}\right)^k P_k'(s)J_k, \tag{5–44}$$

where R_e is the equatorial radius of the earth, $P_k(s)$ ($s \in [-1, +1]$) is the k^{th}-degree Legendre polynomial, P_k' is the derivative of P_k with respect to s, $s = \sin(\phi)$, and J_k represents the zonal harmonic coefficients for $k = (2, 3, 4)$. Next, the acceleration due to thrust is given as

$$\mathbf{\Delta}_T = \frac{g_0 T(1 + 0.01\tau)}{w}\mathbf{u}. \tag{5–45}$$

Finally, the physical constants used in the problem are given as

$$\begin{aligned} &I_{sp} = 450 \text{ s}, && T = 4.446618 \times 10^{-3} \text{ lbf}, \\ &g_0 = 32.174 \text{ ft/s}^2, && \mu = 1.407645794 \times 10^{16} \text{ ft}^3/\text{s}^2, \\ &R_e = 20925662.73 \text{ ft}, && J_2 = 1082.639 \times 10^{-6}, \\ &J_3 = -2.565 \times 10^{-6}, && J_4 = -1.608 \times 10^{-6}. \end{aligned} \tag{5–46}$$

5.5.1.2 Results

Given the problem formulation of the previous section, the continuous and endpoint variables, $\mathbf{x}(t)$ and $\mathbf{v}$, are now identified as

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{y}^\mathsf{T}(t) & \mathbf{u}^\mathsf{T}(t) & \tau(t) \end{bmatrix}^\mathsf{T} \in \mathbb{R}^{11} \tag{5–47}$$

and

$$\mathbf{v} = \begin{bmatrix} \mathbf{y}^\mathsf{T}(t_0) & \mathbf{y}^\mathsf{T}(t_f) & t_0 & t_f & \tau \end{bmatrix}^\mathsf{T} \in \mathbb{R}^{17}, \tag{5–48}$$

where $\tau(t) = \tau \ \forall t$. The continuous function $\mathbf{f} : \mathbb{R}^{11} \rightarrow \mathbb{R}^8$ is then given by combining the dynamics of Eq. (5–30) and the path constraint of Eq. (5–31) such that

$$\mathbf{f}(\mathbf{x}(t)) = \begin{bmatrix} \mathbf{A}(\mathbf{y})\boldsymbol{\Delta} + \mathbf{b} \\ -\dfrac{T(1+0.01\tau)}{I_{sp}} \\ ||\mathbf{u}|| - 1 \end{bmatrix} \in \mathbb{R}^8. \tag{5–49}$$

Similarly, the endpoint function $\boldsymbol{\psi} : \mathbb{R}^{17} \rightarrow \mathbb{R}^5$ is given by combining the cost function of Eq. (5–29) together with the non-trivial boundary conditions of Eq. (5–33) such that

$$\boldsymbol{\psi}(\mathbf{v}) = \begin{bmatrix} -w(t_f) \\ \sqrt{y_1^2(t_f) + y_2^2(t_f)} \\ \sqrt{y_3^2(t_f) + y_4^2(t_f)} \\ y_1(t_f)y_3(t_f) + y_2(t_f)y_4(t_f) \\ y_2(t_f)y_3(t_f) + y_4(t_f)y_1(t_f) \end{bmatrix} \in \mathbb{R}^5. \tag{5–50}$$

The manner in which the initial guess for this example is generated, the MATLAB code corresponding to the functions $\mathbf{F}(\mathbf{X})$ and $\boldsymbol{\psi}(\mathbf{v})$, and the solution produced by GPOPS-II for this problem may be seen in Ref. [47]. The problem was solved using IPOPT in both the Quasi-Newton (first-order) and full Newton (second-order) modes. Computational results of the solutions for the first- and second-order modes are shown in Tables 5-1 and 5-2, respectively. Within Tables 5-1 and 5-2, it is seen that the NLP must be solved on ten different meshes, where the vectorized dimension N ranges from the initial value of 32 to a final value of 587. When comparing the two results, it is seen that by utilizing a second-order approach, the solution may be obtained in 68.04s versus the 221.74s required by the first-order approach. This increase in performance is primarily due to the decrease in required number of NLP iterations as a result of the quadratic convergence property of the full-Newton method. Upon further inspection of Table 5-1, it is also seen that only a small portion (roughly one tenth) of the NLP computation time is spent performing derivative computations using the Quasi-Newton approach. Conversely,

when utilizing the second-order approach, roughly half of the computation time is spent in derivative calculation.

Table 5-1. Low-thrust orbit transfer - GPOPS-II with IPOPT quasi-Newton CPU times.

Mesh #:	1	2	3	4	5	6	7	8	9	10	Total
N:	32	104	296	461	527	551	568	574	584	587	-
CPU Time Spent in NLP Derivative Computation (s)											
NLP:	2.077	4.038	18.595	19.121	31.940	20.452	17.788	65.376	19.021	23.331	221.74
$\nabla\mathbf{F}$:	0.594	0.874	2.527	2.003	2.926	1.763	1.563	5.615	1.653	2.009	21.53
$\nabla\boldsymbol{\psi}$:	0.040	0.051	0.109	0.070	0.097	0.057	0.049	0.177	0.051	0.062	0.76
Number of Derivative Evaluations											
$\nabla\mathbf{F}$:	130	158	334	213	288	169	148	526	153	185	2304
$\nabla\boldsymbol{\psi}$:	260	316	668	426	576	338	296	1052	306	370	4608

Table 5-2. Low-thrust orbit transfer - GPOPS-II with IPOPT full Newton CPU times.

Mesh #:	1	2	3	4	5	6	7	8	9	10	Total
N:	32	104	296	461	527	551	568	574	584	587	-
CPU Time Spent in NLP Derivative Computation (s)											
NLP:	1.373	3.167	3.559	3.512	10.980	7.084	8.840	9.559	10.274	9.695	68.04
$\nabla\mathbf{F}$:	0.138	0.301	0.220	0.169	0.487	0.315	0.389	0.419	0.453	0.422	3.31
$\nabla^2\mathbf{F}$:	0.488	1.305	1.422	1.486	4.721	2.982	3.813	4.069	4.496	4.141	28.92
$\nabla\boldsymbol{\psi}$:	0.009	0.018	0.010	0.006	0.016	0.010	0.012	0.013	0.014	0.013	0.12
$\nabla^2\boldsymbol{\psi}$:	0.009	0.017	0.009	0.006	0.016	0.010	0.012	0.013	0.014	0.013	0.12
Number of Derivative Evaluations											
$\nabla\mathbf{F}$:	30	55	29	18	48	30	37	39	42	39	367
$\nabla^2\mathbf{F}$:	28	53	27	16	46	28	35	37	40	37	347
$\nabla\boldsymbol{\psi}$:	60	110	58	36	96	60	74	78	84	78	734
$\nabla^2\boldsymbol{\psi}$:	28	53	27	16	46	28	35	37	40	37	347

As seen in Tables 5-1 and 5-2, the derivative CPU times of the endpoint function $\boldsymbol{\psi}(\mathbf{v})$ are largely irrelevant when compared to those of the vectorized function $\mathbf{F}(\mathbf{X})$. Thus in order to investigate the efficiency of the approach the vectorized function $\mathbf{F}(\mathbf{X})$ is focused upon. The complexity of the continuous function of Eq. 5–49 is now emphasized by investigating the density of the continuous Jacobian $\nabla\mathbf{f}(\mathbf{x}) \in \mathbb{R}^{8\times 11}$ and Hessian $\nabla^2\boldsymbol{\lambda}^{\mathsf{T}}\mathbf{f}(\mathbf{x}) \in \mathbb{R}^{11\times 11}$. The sparsity structures of the continuous Jacobian and Hessian, as

computed by ADiGator, are given by

$$\text{struct}(\nabla \mathbf{f}(\mathbf{x})) = \left[\begin{array}{cccccccccc} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet & \bullet \\ & & & & & & & & & \bullet \\ & & & & & & \bullet & \bullet & \bullet & \end{array}\right], \tag{5–51}$$

and

$$\text{struct}(\nabla^2 \boldsymbol{\lambda}^{\mathsf{T}} \mathbf{f}(\mathbf{x})) = \left[\begin{array}{cccccccccccc} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & & & & & \bullet & \bullet & \bullet & & & \bullet \\ \bullet & \bullet & \bullet & & & \bullet & \bullet & & \bullet & & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \end{array}\right], \tag{5–52}$$

respectively. The continuous Jacobian is thus incompressible and the vectorized Jacobian $\nabla \mathbf{F}(\mathbf{X})$ is compressible with a seed matrix of column dimension 11.

The comparative efficiency of the ADiGator tool is now explored by computing the first and second derivatives of $\mathbf{F}(\mathbf{X})$ using a variety of MATLAB algorithmic differentiation tools, at different values of N. Figure 5-1 shows the Jacobian to function ratios obtained by computing $\nabla \mathbf{F}$ using the ADiGator, INTLAB, MAD, and ADiMat tools, together with the theoretical minimum ratio required by a single-step finite difference. As may be seen in Fig. 5-1, ratios are given for ADiGator using both the vectorized (as was used to solve the above optimal control problem) and non-vectorized modes, where, in the case of the non-vectorized mode, a new derivative file must be generated for each value of N. Results were obtained utilizing the INTLAB gradient class, the compressed forward mode of MAD, and the scalar mode of ADiMat (by computing the 11 required directional derivatives independently). As a frame of reference, the theoretical ratio for a forward-difference is also given as $n_x + 1 = 12$. From the figure it is seen that

the ADiGator tool in the vectorized mode performs better than a theoretical forward-step finite difference while maintaining the accuracy of an AD tool. In the non-vectorized mode, it is also seen that the tool is rather efficient, however, as the value of N increases, run-time overheads due to array bounds checks begin to dominate the computation time.

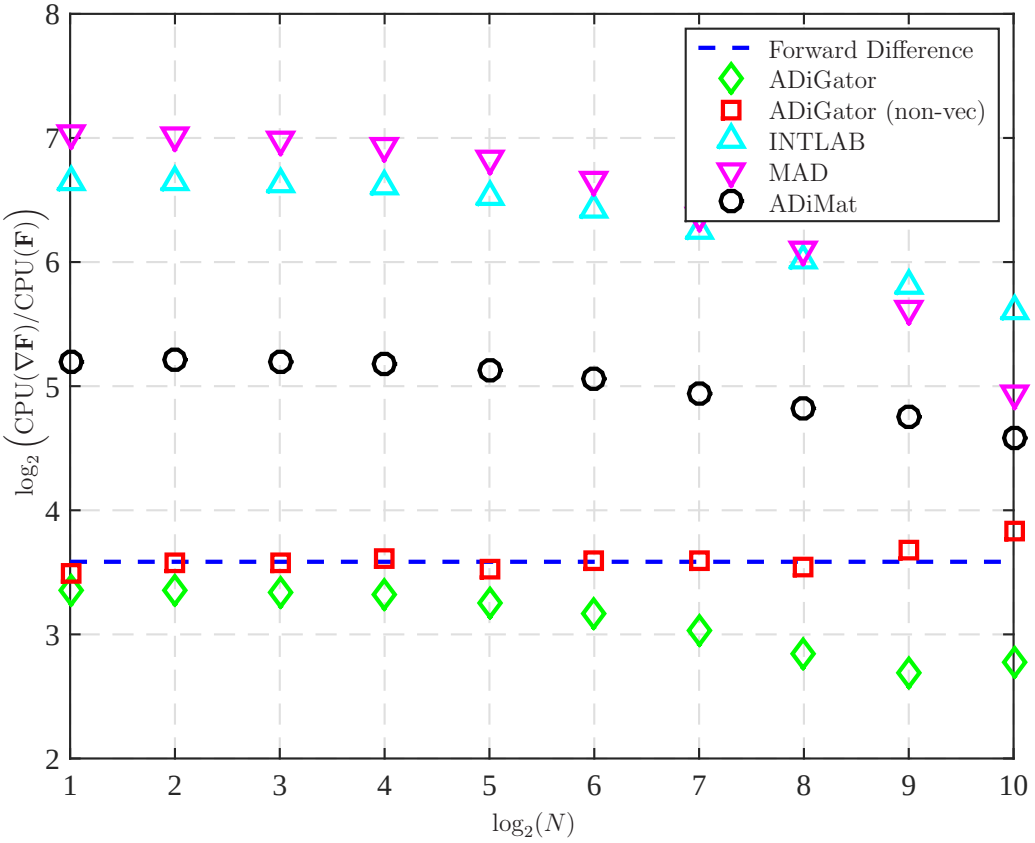


Figure 5-1. Low-thrust orbital transfer Jacobian to function CPU ratios.

