Incremental Computation of Taylor Series and System Jacobian in DAE solving using Automatic Differentiation

INCREMENTAL COMPUTATION OF TAYLOR SERIES AND SYSTEM JACOBIAN IN DAE SOLVING USING AUTOMATIC DIFFERENTIATION

BY

XIAO (SHAWN) LI

A THESIS

SUBMITTED TO THE SCHOOL OF COMPUTATIONAL SCIENCE AND ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

M.Sc. in Computational Science and Engineering (2017) McMaster University

(School of Computational Science and Engineering) Hamilton, Ontario, Canada

TITLE:	Incremental Computation of Taylor Series and System Jacobian in DAE solving using Automatic Differentiation
AUTHOR:	Xiao (Shawn) Li School of Computational Science and Engineering McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Nedialko S. Nedialkov
NUMBER OF PAGES:	viii, 67

Dedicated to my family and

true friends in my life.

Abstract

We propose two efficient automatic differentiation (AD) schemes to compute incrementally Taylor series and System Jacobian for solving differential-algebraic equations (DAEs) by Taylor series. Our schemes are based on topological ordering of a DAE's computational graph and then partitioning the topologically sorted nodes using structural information obtained from the DAE. Solving a DAE by Taylor series is carried out in stages. From one stage to another, partitions of the computational graph are incrementally activated so that we can reuse Taylor coefficients and gradients computed in previous stages. As a result, the computational complexity of evaluating a System Jacobian is independent of the number of stages.

We also develop a common subexpression elimination (CSE) method to build a compact computational graph through operator overloading. The CSE method is of linear time complexity, which makes it suitable as a preprocessing step for general operator overloaded computing. By applying CSE, all successive overloaded computation can save time and memory.

Furthermore, the computational graph of a DAE reveals its internal sparsity structure. Based on it, we devise an algorithm to propagate gradients in the forward mode of AD using compressed vectors. This algorithm can save both time and memory

when computing the System Jacobian for sparse DAEs. We have integrated our approaches into the DAETS solver. Computational results show multiple-fold speedups against two popular AD tools, FADBAD++ and ADOL-C, when solving various sparse and dense DAEs.

Acknowledgements

I thank my supervisor, Professor Ned Nedialkov, for introducing me to the area of automatic differentiation. Without the opportunity to work on DAETS, implementing an automatic differentiation tool with an incremental computing strategy for a DAE solver like DAETS would have been very difficult. It has been a pleasure to learn and work with him. I deeply appreciate his kind guidance and support.

I own my gratitude to Dr. Guangning Tan for the wisdom he shared with me both in academic and personal life. I am also grateful to my loyal friends, Zheng Gu and Lu Zhu, who gave me a lot of help and provided many needed hours of distraction. I thank my office mates, Hongsheng Zhong, Reza Zolfaghari, and John Ernsthausen, for many constructive conversations and happy hours in the graduate office.

Finally, I am in indebted to my family, for their love and support in my life. It is their encouragement that saw me through my graduate study abroad.

Contents

Chapter 1

Introduction

Integrating a system of differential-algebraic equations (DAEs) by Taylor series (TS) involves multiple stages to solve a sequence of nonlinear/linear systems of equations [26]. Since solving a nonlinear system requires solving a linear system in each iteration, the overall performance of a DAE solver based on TS boils down to the efficiency of two building blocks: (a) computing TS coefficients (right-hand side) and System Jacobian (left-hand side) via automatic differentiation (AD) to form the underlying linear system, and (b) dense/sparse linear algebra to solve it.

For part (b), excellent collections of numerical linear algebra routines [14] have been developed. In the Daets solver [25], we use LAPACK [23] for dense linear algebra and SuiteSparse [13] for sparse linear algebra. The performance of these linear solvers has been tuned for decades.

Our goal is to reduce the computing time in (a), which dominates the solution time by Daets, especially when a DAE is sparse. This thesis tackles part (a) by designing efficient AD schemes that exploit the overall DAE solution scheme and structure of the DAE.

1.1 Motivation

First and foremost, the importance of scalability is well recognized, and it is always attractive to see a numerical solver capable of solving larger and larger problems. Our aim is to have the growth rate of computing cost in part (a) as close to that of part (b) as possible.

Since there are multiple stages in the DAE solution scheme by TS (outlined in §2.1), there are opportunities to reuse gradients in the System Jacobian (§5.1) and Taylor series coefficients (TCs, §4) computed in previous stages. Motivated by this consideration, we study a DAE's computational graph to derive active partitions of that graph corresponding to data at different stages. Then, we develop an incremental computing strategy to recompute only those items that depend on the subset of the input variables that have changed.

DAEs can be large while sparse. The gradient patterns in the computational graph of a DAE embody its internal sparsity structure (presented in §5.2). Based on this structure, it is feasible to compute only nonzero components of gradients, which can lead to lower computational complexity of computing the System Jacobian when a DAE is sparse.

Besides scalability, we should also consider the user experience of writing down the code that describes the equations to be solved. It is possible to rearrange the code manually to eliminate common subexpressions in the equations. However, doing it by hand for large and complicated equations is cumbersome and error-prone, if not infeasible. The users of a DAE solver or AD tool should be free to provide equations in a way that is natural to them. They should not be worried about introducing duplicate subexpressions that would impact on performance. This consideration motivates

us to develop a common subexpression elimination (CSE) technique in the context of operator overloading computing to build automatically a compact computational graph.

Also, the DAE to be solved can be the result of differentiating another system, for example, the Euler-Lagrange equation in computer graphics [24] and mechanical systems [33]. In such scenarios, it may be impossible to eliminate identical subexpressions by hand, if the differentiation is automated (which should be the case in practical applications). One has to rely on CSE to obtain runtime and memory savings.

There are various state-of-the-art AD tools developed over the past decades [1]. They have their own advantages and disadvantages in terms of performance, ease of use, and richness of functionalities. The goal of this thesis is not to design a new AD package with the best performance ever. Rather, we aim to present the design of AD schemes at the application level, in particular, solving DAEs in this thesis. Instead of treating an application as a black box and simply using an existing AD package to compute derivatives, we wish to study the structure of our problem so we can derive efficient methods.

Finally, we hope that the idea of coupling AD with incremental computing would be enlightening to other problems involving multiple solving stages.

1.2 Contributions

The main contributions of this thesis are summarized as follows.

1) We propose a CSE method of linear-time computational complexity applied to a computational graph built via operator overloading. This method eliminates

nodes representing the same mathematical expression, leaving only one such node to be evaluated. This results in savings in both time and memory of successive AD computations.

2) We design two efficient incremental schemes for evaluating TCs and System Jacobian, respectively, by reusing TCs and gradients derived in previous stages, and recomputing only those TCs and gradients that have changed. As a result, the computational complexity of evaluating TCs throughout all stages is reduced from cubic to quadratic in terms of the required degree of TS, while the complexity of computing the System Jacobian across all stages is independent of the number of stages.

3) Based on the internal sparsity structure of the given equations, we devise a sparse gradient propagation method in the forward mode of AD. This method can lead to substantial runtime and memory savings in computing a sparse Jacobian.

4) By integrating the above methods into the DAETS solver, we have improved substantially its performance.

We hope that 1) and 3) will be of interest to the AD community, and computing in general.

1.3 Thesis organization

Chapter 2 reviews the theoretical background underpinning the work developed in this thesis. First, we give an overview of the solution scheme for solving DAE by TS. Then, we describe the concepts of code list and computational graph.

Chapter 3 presents the proposed CSE technique for operator overloading computing by showing the main framework and algorithms, and illustrates it with an example.

Chapter 4 describes the incremental computing scheme for evaluating TCs based on topological ordering and partitioning of the computational graph. Following that, we illustrate a reduction method to reduce the computational complexity with respect to the required degree of TS.