Figure 5-2 shows the Hessian to function ratios obtained by computing $\nabla^2\mathbf{F}$ using the ADiGator, INTLAB, and MAD tools, as well as the theoretical minimum ratio required to approximate $\nabla^2\mathbf{F}$ via a finite-difference. As was the case with the first-derivative comparison, ratios are given for ADiGator using both the vectorized and non-vectorized modes, where new derivative files are generated for each value of N when using the non-vectorized mode. Additional results were obtained using the INTLAB hessian class and the compressed forward-over-forward mode of MAD. Moreover, as a frame of reference, the theoretical minimum ratio required to compute $\nabla^2\mathbf{F}$ via a finite-difference

is given as $2n_x + 1 = 23$. From these results it is seen that again the vectorized and non-vectorized modes of ADiGator diverge as the value of N increases and indexing run-time overheads dominate computation time of the vectorized mode.

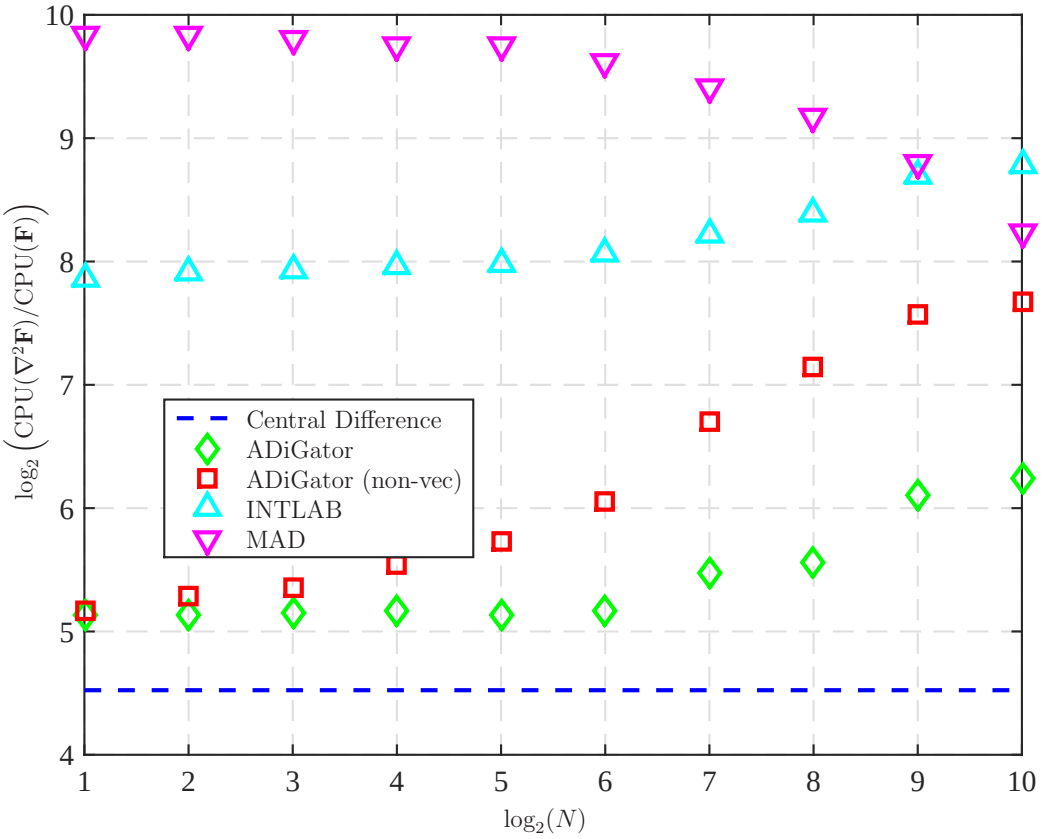


Figure 5-2. Low-thrust orbital transfer Hessian to function CPU ratios.

Finally, in Table 5-3, the file generation times are given for the ADiGator algorithm using both the vectorized and non-vectorized modes (with N varying for the non-vectorized case). From this table it may be seen that the initial overhead required to generate derivative files in the vectorized mode is relatively insignificant, particularly when compared to the time required to solve the optimal control problem. It may also be seen that if the non-vectorized mode was to be be used, the derivative file generation overhead would be magnified, as new files would need to be created for each new mesh.

Table 5-3. Low-thrust orbital transfer vectorized derivative file generation times. First derivative files corresponding to $\nabla\mathbf{F}$ generated by applying ADiGator to the function which computes $\mathbf{F}$. Second derivative files corresponding to $\nabla^2\mathbf{F}$ generated by applying ADiGator to the previously generated first derivative file. The column of $N = \mathbb{Z}_+$ denotes the use of ADiGator in the vectorized mode.

N:	$\mathbb{Z}_+$	2	4	8	16	32	64	128	256	512	1024
ADiGator File Generation Time (s)											
$\nabla\mathbf{F}$:	0.44	0.44	0.43	0.44	0.44	0.45	0.46	0.49	0.61	0.98	2.92
$\nabla^2\mathbf{F}$:	3.51	3.56	3.58	3.62	3.69	3.80	3.98	4.38	5.40	7.68	12.95

5.5.2 Example 2: Minimum Time to Climb of Supersonic Aircraft

The example of this section is the classical minimum time to climb of a supersonic aircraft. The objective is to determine the minimum-time trajectory and control from take-off to a specified altitude and speed. The problem was originally formulated in Ref. [54], however, the formulation used here is a slightly modified version of that of Ref. [18].

5.5.2.1 Problem formulation

The minimum time-to-climb problem for a supersonic aircraft is posed as follows. The state vector is composed of the aircraft altitude, speed, and flight path angle and the control is the load factor. The goal is to determine the state $\mathbf{y}(t) \in \mathbb{R}^3$ and the control $u(t) \in \mathbb{R}$ which minimize the cost functional

$$J = t_f \tag{5–53}$$

subject to the dynamic constraints

$$\dot{\mathbf{x}} = \begin{bmatrix} y_2 \sin(y_3) \\ \frac{F_t - F_d}{m} - g\sin(y_3) \\ g\frac{u - \cos(y_3)}{y_2} \end{bmatrix}, \tag{5–54}$$

the path constraints

$$x_1 \geq 0, \tag{5–55}$$

$$\frac{\pi}{2} \leq x_3 \leq \frac{\pi}{2}, \tag{5–56}$$

and the boundary conditions

$$\begin{aligned} y_1(t_0) &= 0 \text{ m}, & y_1(t_0) &= 19994.88 \text{ m}, \\ y_2(t_0) &= 129.314448 \text{ m/s}, & y_2(t_0) &= 295.0915104 \text{ m/s}, \\ y_3(t_0) &= 0 \text{ rad}, & y_3(t_0) &= 0 \text{ rad}, \end{aligned} \tag{5–57}$$

where m is the mass of the aircraft, g is acceleration due to gravity, F_t is the magnitude of the thrust force, and F_d is the magnitude of the drag force. Moreover, the thrust and drag forces are given by

$$\begin{aligned} F_t &= F_t(T(y_1), y_1, y_2), \\ F_d &= F_d(T(y_1), \rho(y_1), y_1, y_2), \end{aligned} \tag{5–58}$$

where the temperature T and atmospheric density ρ are modeled by the piece-wise continuous functions take from Ref. [41]:

$$\rho(y_1) = \frac{p(y_1)}{0.2869(T(y_1) + 273.1)} \tag{5–59}$$

where

$$\Big(T(y_1),\ p(y_1)\Big) = \begin{cases} \left(15.04 - 0.00649y_1,\ 101.29\left(\frac{T(y_1)+273.1}{288.08}\right)^{5.256}\right) & , \quad y_1 < 11000, \\ \left(-56.46,\ 22.65e^{(1.73-0.000157y_1)}\right) & , \quad \text{otherwise.} \end{cases} \tag{5–60}$$

5.5.2.2 Results

From the problem formulation of the previous section, the continuous and endpoint variables, $\mathbf{x}(t)$ and $\mathbf{v}$, are now identified as

$$\mathbf{x}(t) = \begin{bmatrix} \mathbf{y}^\mathsf{T}(t) & u \end{bmatrix} \in \mathbb{R}^4 \tag{5–61}$$

and

$$\mathbf{v} = \begin{bmatrix} \mathbf{y}^\mathsf{T}(t_0) & \mathbf{y}^\mathsf{T}(t_f) & t_0 & t_f \end{bmatrix} \in \mathbb{R}^8. \tag{5–62}$$

The continuous function $\mathbf{f} : \mathbb{R}^4 \to \mathbb{R}^3$ is then simply given by the dynamics of Eq. 5–54, and the endpoint function, $\boldsymbol{\psi} : \mathbb{R}^8 \to \mathbb{R}$ is the trivial function of Eq. 5–53. In order to

showcase the elegance of the ADiGator generated derivative code, the function code which computes $\mathbf{F}(\mathbf{X})$ and the ADiGator generated derivative code which computes $\nabla\mathbf{F}(\mathbf{X})$ are shown in the Appendix. As with the previous low-thrust orbital transfer problem, the minimum time to climb example was once again solved using GPOPS-II and the NLP solver IPOPT. Unlike the previous example, a solution was unable to be obtained using IPOPT in the Quasi-Newton mode, and thus results are only presented for the solution obtained using IPOPT in the full-Newton mode. Computational results of the solution to the optimal control problem are given in Table 5-4, where it is seen that 22 mesh refinements were required in order to identify the discontinuities presented by Eqs. (5–55) and (5–60) and obtain a suitable solution. Moreover, the vectorized dimension N ranges from the initial value of 40 to the final value of 152.

As with the previous example, the efficiency of the approach is now investigated by computing the first and second derivatives of $\mathbf{F}(\mathbf{X})$ using a variety of well-known MATLAB AD tools at differing values of N. Jacobian to function computation time ratios are given in Fig. 5-3 for the ADiGator, ADiMat, MAD, and INTLAB tools. Results are presented for ADiGator in both the vectorized and non-vectorized modes, the ADiMat tool was used in the scalar mode by computing the $n_x = 4$ required directional derivatives independently, the INTLAB tool was used with the gradient class, and the MAD tool was used in the compressed forward mode with a seed matrix of column dimension $n_x = 4$. Additionally, the benchmark ratio for a forward step finite difference is shown. Results were similarly obtained at the second derivative level using ADiGator in both the vectorized and non-vectorized modes, the INTLAB tool's hessian class, and the forward over forward compressed mode of MAD. The resulting Hessian to function computation time ratios may be seen in Fig. 5-4.

5.6 Discussion

The solution of optimal control problems via direct collocation presents an interesting test case for algorithmic differentiation due in part to the inherently iterative nature of

Table 5-4. Minimum time to climb - GPOPS-II with IPOPT full Newton CPU times.

Mesh #:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	Total
N:	40	85	104	119	126	129	131	133	134	136	138	140	141	142	144	145	146	147	148	149	150	152	-
CPU Time Spent in NLP Derivative Computation (s)																							
NLP:	0.661	0.217	0.290	0.327	0.305	0.256	0.707	0.691	0.689	0.658	0.641	0.645	0.663	0.661	0.657	0.661	0.690	0.664	0.709	0.666	0.680	0.670	12.81
$\nabla\mathbf{F}$:	0.087	0.035	0.044	0.048	0.045	0.037	0.104	0.099	0.101	0.095	0.094	0.094	0.095	0.095	0.095	0.096	0.101	0.096	0.102	0.095	0.096	0.098	1.85
$\nabla^2\mathbf{F}$:	0.135	0.082	0.111	0.122	0.113	0.093	0.274	0.268	0.269	0.256	0.249	0.251	0.255	0.256	0.256	0.256	0.271	0.256	0.277	0.254	0.256	0.257	4.82
Number of Derivative Evaluations																							
$\nabla\mathbf{F}$:	19	12	15	16	15	12	32	31	33	31	31	31	31	31	31	31	32	31	33	31	31	31	591
$\nabla^2\mathbf{F}$:	17	10	13	14	13	10	30	29	31	29	29	29	29	29	29	29	30	29	31	29	29	29	547

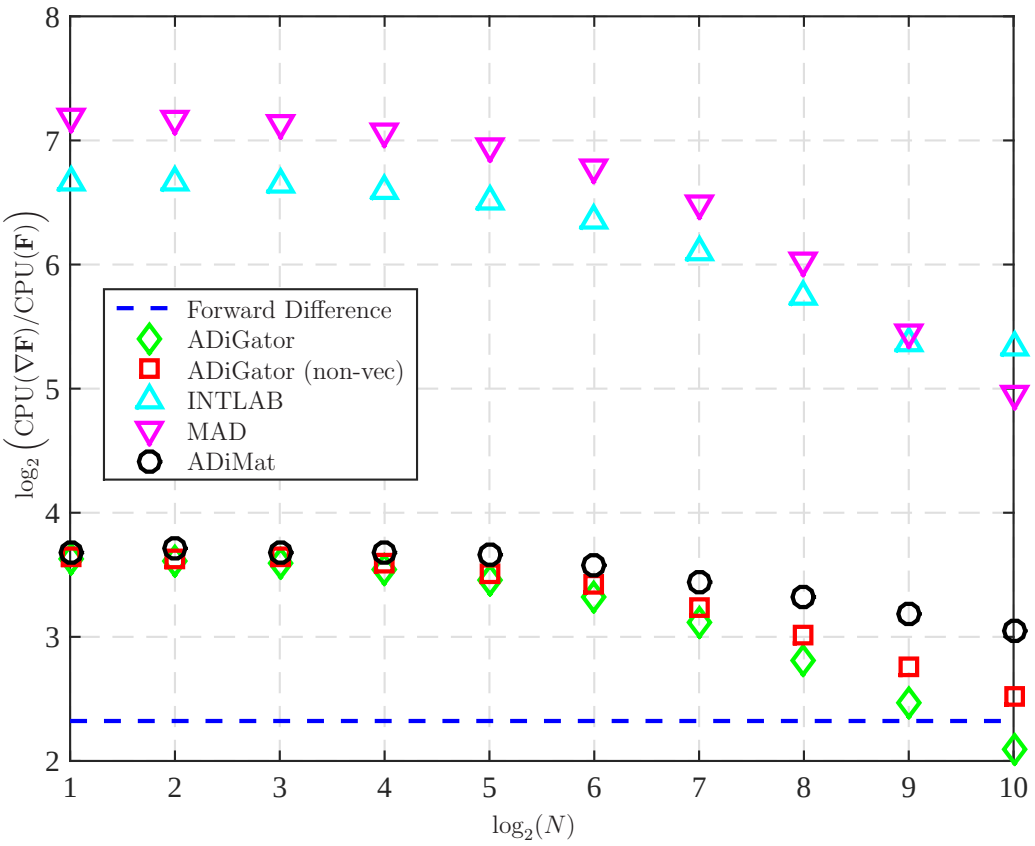


Figure 5-3. Minimum time to climb Jacobian to function CPU ratios.

the solution process. That is, numerous non-linear programs must be solved in repetition as mesh refinement is performed, where the non-linear programs themselves are also solved via iterative Newton based methods. Thus, in order to solve the aforementioned optimal control problems, a large number of derivative computations must be performed. Moreover, by taking advantage of discretization separability, an a priori analysis may be performed in order to determine the sparsity patterns of all future required derivatives.

The examples presented in Section 5.5 illustrate a number of key points. The first take-away is that utilizing a full-Newton NLP solver can be quite beneficial, pending the Lagrangian Hessian computations are performed in a reasonable amount of time. Moreover, it is seen from the two presented examples, that the efficiency of the solution method is largely dependent upon the ability to efficiently compute derivatives of vectorized functions of the form of $\mathbf{F} : \mathbb{R}^{n_x \times N} \to \mathbb{R}^{m_f \times N}$ of Eq. (5–23). The computation

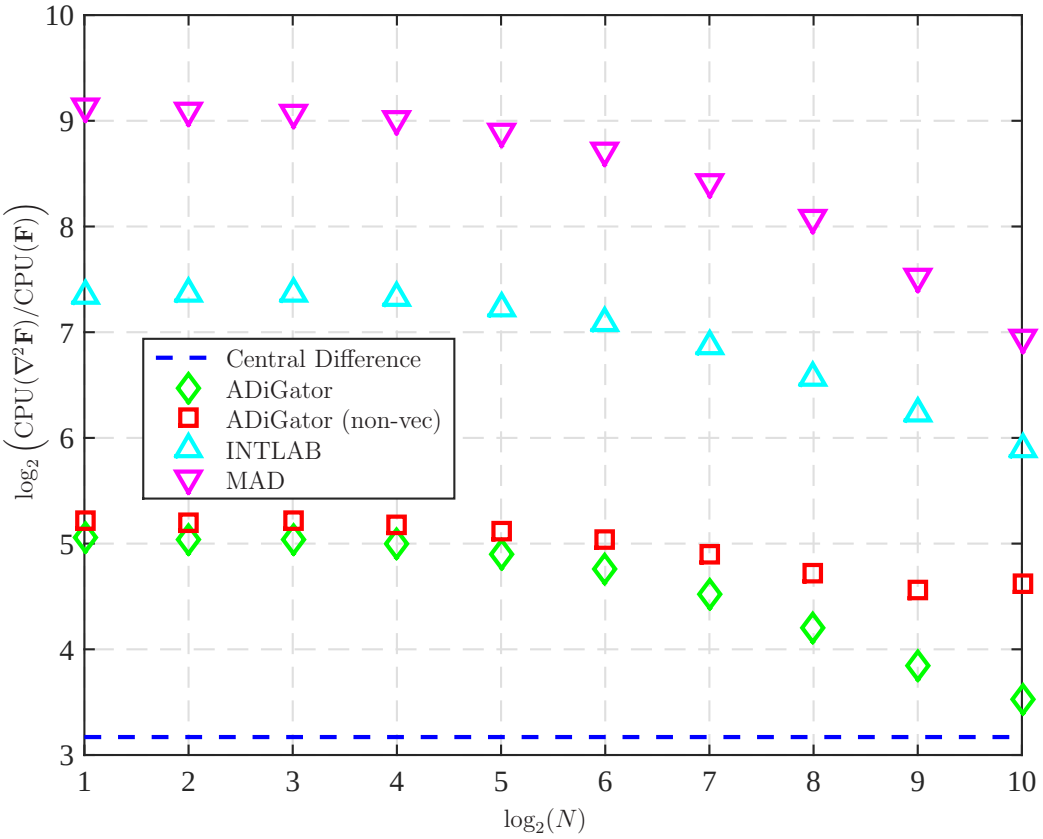


Figure 5-4. Minimum time to climb Hessian to function CPU ratios.

of the derivatives of such vectorized functions also presents an interesting test case. As seen in Section 5.4, the relative complexity of the differentiation of $\mathbf{F}(\mathbf{X})$ is dependent only upon the complexity of the derivatives of the continuous counterpart. This point is emphasized in the results of Figs. 5-1 and 5-3. Within these figures it is seen that the Jacobian to function CPU ratios decreases as N increases for all tested tools, with the exception being the ADiGator tool in the non-vectorized mode for the first example. The reason for this decreasing behavior is primarily due to the fact that the derivative computation overhead becomes relatively less significant as the dimension of the problem increases. For instance, the tools which utilize operator overloading at run-time, INTLAB and MAD, incur run-time overheads proportional to the number of MATLAB operations which are performed. These run-time overheads are primarily due to the type and shape checking/dispatch behavior discussed in Section 2.3. For the vectorized functions, however,

the number of MATLAB operations is constant for all values of N. Thus, as the value of N is increased, the significance of the run-time overhead penalties is decreased. The ADiMat tool suffers from similar run-time overheads proportional to both the number of MATLAB operations and the number of required directional derivatives, both of which are constant for all values of N. For this reason, it is also seen that the ADiMat Jacobian to function CPU ratios decrease as N increases. Moreover, it is seen that the ADiMat tool performs better on the smaller dimensional problem of Section 5.5.2. Unlike that of the other tested tools, the efficiency of the ADiGator tool is primarily dependent upon the number of non-zero derivatives at each link in the chain rule. Moreover, to exploit sparsity at each link in the chain rule, the ADiGator tool is reliant upon performing many index referencing and assignment operations, both of which are penalized by MATLAB's array bounds checking mechanism. The overhead associated with these indexing operations is thus proportional to the number of non-zero derivatives at each link in the chain rule. When using ADiGator in the vectorized mode, this overhead is constant for all values of N, as it is only dependent upon the smaller dimensional continuous problem. Thus, it is seen that at the first-derivative level, the Jacobian to function CPU ratios decrease with an increase in N. It is expected that this behavior would also be present at the second-derivative level, however, as seen by Fig. 5-2, this is not always the case. It is uncertain at this time why this behavior is exhibited for this test case. Unlike that of the vectorized mode, when using ADiGator in the non-vectorized mode, the indexing overhead is directly proportional to the value of N for functions of the vectorized nature. Moreover, any differences in run-times between the files produced by the vectorized and non-vectorized modes are strictly due this increase in indexing overhead. This point is emphasized by the results from Figures 5-1 – 5-4, where it is seen that for small values of N, computation times for the vectorized and non-vectorized modes are roughly equivalent. As N increases, however, the derivative CPU ratios begin to diverge and indexing run-time overheads begin to dominate the computation times of the vectorized

files. This difference is emphasized in Fig. 5-2, where at a value of $N = 1024$ well over half of the non-vectorized Hessian computation is spent performing indexing operations. Thus, by taking advantage of discretization separability and using ADiGator in the vectorized mode, not only can one generate files which are valid for any mesh, but one may also alleviate a great deal of indexing run-time penalties.

From the two example problems it is seen that the ADiGator tool out-performs all other tested tools for computing the derivatives of the vectorized problems. Moreover, when utilizing ADiGator in the vectorized mode, the times required to generate derivative files are largely insignificant. The developed tool is thus extremely well-suited for use with direct collocation optimal control. When comparing the efficiency of the approach to that of a finite-difference, a few points must be emphasized. The first point is that vectorized problems are extremely well suited for performing sparse finite-differences, due to their inherently compressible nature. The second point is that the theoretical ratios for obtaining finite-difference approximations are presented under the assumption that a fixed step size is to be used for each variable $X_{i,j}$, $(i = 1, \ldots, n_x,\ j = 1, \ldots, N)$ and that the step-size is easily computed. Moreover, when using a fixed step-size, derivative accuracy is highly dependent upon the scale of the problem. As a frame of reference, the GPOPS-II sparse finite-difference scheme, was used to compare derivative accuracy at the optimal solutions for the two tested problems, where it is assumed that the derivatives computed by the AD tool are correct to machine precision. The GPOPS-II finite difference scheme was used with the default fixed-step size of $10^{-6}|X_{i,j}|$ for each $X_{i,j}$ $(i = 1, \ldots, n_x,\ j = 1, \ldots, N)$ together with a forward difference approximation of $\nabla\mathbf{F}$ and a central difference approximation of $\nabla^2\mathbf{F}$. For the low thrust orbital transfer problem, the maximum relative errors in the approximations of $\nabla\mathbf{F}$ and $\nabla^2\mathbf{F}$ were found to be 6.01×10^{-2} and 7.29×10^{-2}, respectively. For the minimum time to climb problem, the maximum relative errors in the approximations of $\nabla\mathbf{F}$ and $\nabla^2\mathbf{F}$ were found to be 9.70×10^{-6} and 1.35×10^{-3}, respectively. Considering that the derivatives $\nabla\mathbf{F}$ form a part of the Karush Kuhn Tucker (KKT)

conditions [42] used to verify convergence of the NLP, it is clear that such first-derivative errors are unacceptable with $\epsilon_{NLP} \approx 1.49 \times 10^{-8}$. In order to obtain more accurate finite difference approximations, one could use a different step size for each element of the output. Ignoring the time required to compute the proper step-size, this would increase finite-difference times by at least a factor of m_f at the first derivative level and n_x^2 at the second-derivative level. Thus, while at times the ADiGator tool performs less efficiently than a theoretical sparse finite-difference, the alleviation of derivative accuracy concerns far outweighs the increase in computation time.