Chapter 5 derives methods for computing the System Jacobian of a DAE. First, we design an incremental computing scheme to amortize the overhead in evaluating this Jacobian across all stages. Then, we derive a sparse gradient propagation algorithm in the forward mode of AD based on the internal sparsity structure of the equations.

Chapter 6 presents computational results. We show the effectiveness of the proposed methods and report the overall performance of our AD schemes in solving dense and sparse DAEs

Chapter 7 draws conclusions and suggests directions for future work.

Chapter 2

Theoretical Background

We are interested in solving numerically an initial-value problem in differential-algebraic equations (DAEs) of the form

$$f_i(\, t, \text{ the } x_j \text{ and derivatives of them }) = 0, \quad i = 1 : n, \tag{2.1}$$

where the $x_j(t),\ j = 1 : n$, are state variables, and t is the time variable. The f_i can be arbitrary expressions built from the x_j and t using $+, -, *, /$, other analytic standard functions, and the d^p/dt^p operator. Solving DAE by TS needs to compute derivatives[1] of state variables (up to a required order), which involves solving a sequence of nonlinear/linear systems as presented in the following section.

[1]By a "derivative" $x_j^{(r)}$ we shall also mean $x_j^{(0)} = x_j$, $r = 0$.

2.1 Outline of DAE solution scheme

Here we present the solution scheme for solving DAE by TS based on Pryce's structural analysis (SA) [32]. This SA prescribes the systems of equations to be solved at different stages. More details can be found in [26, 27, 28].

This method constructs an $n \times n$ signature matrix $\mathbf{\Sigma} = (\sigma_{ij})$, whose (i, j) entry is defined as [32]

$$\sigma_{ij} = \begin{cases} \text{highest order of derivative to which } x_j \text{ occurs in } f_i; \text{ or} \\ -\infty \qquad \text{if } x_j \text{ does not occur in } f_i. \end{cases}$$

A *highest-value transversal* (HVT) of $\mathbf{\Sigma}$ is a set T of n positions (i, j) with one entry in each row and each column, such that the sum of these entries is the largest possible. Then we find two *valid* offset vectors $\mathbf{c} = (c_1, \ldots, c_n)$ and $\mathbf{d} = (d_1, \ldots, d_n)$, such that

$$c_i \geq 0 \quad \text{for all } i; \qquad d_j - c_i \geq \sigma_{ij} \quad \text{for all } i, j \text{ with equality on an HVT.} \tag{2.2}$$

A pair of $\mathbf{c}$ and $\mathbf{d}$ satisfying (2.2) is not unique. However, there exists a unique elementwise smallest pair of $\mathbf{c}$ and $\mathbf{d}$ for (2.2), which we refer to as the canonical offset pair [32]. Any valid pair of $\mathbf{c}$ and $\mathbf{d}$ can be used to prescribe a stage-by-stage scheme for solving DAEs by TS. The derivatives of state variables, for integrations by TS, are derived in stages.

Let $k_d = -\max_j d_j$. According to $\mathbf{c}$ and $\mathbf{d}$, at each stage k (for $k = k_d, k_d + 1, \ldots$)

we

$$\text{solve} \quad \left\{ f_i^{(k+c_i)} = 0 \mid k + c_i \geq 0 \right\} \tag{2.3}$$

$$\text{for} \quad \left\{ x_j^{(k+d_j)} \mid k + d_j \geq 0 \right\} \tag{2.4}$$

using values for $\left\{ x_j^{(r)} \mid 0 \leq r < k + d_j \right\}$, which are found at stages $< k$. We refer to (2.3, 2.4) as our DAE *solution scheme*.

The system (2.3) is generally nonlinear and underdetermined for $k < 0$. When $k = 0$, (2.3) can be either nonlinear or linear, while for $k > 0$, (2.3) is always linear. For $k \geq 0$, the number of variables is also equal to the number of equations.

In practice, when solving (2.1) numerically by TS, we compute TCs $x_j^{(k+d_j)}/(k + d_j)!$ directly; that is, we have TCs instead of derivatives in (2.3, 2.4). Since components in $\mathbf{c}$ are not necessarily equal, TCs of f_i's in (2.3) can be of different orders at a given stage.