CHAPTER 6
ADDITIONAL TEST CASES

In this chapter, the developed AD tool is further tested by solving three different classes of problems. In Section 6.1, the developed algorithm is used to integrate an ordinary differential equation with a large, sparse Jacobian. In Section 6.2, a set of three fixed dimension non-linear system of equations problems are investigated, and in Section 6.3, a large sparse unconstrained minimization problem is presented. 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 $\times$ 2.4 GHz Quad-Core Intel Xeon processor with 24 GB 1066 MHz DDR3 RAM using MATLAB version R2014a.

6.1 Burgers' ODE

In this section the well-known Burgers' equation is solved using a moving mesh technique as presented in Ref. [32]. The form of Burgers' equation used for this example is given by:

$$\dot{u} = \alpha \frac{\partial^2 u}{\partial y^2} - \frac{\partial}{\partial y}\left(\frac{u^2}{2}\right), \quad 0 < y < 1,\ t > 0,\ \alpha = 10^{-4}, \tag{6–1}$$

with a smooth initial solution of

$$\begin{aligned} u(0,t) &= u(1,t) = 0, & t > 0, \\ u(x,0) &= \sin(2\pi y) + \tfrac{1}{2}\sin(\pi y), & 0 \leq 1. \end{aligned} \tag{6–2}$$

The partial differential equation (PDE) of Eq. (6–1) is then transformed into an 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

$$\mathbf{M}(t,x)\dot{\mathbf{x}} = \mathbf{f}(t,\mathbf{x}), \tag{6–3}$$

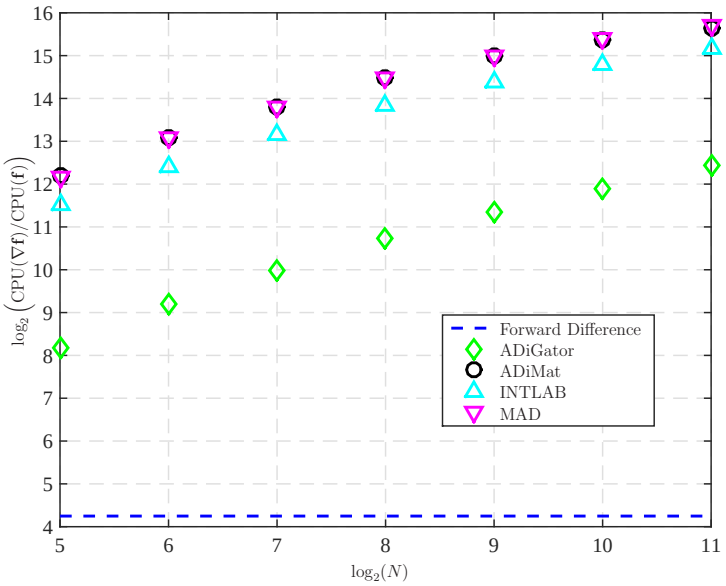
where $\mathbf{M} : \mathbb{R} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_x \times n_x}$ is a mass-matrix function and $\mathbf{f} : \mathbb{R} \times \mathbb{R}^{n_x} \to \mathbb{R}^{n_x}$ is the ODE function. This problem is given as an example problem for the MATLAB ODE suite and is solved with the stiff ODE solver, `ode15s` [55], which allows the user to supply the Jacobian $\nabla_{\mathbf{x}}\mathbf{f}(t, \mathbf{x})$.

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 file `burgersode`. The Jacobian $\nabla_{\mathbf{x}}\mathbf{f}(t, \mathbf{x})$ 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, as was the case with the vectorized examples of Section 5.5, the Jacobian $\nabla_{\mathbf{x}}\mathbf{f}(t, \mathbf{x})$ becomes increasingly sparser 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.[1] A test was then performed by applying AD to the resulting modified file. The results obtained by applying AD to both

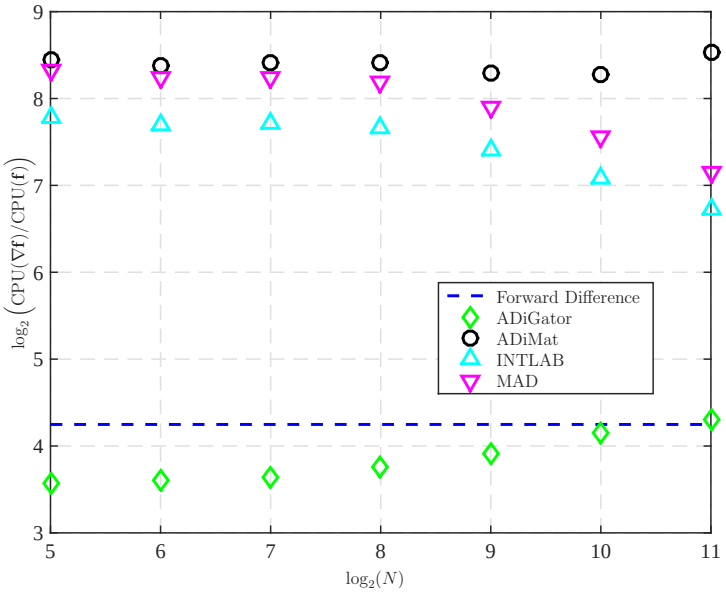
[1] This process of replacing loops with array operations is often referred to as "vectorization". In this dissertation the term "vectorized" has already been used to refer to a specific class of functions in Section 3.6. Thus, in order to avoid any confusion, use of the term "vectorization" is avoided when referring to functions whose loops have been replaced.

the original and modified files are given in Fig. 6-1. Results were obtained using ADiGator in the default mode, ADiMat in the scalar mode (by computing the 18 required directional derivatives independent of one another), 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. It should be noted, however, that the ADiGator Jacobian to function CPU ratios increase as the value of n_x increases, even though the derivative is becoming relatively sparser. This increase is again due to the increasing cost of performing indexing operations. To further investigate the handling of explicit loops, absolute function CPU times and ADiGator file generation times are given in Table 6-1. 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



A With Explicit Loops



B Without Explicit Loops

Figure 6-1. Burgers' ODE Jacobian to function CPU ratios. A) Ratios obtained by differentiating the original implementation of $\mathbf{f}$ containing explicit loops. B) Ratios obtained by differentiating the modified implementation of $\mathbf{f}$ containing no explicit loops.

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 [55]. Moreover, it is

Table 6-1. 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

known that the algorithm of `ode15s` is not extremely reliant upon precise Jacobian computations, and thus the `numjac` algorithm is not required to compute extremely accurate Jacobian approximations [55]. 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 plotted in Fig. 6-1. 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 6-2. Within the table 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 6-2. 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

6.2 Fixed Dimension Non-linear Systems of Equations

In this section, analysis is performed on a set of fixed dimension non-linear system of equations problems taken from the MINPACK-2 problem set [3]. While originally coded in Fortran, the implementations used for the tests of this section were obtained from Ref. [36]. 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 methods and thus require iterative Jacobian computations.

Prior to solving the non-linear 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 Ref. [36] provides hand-coded Jacobian files which also provide a nice base-line for computation efficiency. For each of the problems the ADiGator tool was tested against the ADiMat tool in the scalar compressed mode, the INTLAB tool's `gradient` class, the MAD tool in the compressed forward mode, and the hand-coded Jacobian as provided by Ref. [36]. 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 6. 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 6-3 together with the theoretical ratio for a sparse finite difference (sfd). From Table 6-3 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, and in the case of the compressible CTS problem, the ADiGator tool performs more efficiently than the hand-coded Jacobian file.

Table 6-3. Jacobian to function CPU ratios for CPF, HHD, and CTS problems.

Problem:	CPF	HHD	CTS
Jacobian to Function CPU Ratios, CPU($\nabla\mathbf{f}$)/CPU($\mathbf{f}$)			
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

The three test problems were next solved via the MATLAB optimization toolbox functions `fsolve` (for the CPF and HHD non-linear root-finding problems) and `lsqnonlin` (for the CTS non-linear 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 6-4, 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 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 6-4. Solution times for fixed dimension non-linear 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

6.3 Large Scale Unconstrained Minimization

In this section the 2-D Ginzburg-Landau (GL2) minimization problem is tested from the MINPACK-2 test suite [3]. The problem is to minimize the Gibbs free energy in the discretized Ginzburg-Landau superconductivity equations. The objective, f, is given by

$$f = \sum_{i=1}^{n_x} \sum_{j=1}^{n_y} -|v_{i,j}|^2 + \frac{1}{2}|v_{i,j}|^4 + \phi_{i,j}(\mathbf{v}, \mathbf{a}^{(1)}, \mathbf{a}^{(2)}), \tag{6–4}$$

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 Ref. [36], which also contains a hand-coded gradient file.[2] For the remainder of this

[2] The files obtained from Ref. [36] 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.

section, the objective function will be denoted by f, where $f : \mathbb{R}^n \to \mathbb{R}$ and the gradient function will be denoted $\mathbf{g}$, where $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$.

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 $\mathbf{g}$, were differentiated. Thus, three different tests were performed by computing: the gradient of the objective, ∇f, the Jacobian of the gradient, $\nabla \mathbf{g}$, and the Hessian of the objective, $\nabla^2 f$, where $\nabla \mathbf{g} = \nabla^2 f$. The aforementioned derivatives were computed using the ADiGator, ADiMat, INTLAB, and MAD tools and results are given in Table 6-5. Additionally given in Table 6-5 are the theoretical CPU ratios which would be required if a finite difference was to be used, together with the ratio required by the hand-coded gradient file. The results presented in Table 6-5 were obtained as follows. For the gradient computation, ∇f, 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 $\mathbf{g}$ was evaluated in order to get the hand-coded ratios, and the ratio given for a finite-difference is equal to $n + 1$. For the Jacobian computation, $\nabla \mathbf{g}$, the tested AD tools were applied to the gradient function 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, $\nabla^2 f$, 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 $2n + 1$; the number of function evaluations required to approximate the Hessian via a central difference. As witnessed from Table 6-5, 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. What may also be witnessed, is that for all three test cases, the ADiGator ratios tend to increase as the value of n is increased, even though the problem becomes sparser. This is again expected to be due to the relative increase in overhead due to indexing operations, where the relative efficiencies gained by statically exploiting sparsity at the operation level begin to be overshadowed by the indexing operations which are used in order to do so.

Table 6-5. Derivative to function CPU ratios for 2-D Ginzburg-Landau problem. Shown are the function gradient to function CPU ratios, CPU(∇f)/CPU(f), gradient Jacobian to function CPU ratios, CPU($\nabla \mathbf{g}$)/CPU(f), and function Hessian to function CPU ratios, CPU($\nabla^2 f$)/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(∇f)/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($\nabla \mathbf{g}$)/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($\nabla^2 f$)/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:	33.0	129.0	513.0	2049.0	8193.0	32769.0
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 6-5, the files generated by ADiGator are quite efficient at run-time. Unlike the problems of Section 6.2, 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 6-6. This table shows that the cost of generation of the objective gradient file, ∇f, and gradient Jacobian file, $\nabla \mathbf{g}$, are relatively small, while the cost of generating the objective Hessian file becomes quite expensive at $n = 16384$.[3] 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 6-5 and 6-6, 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 6-7, 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

[3] This expense is due to the fact that the overloaded `sum` operation performs the affine transformations of Eq. (3–47), which become quite expensive as n is increased.

obtains the Hessian sparsity pattern and an objective gradient file as a direct result of the Hessian file generation.

Table 6-6. 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 ∇f, gradient function $\mathbf{g}$ to Hessian function $\nabla \mathbf{g}$, and gradient function ∇f to Hessian function $\nabla^2 f$.

n:	16	64	256	1024	4096	16384
ADiGator File Generation Time (s)						
∇f:	0.51	0.51	0.52	0.53	0.58	0.90
$\nabla \mathbf{g}$:	2.44	2.51	2.51	2.57	2.89	4.33
$\nabla^2 f$:	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

Table 6-7. 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 6-6.

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	-

CHAPTER 7
CONCLUSIONS AND FUTURE WORK

The research of this dissertation is concerned with computing derivatives of mathematical functions described by MATLAB source code by utilizing algorithmic differentiation (AD). Applications which require numerical derivative computations are typically iterative, and thus require numerous evaluations of the same derivative. Due to the iterative nature of the applications of AD, it is often advantageous to perform an a priori analysis of the problem at compile-time in order to optimize derivative computation run-times. The research of this dissertation develops a run-time efficient method of affecting the forward-mode of AD by focusing upon two key concepts: source transformation and static sparsity exploitation. In a source transformation AD approach, function source code is transformed into a derivative source code which may then be evaluated numerically to compute the desired derivative. A source transformation approach is thus a static approach, where analysis of the problem may be performed at compile time in order to optimize the run-time of the resulting derivative files. Classically, source transformation is performed via four fundamental steps: parsing of the original source code, transformation of the program, optimization of the new program, and the printing of the optimized program. Typical optimizations include dead code elimination and common sub-expression elimination. The method developed in this dissertation performs source transformation AD via a non-classical operator overloaded approach. Moreover, rather than performing the typical dead code elimination and common sub-expression elimination, the developed method optimizes the resulting derivative code by exploiting sparsity at the operation level. Thus, unlike the conventional method of matrix compression, the method statically exploits sparsity at each link in the chain rule.

The manner in which the developed overloaded class is used to print run-time efficient derivative computations is detailed in Chapter 3. The method is based upon printing the most efficient derivative computations possible, given the information

known within each overloaded function-level operation. The primary way in which these optimizations are performed is by exploiting derivative sparsity, where derivative sparsity patterns are computed at compile-time. The resulting code is quite similar to that produced by the methods of Ref. [22], yet the approach is unique. 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 which are optimal for the sequence of elementary operations which the high-level operation performs, rather than printing derivative procedures which are optimal for each of the individual elementary operations. In order to exploit derivative sparsity, the overloaded operations print derivative procedures which typically only operate on vectors of non-zero derivatives, and thus rely heavily index array reference and assignment operations (e.g. `a(ind)`, where `ind` is a vector of integers). Due to the interpreted nature of the MATLAB language, however, 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 (e.g. `ind`). 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 methods of Chapter 3 allow for the automatic generation of efficient derivative procedures by evaluating sections of code on overloaded objects. The methods do not, however, allow for the transformation of function code containing flow control statements as flow control statements cannot simply be overloaded. The methods of Chapter 4 allow for the pseudo-overloading of flow control statements by performing a lexical analysis on the original function source code, and producing an intermediate source code, where flow control statements are replaced by calls to transformation routines. The intermediate

source code may then be evaluated on overloaded objects, where the AD algorithm is both made aware of the flow control constructs of the originating program and given control over the flow of the intermediate program. An algorithm is then developed such that the intermediate source code is evaluated on overloaded objects in order to print a new derivative source code, where the flow control of the originating function code is preserved in the derivative code. In order to do so, the methods must deal with the fact that the procedures printed by each overloaded evaluation are only valid for fixed derivative sparsity patterns. In the presence of flow control, however, derivative sparsity patterns corresponding to a variable may be dependent upon which branch of a conditional statement is taken, or which iteration of a loop is being performed. As a solution to this issue, an overloaded union operation is developed, where the procedures printed by operating on the result of the union of two overloaded objects are valid for each of the overloaded objects. At compile-time, overloaded unions are performed where branches of code join: on the output of conditional fragments, on the input to loop statements, and on both the input and output of sub-routines. The union of all possible overloaded objects which may be assigned to a variable is termed an "overmapped object". By evaluating each basic block of code on the proper overmapped objects and printing flow control statements where appropriate, a derivative source code is generated. Each overloaded operation thus prints derivative procedures which are deemed to be optimal given the information stored within the overmapped inputs to the operation. The methods developed in Chapters 3 and 4 are implemented in the open-source software, ADiGator.

In Chapter 5, a discussion is given on the efficient use of the ADiGator algorithm to compute the first and second derivatives of the non-linear program (NLP) which arises from direct collocation optimal control. It is shown that, if one takes advantage of discretization separability, the efficiency of differentiation of the discretized optimal control problem is strongly dependent upon the efficiency of the differentiation of vectorized functions of the form presented in Section 3.6. The methods of Chapter 5 thus make use

of the vectorized mode of the ADiGator algorithm, the advantages of which are shown to be three-fold. The first advantage of using the vectorized mode is that the generated derivative files are themselves vectorized and thus may be used for any given mesh when solving the optimal control problem. The next two advantages come from the fact that the ADiGator algorithm in the vectorized mode propagates and operates on the sparsity patterns of the smaller dimensional continuous-time functions rather than the sparsity patterns of the larger-dimensional discretized functions. Thus, sparsity propagation costs at compile-time and reference/assignment indexing overheads at run-time are effectively reduced by an order equal the vectorized dimension, N. The examples problems given in Chapter 5 demonstrate the efficiency of the approach and high-light the advantages of utilizing the vectorized mode. Moreover, by analyzing the run-times of the ADiGator generated derivative files produced by the non-vectorized and vectorized modes, run-time indexing penalties are quantified. It is shown that, while the derivative files produced by the non-vectorized modes are still quite efficient compared to other tested AD tools, a large portion of computation time is spent performing indexing operations, particularly at larger values of N.

In Chapter 6, the developed AD tool was applied to three different classes of problems. In the first example, the developed tool was used to compute a large sparse Jacobian used for integrating a stiff ordinary-differential equation (ODE). The method was shown to be more efficient at run-time than the other tested AD tools with efficiencies comparable to those of a sparse finite-difference. Unlike sparse finite-differencing routines, however, no a priori knowledge of the Jacobian sparsity pattern is required to efficiently use the ADiGator tool. For this example it was also shown that differentiating function codes containing explicit loops using the tested AD tools can be quite inefficient. The inefficiencies of the ADiGator algorithm when applied to an explicit loop are due largely to a loss of sparsity exploitation when operating on overmapped objects together with increases in run-time overheads when evaluating the generated derivative loops. In the

second example, the ADiGator tool was used together with the MATLAB solvers `fsolve` and `lsqnonlin` in order to solve three different systems of non-linear equations. The generated derivative files were shown to have run-times comparable to those of hand-coded Jacobians and sparse finite-differences. In the final example, the ADiGator tool was used to compute the dense gradient and large, sparse Hessian of an unconstrained minimization problem. It was found that the derivative files produced by the developed algorithm were always very efficient at run-time. The number of Hessian evaluations required to solve the minimization problem via a full-Newton solver were then used in order to attempt to quantify the efficiency of the ADiGator transformation process. It was found that when generation times were factored into the total time required to solve the minimization problem, that the ADiGator tool was not always the most efficient approach. This was particularly the case when only a small number of Hessian evaluations were required where the Hessian file generation times began to overshadow the time saved in the Hessian computation.

The developed method of this dissertation has been shown to automatically produce run-time efficient derivative files by statically exploiting sparsity at compile-time. No a priori knowledge of Jacobian and Hessian sparsity patterns is required to do so, and Jacobian and Hessian sparsity patterns are computed as a by-product of the derivative file generation. The developed approach is well-suited for applications requiring many repeated derivative computations, such as those of direct collocation optimal control. When used for such applications, the cost of the file generation becomes relatively insignificant when compared to the time saved at run-time. A result of concern is the large amount of run-time overhead incurred due to reference and assignment indexing operations, particularly for large dimensional problems. What is promising, however, is that the printed derivative procedures do not rely upon operator overloading and thus could presumably be printed as source code in an executable language, where indexing overheads would not be present at run-time. In order to do so, the code could either be

modified to directly print derivative source code in an executable language, or to print MATLAB derivative source code which may then be transformed via the MATLAB auto-coder. Given the existence of the MATLAB auto-coder, the latter is likely the easier approach. The algorithm in its current form produces derivative code which is meant to be efficient given the run-time overheads of the MATLAB language. With some hand modifications, the code can be transformed to an executable via the MATLAB auto-coder, however, the procedures which are most efficient when coded in MATLAB are not necessarily the procedures which are most efficient when coded in MATLAB and transformed to an executable via the MATLAB auto-coder. It is the opinion of the author of this dissertation that the best way to improve upon the developed methods is to modify the algorithm to operate under the assumption that the produced derivative code will be transformed via the MATLAB auto-coder. The algorithm could thus generate derivative files which are both valid for use with the auto-coder and optimal once transformed to an executable.

APPENDIX: MINIMUM TIME TO CLIMB FUNCTION AND DERIVATIVE CODE

Minimum Time to Climb Vectorized Function Code

```
function phaseout = MinimumClimbContinuous(input)

% Auxdata
CoF  = input.auxdata.CoF;
g    = input.auxdata.g;
m    = input.auxdata.m;
S    = input.auxdata.S;

x = input.phase.state;
u = input.phase.control;

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

% NOAA Atmosphere Model
Nzeros = 0.*h;
T = Nzeros;
p = Nzeros;

ihlow     = h < 11000;
if any(ihlow)
  T(ihlow)  = 15.04 - 0.00649*h(ihlow);
  p(ihlow)  = 101.29*((T(ihlow)+273.1)./288.08).^5.256;
end

ihhigh    = ~ihlow;
if any(ihhigh)
  T(ihhigh) = -56.46;
  p(ihhigh) = 22.65* exp(1.73 - 0.000157*h(ihhigh));
end

rho = p./(0.2869*(T+273.1));
q = 0.5.*rho.*v.*v.*S;

% Mach Number
hbar = h./1000;
theta = 292.1-8.87743.*hbar+0.193315.*hbar.^2+(3.72e-3).*hbar.^3;
a     = 20.0468.*sqrt(theta);

M  = v./a;
Mp = repmat(Nzeros,[1 6]);
for i = 0:5
  Mp(:,i+1) = M.^i;
end

% Compute Thrust and Drag Using Table Data
numeratorCD0 = Nzeros; denominatorCD0 = Nzeros;
numeratorK   = Nzeros; denominatorK   = Nzeros;
for i = 1:6
  Mpi = Mp(:,i);
  if i < 6
    numeratorCD0   = numeratorCD0   + CoF(1,i).*Mpi;
    denominatorCD0 = denominatorCD0 + CoF(2,i).*Mpi;
    numeratorK     = numeratorK     + CoF(3,i).*Mpi;
  end
  denominatorK = denominatorK  + CoF(4,i).*Mpi;
end
Cd0 = numeratorCD0./denominatorCD0;
K   = numeratorK./denominatorK;
% Drag
FD = q.*(Cd0+K.*((m.^2).*(g.^2)./(q.^2)).*(u.^2));

FT = Nzeros;
for i = 1:6
  ei = Nzeros;
  for j = 1:6
    ei = ei + CoF(4+j,i).*Mp(:,j);
  end
  FT = FT + ei.*hbar.^(i-1);
end
% Thrust
FT = FT.*9.80665./2.2;

% Dynamics
hdot = v.*sin(fpa);
vdot = (FT-FD)./m-g.*sin(fpa);
fpadot = g.*(u-cos(fpa))./v;

phaseout.dynamics = [hdot, vdot, fpadot];
```