Definition 2.1.1 *System Jacobian* $\mathbf{J}$ *is a* $n \times n$ *matrix, defined as*

$$\mathbf{J}_{ij} = \begin{cases} \dfrac{\partial f_i}{\partial x_j^{(d_j - c_i)}} & \text{if this derivative is present in } f_i, \\ 0 & \text{otherwise.} \end{cases} \tag{2.5}$$

Definition 2.1.2 *At stage* k, $\mathbf{J}_k$ *is a* $m_k \times n_k$ *submatrix of* $\mathbf{J}$*, formed by differentiating* (2.3) *with respect to* (2.4) [27]*; that is,*

$$\frac{\partial f_i^{(k+c_i)}}{\partial x_j^{(k+d_j)}} = \frac{\partial f_i}{\partial x_j^{(d_j - c_i)}}, \tag{2.6}$$

where m_k is the number of equations in (2.3), and n_k is the number of variables in (2.4).

The "=" in (2.6) is based on Griewank's Lemma [21] given below.

Lemma 2.1.1 (Griewank's Lemma) *Let v be a function of t, the $x_j(t)$, $j = 1, \ldots, n$, and derivatives of them. Denote $v^{(p)} = d^p v/dt^p$, where $p \geq 0$. If v does not depend on any derivative of x_j higher than the qth, then*

$$\frac{\partial v}{\partial x_j^{(q)}} = \frac{\partial v'}{\partial x_j^{(q+1)}} = \cdots = \frac{\partial v^{(p)}}{\partial x_j^{(q+p)}}. \tag{2.7}$$

To solve (2.3) either linearly or nonlinearly (typically via either Gauss-Newton or Newton's method), we need to form a system of linear equations. At stage $k \geq 0$, the matrix of this linear system is $\mathbf{J}$ [32]. At stage $k < 0$, the matrix is $\mathbf{J}_k$.

The right-hand side is a vector of TCs obtained by evaluating (2.3) with trial values for (2.4). They are obtained from previous stages, integration by TS from a previous time step at the first stage, or the user at the initial integration step. Denote the right-hand side vector by $\mathbf{b}$, and the vector of trial values and corrected values for (2.4) by $\mathbf{x}_t$ and $\mathbf{x}_c$, respectively. The underlying linear system at stage k is

$$\mathbf{J}_k(\mathbf{x}_t - \mathbf{x}_c) = \mathbf{b}.$$

This thesis will focus on the methods for efficient evaluations of $\mathbf{J}$ and TCs in $\mathbf{b}$ using automatic differentiation.

Example 2.1.1 The simple pendulum as a DAE is:

$$\begin{aligned} 0 = f &= x'' + x\lambda \\ 0 = g &= y'' + y\lambda - G \\ 0 = h &= x^2 + y^2 - L^2, \end{aligned} \tag{2.8}$$

where L (length) and G (gravity) are constants. The input variables are x, y, and λ; the output variables are f, g, and h. By Pryce's SA, we have a canonical offset pair: $\mathbf{c} = (0, 0, 2)$, $\mathbf{d} = (2, 2, 0)$, and $k_d = -2$ for this DAE. In Table 2.1, we illustrate the DAE solution scheme (2.3, 2.4) when applied to (2.8).

Table 2.1: Solution scheme for the simple pendulum DAE.

k	solve	for	$\mathbf{J}_k$
-2	$0 = h = x^2 + y^2 - L^2$	x, y	$[2x \quad 2y]$
-1	$0 = h' = 2(xx' + yy')$	x', y'	$[2x \quad 2y]$
0	$0 = f = x'' + x\lambda$		$[1 \quad 0 \quad x]$
	$0 = g = y'' + y\lambda - G$	x'', y'', λ	$[0 \quad 1 \quad y]$
	$0 = h'' = 2(x''x + x'^2 + y''y + y'^2)$		$[2x \quad 2y \quad 0]$
> 0	$0 = f^{(k)}, g^{(k)}, h^{(k+2)}$	$x^{(k+2)}, y^{(k+2)}, \lambda^{(k)}$	$\mathbf{J}$

2.2 Code list

Each f_i in (2.1) is described by an expression containing arithmetic operations, standard functions, and the d^p/dt^p operator. Such an expression can be represented by code list containing input, intermediate, and output variables as follows.

The input variables are t and x_j for $j = 1:n$. We rename them as $v_{-n} = t$ and $v_{j-n} = x_j$. Then each subsequent variable v_r for $r > 0$ is defined as the result of previous variables that v_r directly depends on through an elementary function ϕ_r. Each ϕ_r can be an arithmetic operation, a call to standard function, or d^p/dt^p. That is, ϕ_r represents an operation of either binary or unary form. We write

$$v_r = \phi_r(\{\, v_s \mid s \prec r \,\}), \tag{2.9}$$

where the precedence relation $s \prec r$ means that v_r directly depends on v_s and also implies $-n \leq s < r$. If $i \prec\ k \prec j$, then $i \prec^* j$, where $\prec^*$ is the transitive closure of $\prec$. $i \prec^* j$ indicates that v_j depends on v_i either directly or indirectly.

If an operand of a binary operator is a constant number, we refer to that number directly instead of assigning it to a variable. We refer to the last variable in the code list of an f_i as an output variable. Assuming that we evaluate all the f_i's and that the last n variables are output variables, we can illustrate the above as

$$\big[\underbrace{v_{-n}}_{t}, \underbrace{v_{1-n}, \ldots, v_0}_{x_j},\ v_1, \ldots, v_q,\ \underbrace{v_{q+1}, \ldots, v_{q+n}}_{f_i}\big], \tag{2.10}$$

where $v_1, \ldots, v_q$ are intermediate variables.

Example 2.2.1 For the simple pendulum DAE in (2.8), the input variables are x, y, and λ; the output variables are f, g, and h. Table 2.2 shows a possible code list for (2.8).

Table 2.2: Code list for the simple pendulum

		code list	expression
$x_1 =$	v_{-2}	$= x$	x
$x_2 =$	v_{-1}	$= y$	y
$x_3 =$	v_0	$= \lambda$	λ
	v_1	$= \mathrm{d}^2 v_{-2}/\mathrm{d}t^2$	x''
	v_2	$= v_{-2} * v_0$	$x\lambda$
	v_3	$= \mathrm{d}^2 v_{-1}/\mathrm{d}t^2$	y''
	v_4	$= v_{-1} * v_0$	$y\lambda$
	v_5	$= v_3 + v_4$	$y'' + y\lambda$
	v_6	$= \mathrm{sqr}\,(v_{-2})$	x^2
	v_7	$= \mathrm{sqr}\,(v_{-1})$	y^2
	v_8	$= v_6 + v_7$	$x^2 + y^2$
$f_1 =$	v_9	$= v_1 + v_2$	$f = x'' + x\lambda$
$f_2 =$	v_{10}	$= v_5 - G$	$g = y'' + y\lambda - G$
$f_3 =$	v_{11}	$= v_8 - L^2$	$h = x^2 + y^2 - L^2$

2.3 Computational graph

The techniques developed in this work are based on a computational graph [5] representing a code list.

Definition 2.3.1 *For a code list, its* computational graph *is a directed acyclic graph (DAG) $G = (V, E)$. We label the vertices with the variable names in the code list: $V = \{v_s \mid -n \leq s \leq q + n\}$. There is a directed edge from vertex v_s to vertex v_r, if and only if variable v_r depends directly on variable v_s: $E = \{(v_s, v_r) \mid s \prec r\}$.*

For an edge (v_s, v_r) in G, we say v_r is the parent of v_s, and v_s is the child of v_r.