Minimum Time to Climb Vectorized Derivative Code

```
% This code was generated using ADiGator version 1.0
% l'2010-2014 Matthew J. Weinstein and Anil V. Rao
% ADiGator may be obtained at https://sourceforge.net/projects/adigator/
% Contact: mweinstein@ufl.edu
% Bugs/suggestions may be reported to the sourceforge forums
%                    DISCLAIMER
% ADiGator is a general-purpose software distributed under the GNU General
% Public License version 3.0. While the software is distributed with the
% hope that it will be useful, both the software and generated code are
% provided 'AS IS' with NO WARRANTIES OF ANY KIND and no merchantability
% or fitness for any purpose or application.

function phaseout = MinimumClimbContinuousADiGatorGrd(input)
global ADiGator_MinimumClimbContinuousADiGatorGrd
if isempty(ADiGator_MinimumClimbContinuousADiGatorGrd); ADiGator_LoadData(); end
Gator1Data = ...
    ADiGator_MinimumClimbContinuousADiGatorGrd.MinimumClimbContinuousADiGatorGrd.Gator1Data;
% ADiGator Start Derivative Computations
%User Line: % Auxdata
CoF.f = input.auxdata.CoF;
%User Line: CoF  = input.auxdata.CoF;
g.f = input.auxdata.g;
%User Line: g    = input.auxdata.g;
m.f = input.auxdata.m;
%User Line: m    = input.auxdata.m;
S.f = input.auxdata.S;
%User Line: S    = input.auxdata.S;
x.dV = input.phase.state.dV; x.f = input.phase.state.f;
%User Line: x = input.phase.state;
u.dV = input.phase.control.dV; u.f = input.phase.control.f;
%User Line: u = input.phase.control;
h.dV = x.dV(:,1);
h.f = x.f(:,1);
%User Line: h = x(:,1);
v.dV = x.dV(:,2);
v.f = x.f(:,2);
%User Line: v = x(:,2);
fpa.dV = x.dV(:,3);
fpa.f = x.f(:,3);
%User Line: fpa = x(:,3);
%User Line: % NOAA Atmosphere Model
Nzeros.f = 0*h.f;
%User Line: Nzeros = 0.*h;
T.f = Nzeros.f;
%User Line: T = Nzeros;
p.f = Nzeros.f;
%User Line: p = Nzeros;
ihlow.f = lt(h.f,11000);
%User Line: ihlow     = h < 11000;
cadaconditional1 = any(ihlow.f,1);
%User Line: cadaconditional1 = any(ihlow);
if cadaconditional1
    cada1f1dV = h.dV(:,1);
    cada1f1 = h.f(:);
    cada1f2dV = 0.00649.*cada1f1dV;
    cada1f2 = 0.00649*cada1f1;
    cada1f3dV = -cada1f2dV;
    cada1f3 = 15.04 - cada1f2;
    cada1td2 = zeros(size(T.f,1),1);
    cada1tind1 = ihlow.f(:,1);
    cada1td2(cada1tind1) = cada1f3dV(cada1tind1);
    T.dV = cada1td2;
    T.f(ihlow.f) = cada1f3(ihlow.f);
    %User Line: T(ihlow)  = 15.04 - 0.00649*h(ihlow);
    cada1f1dV = T.dV(:,1);
    cada1f1 = T.f(:);
    cada1f2dV = cada1f1dV;
    cada1f2 = cada1f1 + 273.1;
    cada1f3dV = cada1f2dV./288.08;
    cada1f3 = cada1f2/288.08;
    cada1f4dV = 5.256.*cada1f3.^(5.256-1).*cada1f3dV;
    cada1f4 = cada1f3.^5.256;
    cada1f5dV = 101.29.*cada1f4dV;
    cada1f5 = 101.29*cada1f4;
    cada1td2 = zeros(size(p.f,1),1);
    cada1tind1 = ihlow.f(:,1);
    cada1td2(cada1tind1) = cada1f5dV(cada1tind1);
    p.dV = cada1td2;
    p.f(ihlow.f) = cada1f5(ihlow.f);
    %User Line: p(ihlow)  = 101.29*((T(ihlow)+273.1)./288.08).^5.256;
else
    T.dV = zeros(size(T.f,1),1);
    p.dV = zeros(size(p.f,1),1);
```

```
end
ihhigh.f = not(ihlow.f);
%User Line: ihhigh    = ~ihlow;
cadaconditional1 = any(ihhigh.f,1);
%User Line: cadaconditional1 = any(ihhigh);
if cadaconditional1
    cada1td2 = T.dV;
    cada1tind1 = ihhigh.f(:,1);
    cada1td2(cada1tind1) = 0;
    T.dV = cada1td2;
    T.f(ihhigh.f) = -56.46;
    %User Line: T(ihhigh) = -56.46;
    cada1f1dV = h.dV(:,1);
    cada1f1 = h.f(:);
    cada1f2dV = 0.000157.*cada1f1dV;
    cada1f2 = 0.000157*cada1f1;
    cada1f3dV = -cada1f2dV;
    cada1f3 = 1.73 - cada1f2;
    cada1f4dV = exp(cada1f3).*cada1f3dV;
    cada1f4 = exp(cada1f3);
    cada1f5dV = 22.65.*cada1f4dV;
    cada1f5 = 22.65*cada1f4;
    cada1td2 = p.dV;
    cada1tind1 = ihhigh.f(:,1);
    cada1td2(cada1tind1) = cada1f5dV(cada1tind1);
    p.dV = cada1td2;
    p.f(ihhigh.f) = cada1f5(ihhigh.f);
    %User Line: p(ihhigh) = 22.65* exp(1.73 - 0.000157*h(ihhigh));
else
end
cada1f1dV = T.dV;
cada1f1 = T.f + 273.1;
cada1f2dV = 0.2869.*cada1f1dV;
cada1f2 = 0.2869*cada1f1;
cada1td1 = p.dV./cada1f2;
cada1td1 = cada1td1 + -p.f./cada1f2.^2.*cada1f2dV;
rho.dV = cada1td1;
rho.f = p.f./cada1f2;
%User Line: rho = p./(0.2869*(T+273.1));
cada1f1dV = 0.5.*rho.dV;
cada1f1 = 0.5*rho.f;
cada1td1 = zeros(size(cada1f1dV,1),2);
cada1td1(:,1) = v.f.*cada1f1dV;
cada1td1(:,2) = cada1td1(:,2) + cada1f1.*v.dV;
cada1f2dV = cada1td1;
cada1f2 = cada1f1.*v.f;
cada1tf1 = v.f(:,Gator1Data.Index4);
cada1td1 = cada1tf1.*cada1f2dV;
cada1td1(:,2) = cada1td1(:,2) + cada1f2.*v.dV;
cada1f3dV = cada1td1;
cada1f3 = cada1f2.*v.f;
q.dV = S.f.*cada1f3dV;
q.f = cada1f3*S.f;
%User Line: q = 0.5.*rho.*v.*v.*S;
%User Line: % Mach Number
hbar.dV = h.dV./1000;
hbar.f = h.f/1000;
%User Line: hbar = h./1000;
cada1f1dV = 8.87743.*hbar.dV;
cada1f1 = 8.87743*hbar.f;
cada1f2dV = -cada1f1dV;
cada1f2 = 292.1 - cada1f1;
cada1f3dV = 2.*hbar.f.^(2-1).*hbar.dV;
cada1f3 = hbar.f.^2;
cada1f4dV = 0.193315.*cada1f3dV;
cada1f4 = 0.193315*cada1f3;
cada1td1 = cada1f2dV;
cada1td1 = cada1td1 + cada1f4dV;
cada1f5dV = cada1td1;
cada1f5 = cada1f2 + cada1f4;
cada1f6dV = 3.*hbar.f.^(3-1).*hbar.dV;
cada1f6 = hbar.f.^3;
cada1f7dV = 0.00372.*cada1f6dV;
cada1f7 = 0.00372*cada1f6;
cada1td1 = cada1f5dV;
cada1td1 = cada1td1 + cada1f7dV;
theta.dV = cada1td1;
theta.f = cada1f5 + cada1f7;
%User Line: theta = 292.1-8.87743.*hbar+0.193315.*hbar.^2+(3.72e-3).*hbar.^3;
cada1f1dV = (1/2)./sqrt(theta.f).*theta.dV;
cada1f1dV(theta.f == 0 & theta.dV == 0) = 0;
cada1f1 = sqrt(theta.f);
a.dV = 20.0468.*cada1f1dV;
a.f = 20.0468*cada1f1;
%User Line: a     = 20.0468.*sqrt(theta);
cada1td1 = zeros(size(v.dV,1),2);
cada1td1(:,2) = v.dV./a.f;
cada1td1(:,1) = cada1td1(:,1) + -v.f./a.f.^2.*a.dV;
M.dV = cada1td1;
```

```
M.f = v.f./a.f;
%User Line: M   = v./a;
Mp.f = repmat(Nzeros.f,1,6);
%User Line: Mp  = repmat(Nzeros,[1 6]);
cadaforvar1.f =  0:5;
%User Line: cadaforvar1 = 0:5;
Mp.dV = zeros(size(Mp.f,1),10);
for cadaforcount1 = 1:6
    i.f = cadaforvar1.f(:,cadaforcount1);
    %User Line: i = cadaforvar1(:,cadaforcount1);
    cada1tf2 = M.f(:,Gator1Data.Index5);
    cada1f1dV = i.f.*cada1tf2.^(i.f-1).*M.dV;
    cada1f1dV((cada1tf2 == 0 & M.dV == 0) | i.f == 0) = 0;
    cada1f1 = M.f.^i.f;
    cada1f2 = i.f + 1;
    Mp.dV(:,logical(Gator1Data.Index1(:,cadaforcount1))) = ...
        cada1f1dV(:,nonzeros(Gator1Data.Index1(:,cadaforcount1)));
    Mp.f(:,cada1f2) = cada1f1;
    %User Line: Mp(:,i+1) = M.^i;
end
%User Line: % Compute Thrust and Drag Using Table Data
numeratorCD0.f = Nzeros.f;
%User Line: numeratorCD0 = Nzeros;
denominatorCD0.f = Nzeros.f;
%User Line: denominatorCD0 = Nzeros;
numeratorK.f = Nzeros.f;
%User Line: numeratorK   = Nzeros;
denominatorK.f = Nzeros.f;
%User Line: denominatorK   = Nzeros;
cadaforvar2.f =  1:6;
%User Line: cadaforvar2 = 1:6;
numeratorCD0.dV = zeros(size(numeratorCD0.f,1),2);
denominatorCD0.dV = zeros(size(denominatorCD0.f,1),2);
numeratorK.dV = zeros(size(numeratorK.f,1),2);
denominatorK.dV = zeros(size(denominatorK.f,1),2);
for cadaforcount2 = 1:6
    i.f = cadaforvar2.f(:,cadaforcount2);
    %User Line: i = cadaforvar2(:,cadaforcount2);
    cada1td1 = zeros(size(Mp.f,1),2);
    cada1td1(:,logical(Gator1Data.Index2(:,cadaforcount2))) = ...