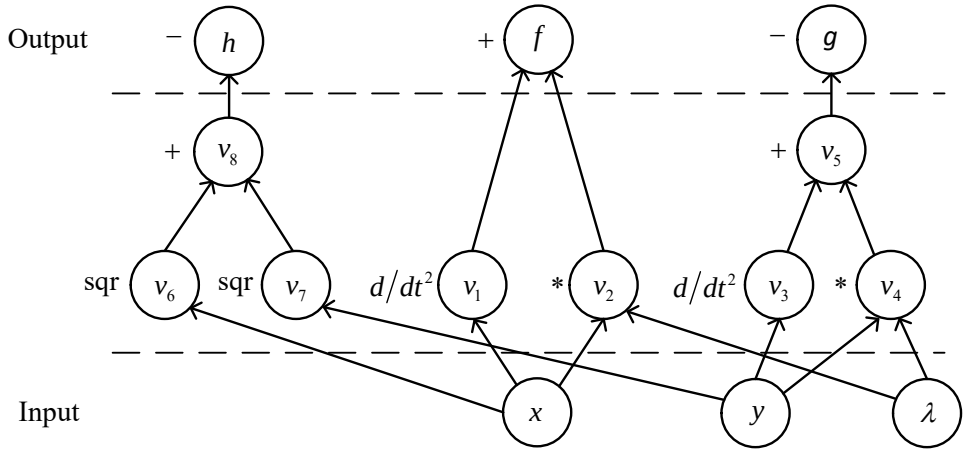


Figure 2.1: Computational graph for the code list in Table 2.2. The "−" at node h denotes the subtraction by the constant L. Similarly for node g: subtraction by the constant G.

Example 2.3.1 Figure 2.1 shows the computational graph corresponding to the code list in Table 2.2.

We denote the number of vertices and the number edges in graph $G = (V, E)$ by $|V|$ and $|E|$, respectively. The transpose of a directed graph G, denoted by G^T, is obtained by reversing the directions of the edges of G. The transpose G^T of a computational graph G is a *binary* DAG [35]. That is, at most two edges leave each node in G^T.

Figure 2.2 shows a topological ordering of the computational graph from Figure 2.1. Note that in this ordering, all the descendants of a given node must appear before the node itself. Thus, if we evaluate nodes v_s for $s = -n : q + n$, in the computational graph following its topological order, we can obtain the values of all corresponding expressions. Moreover, for any expression represented by a v_s, we can compute the value of that expression by evaluating all nodes in $\{ v_r \mid r \prec^* s \}$ and v_s itself with respect to this ordering.

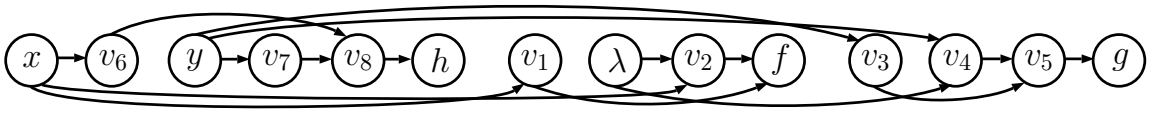


Figure 2.2: Topological ordering of the computational graph in Figure 2.1.

Note that the topological ordering of a computational graph is not necessarily unique. Different orderings may result in different performance, due to data locality and cache efficiency. However, we do not study the effect of various topological orderings in this thesis.

Definition 2.3.2 *A node is* shared *in a computational graph G, if this node is reachable from more than one node in G^T.*

It is not unusual for a set of f_i's in (2.1) to have common subexpressions. As a result, there are *shared* nodes in a computational graph built by evaluating (2.1). Accordingly, the evaluations of common subexpressions can be done by evaluating these shared nodes with respect to the topological ordering of f_i's computational graph.

To avoid the overhead of repeated dynamic memory allocations and temporary objects creations (typical in an operator overloaded implementation of AD), we build the computational graph in advance during a parsing stage, as described in next chapter.

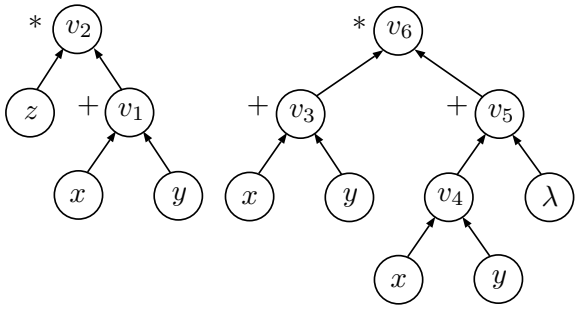
Chapter 3

Common Subexpression Elimination

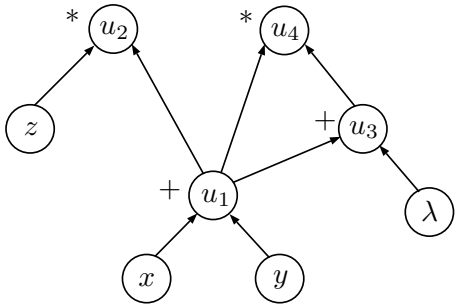
Common subexpression elimination (CSE) is a standard compiler optimization technique [22]. However, to the best of the author's knowledge, no compilers perform CSE when operators in expressions are overloaded. We develop a CSE technique that applies to a computational graph built through operator overloading. Before we present our method, we illustrate our idea on the following example.

Example 3.0.1 Suppose we are evaluating two expressions $z * (x + y)$ and $(x + y) * (x+y+\lambda)$. They can be represented by either (a) or (b) in Figure 3.1. In Figure 3.1(a), the subexpression $x + y$ would be evaluated three times. In Figure 3.1(b), the nodes v_1, v_3, and v_4 are merged into node u_1. Hence, the subexpression $x + y$ appears only once in the computational graph and would be evaluated only once.

In [35, §19.5], Sedgewick discusses how to convert the DAG in Figure 3.1(a) to the DAG in Figure 3.1(b). Our goal is, however, to build a computational graph like



(a)



(b)

Figure 3.1: Computational graphs of expressions $z*(x+y)$ and $(x+y)*(x+y+\lambda)$. (a) A DAG consists of two binary trees representing $z*(x+y)$ (left) and $(x+y)*(x+y+\lambda)$ (right), respectively. (b) A more compact DAG with common subexpression $x+y$ appearing only once.

the compact one in Figure 3.1(b) directly from the code list without constructing the one in Figure 3.1(a) first.

3.1 Framework of CSE

We show the flow digram of our CSE method in Figure 3.2. While processing the code list of expressions through operator overloading, we build a computational graph, $G = (V, E)$, by inserting nodes into V and edges into E as the operators are evaluated.