        Mp.dV(:,nonzeros(Gator1Data.Index2(:,cadaforcount2)));
    Mpi.dV = cada1td1;
    Mpi.f = Mp.f(:,i.f);
    %User Line: Mpi = Mp(:,i);
    cadaconditional1 = lt(i.f,6);
    %User Line: cadaconditional1 = i < 6;
    if cadaconditional1
        cada1f1 = CoF.f(1,i.f);
        cada1f2dV = cada1f1.*Mpi.dV;
        cada1f2 = cada1f1*Mpi.f;
        cada1td1 = numeratorCD0.dV;
        cada1td1 = cada1td1 + cada1f2dV;
        numeratorCD0.dV = cada1td1;
        numeratorCD0.f = numeratorCD0.f + cada1f2;
        %User Line: numeratorCD0   = numeratorCD0   + CoF(1,i).*Mpi;
        cada1f1 = CoF.f(2,i.f);
        cada1f2dV = cada1f1.*Mpi.dV;
        cada1f2 = cada1f1*Mpi.f;
        cada1td1 = denominatorCD0.dV;
        cada1td1 = cada1td1 + cada1f2dV;
        denominatorCD0.dV = cada1td1;
        denominatorCD0.f = denominatorCD0.f + cada1f2;
        %User Line: denominatorCD0 = denominatorCD0 + CoF(2,i).*Mpi;
        cada1f1 = CoF.f(3,i.f);
        cada1f2dV = cada1f1.*Mpi.dV;
        cada1f2 = cada1f1*Mpi.f;
        cada1td1 = numeratorK.dV;
        cada1td1 = cada1td1 + cada1f2dV;
        numeratorK.dV = cada1td1;
        numeratorK.f = numeratorK.f + cada1f2;
        %User Line: numeratorK     = numeratorK     + CoF(3,i).*Mpi;
    end
    cada1f1 = CoF.f(4,i.f);
    cada1f2dV = cada1f1.*Mpi.dV;
    cada1f2 = cada1f1*Mpi.f;
    cada1td1 = denominatorK.dV;
    cada1td1 = cada1td1 + cada1f2dV;
    denominatorK.dV = cada1td1;
    denominatorK.f = denominatorK.f + cada1f2;
    %User Line: denominatorK = denominatorK  + CoF(4,i).*Mpi;
end
cada1tf1 = denominatorCD0.f(:,Gator1Data.Index6);
cada1td1 = numeratorCD0.dV./cada1tf1;
cada1tf1 = numeratorCD0.f(:,Gator1Data.Index7);
cada1tf2 = denominatorCD0.f(:,Gator1Data.Index8);
cada1td1 = cada1td1 + -cada1tf1./cada1tf2.^2.*denominatorCD0.dV;
Cd0.dV = cada1td1;
Cd0.f = numeratorCD0.f./denominatorCD0.f;
%User Line: Cd0 = numeratorCD0./denominatorCD0;
```

```
cada1tf1 = denominatorK.f(:,Gator1Data.Index9);
cada1td1 = numeratorK.dV./cada1tf1;
cada1tf1 = numeratorK.f(:,Gator1Data.Index10);
cada1tf2 = denominatorK.f(:,Gator1Data.Index11);
cada1td1 = cada1td1 + -cada1tf1./cada1tf2.^2.*denominatorK.dV;
K.dV = cada1td1;
K.f = numeratorK.f./denominatorK.f;
%User Line: K   = numeratorK./denominatorK;
%User Line: % Drag
cada1f1 = m.f^2;
cada1f2 = g.f^2;
cada1f3 = cada1f1*cada1f2;
cada1tf2 = q.f(:,Gator1Data.Index12);
cada1f4dV = 2.*cada1tf2.^(2-1).*q.dV;
cada1f4 = q.f.^2;
cada1tf2 = cada1f4(:,Gator1Data.Index13);
cada1f5dV = -cada1f3./cada1tf2.^2.*cada1f4dV;
cada1f5 = cada1f3./cada1f4;
cada1tf1 = cada1f5(:,Gator1Data.Index14);
cada1td1 = cada1tf1.*K.dV;
cada1tf1 = K.f(:,Gator1Data.Index15);
cada1td1 = cada1td1 + cada1tf1.*cada1f5dV;
cada1f6dV = cada1td1;
cada1f6 = K.f.*cada1f5;
cada1f7dV = 2.*u.f.^(2-1).*u.dV;
cada1f7 = u.f.^2;
cada1tf1 = cada1f7(:,Gator1Data.Index16);
cada1td1 = zeros(size(cada1f6dV,1),3);
cada1td1(:,Gator1Data.Index17) = cada1tf1.*cada1f6dV;
cada1td1(:,3) = cada1td1(:,3) + cada1f6.*cada1f7dV;
cada1f8dV = cada1td1;
cada1f8 = cada1f6.*cada1f7;
cada1td1 = zeros(size(Cd0.dV,1),3);
cada1td1(:,Gator1Data.Index18) = Cd0.dV;
cada1td1 = cada1td1 + cada1f8dV;
cada1f9dV = cada1td1;
cada1f9 = Cd0.f + cada1f8;
cada1tf1 = cada1f9(:,Gator1Data.Index19);
cada1td1 = zeros(size(q.dV,1),3);
cada1td1(:,Gator1Data.Index20) = cada1tf1.*q.dV;
cada1tf1 = q.f(:,Gator1Data.Index21);
cada1td1 = cada1td1 + cada1tf1.*cada1f9dV;
FD.dV = cada1td1;
FD.f = q.f.*cada1f9;
%User Line: FD = q.*(Cd0+K.*((m.^2).*(g.^2)./(q.^2)).*(u.^2));
FT.f = Nzeros.f;
%User Line: FT = Nzeros;
cadaforvar3.f =  1:6;
%User Line: cadaforvar3 = 1:6;
FT.dV = zeros(size(FT.f,1),2);
for cadaforcount3 = 1:6
    i.f = cadaforvar3.f(:,cadaforcount3);
    %User Line: i = cadaforvar3(:,cadaforcount3);
    ei.f = Nzeros.f;
    ei.dV = zeros(size(ei.f,1),2);
    %User Line: ei = Nzeros;
    cadaforvar4.f =  1:6;
    %User Line: cadaforvar4 = 1:6;
    for cadaforcount4 = 1:6
        j.f = cadaforvar4.f(:,cadaforcount4);
        %User Line: j = cadaforvar4(:,cadaforcount4);
        cada1f1 = 4 + j.f;
        cada1f2 = CoF.f(cada1f1,i.f);
        cada1td1 = zeros(size(Mp.f,1),2);
        cada1td1(:,logical(Gator1Data.Index3(:,cadaforcount4))) = ...
            Mp.dV(:,nonzeros(Gator1Data.Index3(:,cadaforcount4)));
        cada1f3dV = cada1td1;
        cada1f3 = Mp.f(:,j.f);
        cada1f4dV = cada1f2.*cada1f3dV;
        cada1f4 = cada1f2*cada1f3;
        cada1td1 = ei.dV;
        cada1td1 = cada1td1 + cada1f4dV;
        ei.dV = cada1td1;
        ei.f = ei.f + cada1f4;
        %User Line: ei = ei + CoF(4+j,i).*Mp(:,j);
    end
    cada1f1 = i.f - 1;
    cada1f2dV = cada1f1.*hbar.f.^(cada1f1-1).*hbar.dV;
    cada1f2dV((hbar.f == 0 & hbar.dV == 0) | cada1f1 == 0) = 0;
    cada1f2 = hbar.f.^cada1f1;
    cada1tf1 = cada1f2(:,Gator1Data.Index22);
    cada1td1 = cada1tf1.*ei.dV;
    cada1td1(:,1) = cada1td1(:,1) + ei.f.*cada1f2dV;
    cada1f3dV = cada1td1;
    cada1f3 = ei.f.*cada1f2;
    cada1td1 = FT.dV;
    cada1td1 = cada1td1 + cada1f3dV;
    FT.dV = cada1td1;
    FT.f = FT.f + cada1f3;
```

```
    %User Line: FT = FT + ei.*hbar.^(i-1);
end
%User Line: % Thrust
cada1f1dV = Gator1Data.Data1.*FT.dV;
cada1f1 = FT.f*Gator1Data.Data1;
FT.dV = cada1f1dV./2.2;
FT.f = cada1f1/2.2;
%User Line: FT = FT.*9.80665./2.2;
%User Line: % Dynamics
cada1f1dV = cos(fpa.f).*fpa.dV;
cada1f1 = sin(fpa.f);
cada1td1 = zeros(size(v.dV,1),2);
cada1td1(:,1) = cada1f1.*v.dV;
cada1td1(:,2) = cada1td1(:,2) + v.f.*cada1f1dV;
hdot.dV = cada1td1;
hdot.f = v.f.*cada1f1;
%User Line: hdot = v.*sin(fpa);
cada1td1 = zeros(size(FT.dV,1),3);
cada1td1(:,Gator1Data.Index23) = FT.dV;
cada1td1 = cada1td1 + -FD.dV;
cada1f1dV = cada1td1;
cada1f1 = FT.f - FD.f;
cada1f2dV = cada1f1dV./m.f;
cada1f2 = cada1f1/m.f;
cada1f3dV = cos(fpa.f).*fpa.dV;
cada1f3 = sin(fpa.f);
cada1f4dV = g.f.*cada1f3dV;
cada1f4 = g.f*cada1f3;
cada1td1 = zeros(size(cada1f2dV,1),4);
cada1td1(:,Gator1Data.Index24) = cada1f2dV;
cada1td1(:,3) = cada1td1(:,3) + -cada1f4dV;
vdot.dV = cada1td1;
vdot.f = cada1f2 - cada1f4;
%User Line: vdot = (FT-FD)./m-g.*sin(fpa);
cada1f1dV = -sin(fpa.f).*fpa.dV;
cada1f1 = cos(fpa.f);
cada1td1 = zeros(size(u.dV,1),2);
cada1td1(:,2) = u.dV;
cada1td1(:,1) = cada1td1(:,1) + -cada1f1dV;
cada1f2dV = cada1td1;
cada1f2 = u.f - cada1f1;
cada1f3dV = g.f.*cada1f2dV;
cada1f3 = g.f*cada1f2;
cada1tf1 = v.f(:,Gator1Data.Index25);
cada1td1 = zeros(size(cada1f3dV,1),3);
cada1td1(:,Gator1Data.Index26) = cada1f3dV./cada1tf1;
cada1td1(:,1) = cada1td1(:,1) + -cada1f3./v.f.^2.*v.dV;
fpadot.dV = cada1td1;
fpadot.f = cada1f3./v.f;
%User Line: fpadot = g.*(u-cos(fpa))./v;
cada1td1 = zeros(size(hdot.f,1),9);
cada1td1(:,Gator1Data.Index27) = hdot.dV;
cada1td1(:,Gator1Data.Index28) = vdot.dV;
cada1td1(:,Gator1Data.Index29) = fpadot.dV;
phaseout.dynamics.dV = cada1td1;
phaseout.dynamics.f = [hdot.f vdot.f fpadot.f];
%User Line: phaseout.dynamics = [hdot, vdot, fpadot];
phaseout.dynamics.dV_size = [3,5];
phaseout.dynamics.dV_location = Gator1Data.Index30;
end


function ADiGator_LoadData()
global ADiGator_MinimumClimbContinuousADiGatorGrd
ADiGator_MinimumClimbContinuousADiGatorGrd = load('MinimumClimbContinuousADiGatorGrd.mat');
return
end
```

REFERENCES

[1] Ascher, U., Mattheij, R., and Russell, R. *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations.* Society for Industrial and Applied Mathematics, 1995.

[2] Aubert, Pierre, Di Césaré, Nicolas, and Pironneau, Olivier. "Automatic Differentiation in C++ Using Expression Templates and Application to a Flow Control Problem." *Computing and Visualization in Science* 3 (2001): 197–208.

[3] Averick, Brett M., Carter, Richard G., and MorÃĺ, Jorge J. "The Minpack-2 Test Problem Collection." 1991.

[4] Bendtsen, C. and Stauning, Ole. "FADBAD, a Flexible C++ Package for Automatic Differentiation." Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, 1996.

[5] Berz, Martin. "The Differential Algebra Fortran Precompiler DAFOR." Tech. Report AT–3: TN–87–32, Los Alamos National Laboratory, Los Alamos, N.M., 1987.

[6] Berz, Martin, Makino, Kyoko, Shamseddine, Khodr, Hoffstätter, Georg H., and Wan, Weishi. "COSY INFINITY and Its Applications in Nonlinear Dynamics." *Computational Differentiation: Techniques, Applications, and Tools.* eds. Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. Philadelphia, PA: SIAM, 1996. 363–365.

[7] Betts, J. T. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming.* Philadelphia: SIAM Press, 2009, 2 ed.

[8] Betts, John T and Huffman, William P. "Sparse optimal control software SOCS." *Mathematics and Engineering Analysis Technical Document MEA-LR-085, Boeing Information and Support Services, The Boeing Company, PO Box* 3707 (1997): 98124–2207.

[9] Bischof, C., Lang, B., and Vehreschild, A. "Automatic differentiation for MATLAB programs." *Proceedings in Applied Mathematics and Mechanics* 2 (2003).1 Joh Wiley: 50–53.

[10] Bischof, Christian H., Bücker, H. Martin, Lang, Bruno, Rasch, A., and Vehreschild, Andre. "Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs." *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002).* Los Alamitos, CA, USA: IEEE Computer Society, 2002, 65–72.

[11] Bischof, Christian H., Carle, Alan, Corliss, George F., Griewank, Andreas, and Hovland, Paul D. "ADIFOR: Generating Derivative Codes from Fortran Programs." *Scientific Programming* 1 (1992).1: 11–29.

[12] Bischof, Christian H., Carle, Alan, Khademi, Peyvand, and Mauer, Andrew. "ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs." *IEEE Computational Science & Engineering* 3 (1996).3: 18–32.

[13] Coleman, Thomas F and Verma, Arun. *ADMAT: An Automatic Differentiation Toolbox for MATLAB. Technical Report.* Computer Science Department, Cornell University, 1998.

[14] ———. "The Efficient Computation of Sparse Jacobian Matrices Using Automatic Differentiation." *SIAM Journal on Scientific Computing* 19 (1998).4: 1210–1233.

[15] ———. "ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox." *ACM Transactions on Mathematical Software* 26 (2000).1: 150–175.

[16] Curtis, A. R., Powell, M. J. D., and Reid, J. K. "On the Estimation of Sparse Jacobian Matrices." *IMA Journal of Applied Mathematics* 13 (1974).1: 117–119.

[17] Darby, C. L., Hager, W. W., and Rao, A. V. "An *hp*–Adaptive Pseudospectral Method for Solving Optimal Control Problems." *Optimal Control Applications and Methods* 32 (2011).4: 476–502.

[18] Darby, Christopher L., Hager, W. W., and Rao, Anil V. "Direct Trajectory Optimization Using a Variable Low-Order Adaptive Pseudospectral Method." *Journal of Spacecraft and Rockets* 48 (2011).3: 433–445.

[19] Dobmann, M., Liepelt, M., and Schittkowski, K. "Algorithm 746: PCOMP—a Fortran Code for Automatic Differentiation." *ACM Transactions on Mathematical Software* 21 (1995).3: 233–266.

[20] Duff, Iain S. "MA57—a Code for the Solution of Sparse Symmetric Definite and Indefinite Systems." *ACM Transactions on Mathematical Software* 30 (2004).2: 118–144.

[21] Forth, S. A. "An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in MATLAB." *ACM Transactions on Mathematical Software* 32 (2006).2: 195–222.

[22] Forth, S. A., Tadjouddine, M., Pryce, J. D., and Reid, J. K. "Jacobian Code Generated by Source Transformation and Vertex Elimination Can Be As Efficient As Hand–Coding." *ACM Transactions on Mathematical Software* 30 (2004).4: 266–299.

[23] Garg, D., Hager, W. W., and Rao, A. V. "Pseudospectral Methods for Solving Infinite-Horizon Optimal Control Problems." *Automatica* 47 (2011).4: 829–837.

[24] Garg, D., Patterson, M. A., Darby, C. L., Françolin, C., Huntington, G. T., Hager, W. W., and Rao, A. V. "Direct Trajectory Optimization and Costate Estimation of Finite-Horizon and Infinite-Horizon Optimal Control Problems via a Radau

Pseudospectral Method." *Computational Optimization and Applications* 49 (2011).2: 335–358.

[25] Garg, D., Patterson, M. A., Hager, W. W., Rao, A. V., Benson, D. A., and Huntington, G. T. "A Unified Framework for the Numerical Solution of Optimal Control Problems Using Pseudospectral Methods." *Automatica* 46 (2010).11: 1843–1851.

[26] Gebremedhin, Assefaw Hadish, Manne, Fredrik, and Pothen, Alex. "What color is your Jacobian? Graph coloring for computing derivatives." *SIAM REV* 47 (2005): 629–705.

[27] Giering, Ralf and Kaminski, Thomas. "Recipes for Adjoint Code Construction." Tech. Rep. 212, Max-Planck-Institut für Meteorologie, 1996.

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

[29] Griewank, A., Juedes, D., and Utke, J. "Algorithm 755: ADOL-C, a Package for the Automatic Differentiation of Algorithms Written in C/C++." *ACM Transactions on Mathematical Software* 22 (1996).2: 131–167.

[30] Hascoët, Laurent and Pascual, Valérie. "TAPENADE 2.1 User's Guide." Rapport Technique 300, INRIA, Sophia Antipolis, 2004.

[31] Horwedel, Jim E. "GRESS, A Preprocessor for Sensitivity Studies of Fortran Programs." *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* eds. Andreas Griewank and George F. Corliss. Philadelphia, PA: SIAM, 1991. 243–250.

[32] Huang, Weizhang, Ren, Yuhe, and Russell, Robert D. "Moving Mesh Methods Based on Moving Mesh Partial Differential Equations." *J. Comput. Phys* 113 (1994): 279–290.

[33] Kharche, R. V. *MATLAB Automatic Differentiation using Source Transformation.* Ph.D. thesis, Department of Informatics, Systems Engineering, Applied Mathematics, and Scientific Computing, Cranfield University, 2011.

[34] Kharche, R. V. and Forth, S. A. "Source Transformation for MATLAB Automatic Differentiation." *Computational Science – ICCS, Lecture Notes in Computer Science.* eds. V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, vol. 3994. Heidelberg, Germany: Springer, 2006. 558–565.

[35] Kubota, Koichi. "PADRE2, A Fortran Precompiler Yielding Error Estimates and Second Derivatives." *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* eds. Andreas Griewank and George F. Corliss. Philadelphia, PA: SIAM, 1991. 251–262.

[36] Lenton, Katharine. *An Efficient, Validated Implementation of the MINPACK-2 Test Problem Collection in MATLAB.* Master's thesis, Cranfield University (Shrivenham Campus), Applied Mathematics & Operational Research Group, Engineering Systems Department, RMCS Shrivenham, Swindon SN6 8LA, UK, 2005.

[37] Mathworks. *MATLAB Version R2014b.* Natick, Massachusetts: The MathWorks Inc., 2014.

[38] Menon, Vijay and Pingali, Keshav. "A Case for Source-level Transformations in MATLAB." *SIGPLAN Not.* 35 (1999).1: 53–65.

[39] Michelotti, Leo. "MXYZPTLK: A C++ Hacker's Implementation of Automatic Differentiation." *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* eds. Andreas Griewank and George F. Corliss. Philadelphia, PA: SIAM, 1991. 218–227.

[40] Monagan, Michael B., Geddes, Keith O., Heal, K. Michael, Labahn, George, Vorkoetter, Stefan M., McCarron, James, and DeMarco, Paul. *Maple 10 Programming Guide.* Waterloo ON, Canada: Maplesoft, 2005.

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

[42] Nocedal, Jorge and Wright, Stephen J. *Numerical optimization*, vol. 2. Springer New York, 1999.

[43] Patterson, M. A. and Rao, A. V. "Exploiting Sparsity in Direct Collocation Pseudospectral Methods for Solving Continuous-Time Optimal Control Problems." *Journal of Spacecraft and Rockets,* 49 (2012).2: 364–377.

[44] Patterson, M. A., Weinstein, M. J., and Rao, A. V. "An Efficient Overloaded Method for Computing Derivatives of Mathematical Functions in MATLAB." *ACM Transactions on Mathematical Software,* 39 (2013).3: 17:1–17:36.

[45] Patterson, Michael A., Hager, William W., and Rao, Anil V. "A ph mesh refinement method for optimal control." *Optimal Control Applications and Methods* (2014).

[46] Patterson, Michael A. and Rao, Anil V. "GPOPS-II: A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp-Adaptive Gaussian Quadrature Collocation Methods and Sparse Nonlinear Programming." *ACM Trans. Math. Softw.* 41 (2014).1: 1:1–1:37.

[47] Patterson, Michael A and Rao, Anil V. "User's Guide for GPOPS-II, A General-Purpose MATLAB Software for Solving Multiple-Phase Optimal Control Problems. Version 2.0." (2014).

[48] Pryce, John D. and Reid, John K. "ADO1, a Fortran 90 code for Automatic Differentiation." Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998.

[49] Rao, Anil V, Benson, David A, Darby, Christopher, Patterson, Michael A, Francolin, Camila, Sanders, Ilyssa, and Huntington, Geoffrey T. "Algorithm 902: Gpops, a matlab software for solving multiple-phase optimal control problems using the gauss pseudospectral method." *ACM Transactions on Mathematical Software (TOMS)* 37 (2010).2: 22.

[50] Rhodin, Andreas. "{IMAS} Integrated Modeling and Analysis System for the solution of optimal control problems." *Computer Physics Communications* 107 (1997).1?3: 21 – 38.

[51] Ross, IM and Fahroo, F. "UserâĂŹs Manual for DIDO 2001 α: A MATLAB Application for Solving Optimal Control Problems." Tech. rep., Tech. rep. AAS-01-03. Department of Aeronautics and Astronautics, Naval Postgraduate School, Monterey, CA, 2001.

[52] Rostaing-Schmidt, Nicole. *Différentiation Automatique: Application à un Problème d'Optimisation en Météorologie*. Ph.D. thesis, Université de Nice-Sophia Antipolis, 1993.

[53] Rump, S. M. "INTLAB – INTerval LABoratory." *Developments in Reliable Computing*. ed. Tibor Csendes. Dordrecht, Germany: Kluwer Academic Publishers, 1999. 77–104.

[54] Seywald, Hans, Cliff, Eugene M., and Well, Klaus H. "Range optimal trajectories for an aircraft flying in the vertical plane." *Journal of Guidance, Control, and Dynamics* 17 (1994).2: 389–398.

[55] Shampine, Lawrence F and Reichelt, Mark W. "The MATLAB ODE Suite." *SIAM journal on scientific computing* 18 (1997).1: 1–22.

[56] Shiriaev, D. and Griewank, A. "ADOL-F Automatic Differentiation of Fortran Codes." *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, 1996, 375–384.

[57] Speelpenning, B. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1980.

[58] Tadjouddine, Mohamed, Forth, Shaun A., and Pryce, John D. "Hierarchical Automatic Differentiation by Vertex Elimination and Source Transformation." *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*. eds. V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, vol. 2668 of *Lecture Notes in Computer Science*. Springer, 2003, 115–124.

[59] von Stryk, Oskar. "User's Guide for DIRCOL, a Direct Collocation Method for the Numerical Solution of Optimal Control Problems. Version 2.1." *Simulation, Systems Optimization and Robotics Group, Technical University of Darmstadt.* (1999).

[60] Waechter, A. and Biegler, L. T. "On the Implementation of a Primal-Dual Interior-Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming." *Mathematical Programming* 106 (2006).1: 575–582.

[61] Wolfram. *Mathematica Edition: Version 7.0.* Champaign, Illinois: Wolfram Research, Inc., 2008.

BIOGRAPHICAL SKETCH

Matthew J. Weinstein was born and raised outside of Pittsburgh, Pennsylvania. He received his Bachelor of Science degree in mechanical engineering with a minor in electrical engineering from the University of Pittsburgh in December 2010. He then received his Master of Science degree in mechanical engineering in August 2014, and his Doctor of Philosophy in mechanical engineering in May 2015 from the University of Florida. His research interests include algorithmic differentiation, numerical approximations to differential equations, and numerical approximations to the solution of optimal control problems.