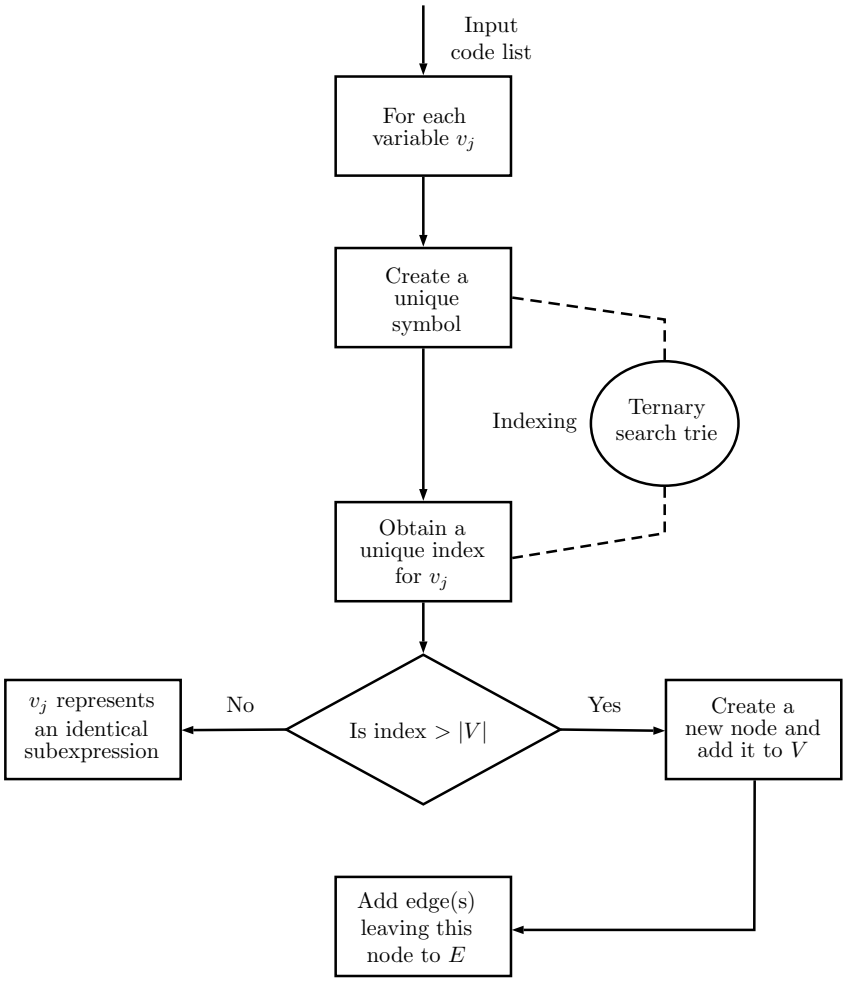


Figure 3.2: Flow diagram of the proposed CSE method. A ternary search trie is a data structure that implements a symbol table (see Definition 3.1.1).

Definition 3.1.1 *A symbol table is a collection of key-value mappings, where there are no duplicate keys, and each key maps to a unique value.*

We build a symbol table, where keys are string literal representations of symbols, while values are consecutive integer numbers starting from one. Such a symbol table can be implemented by the ternary search trie [34, §15].

We denote a query on table T with key e by $T.\texttt{map}(e)$. If T contains a mapping

for e, T.`map`(e) returns the mapped value of e. Otherwise, T.`map`(e) inserts a new mapping of e in T and then returns the current number of mappings in T, denoted by $|T|$; the mapped value of e is $|T|$.

We associate a symbol with each variable v_j (input, intermediate, or output), for $j = -n : q + n$. Then each v_j's symbol s is mapped to a unique integer by T.`map`(s). We refer to this integer as the *index* of v_j and denote this index by v_j.index. It starts from one and increases by one when a new symbol appears. Variables that represent same subexpressions must carry the same symbol, which maps to the same index value. This is how we recognize common subexpressions.

We process one variable at a time. For each variable v_j, if v_j.index $> |V|$, then no variables representing the same subexpression appear before v_j. Accordingly, we should create a new node for v_j. If v_j.index $\leq |V|$, then a variable representing the same subexpression has appeared before v_j. At the end of this process, variables representing the same subexpression are associated with a single node (e.g., u_1 in Figure 3.1(b)) in the resulting computational graph.

To make the above scheme work, we need to associate with each variable a symbol such that it characterizes a subexpression uniquely. Specifically, the variables representing the same subexpression should be associated with the same symbol, while those representing different subexpressions should be associated with different symbols.

For each input variable $v_{j-n} = x_j$, for $j = 1 : n$, we set the symbol associated with x_j to be `"j"`. For a non-input variable, we use either of the two routines in Figure 3.3 to create a unique string literal for such a variable, depending on whether its associated operator is binary or unary. These two routines rely on two subroutines, denoted

by `enc`$(\cdot)$ and `name`$(\cdot)$, which encode the operand and the operator, respectively. By encoding via `enc`(v), we mean

(a) the decimal literal representing the value of v.index, if v is a variable; or

(b) concatenating the decimal literal representing v's value with `"c"`, if v is a constant.

For instance, if v.index $= 10$, then `enc`(v) = `"10"` according to (a); if v = `1.23e-5`, then `enc`(v) = `"1.23e-5c"` according to (b). By `name(op)`, we mean the string encoding of an operator `op`, e.g., `"+"`, `"-"`, `"*"`, `"/"`, `"sin"`, `"cos"`, `"exp"`, etc.

1. `BinaryMap(op, l, r)`

   `op` is a binary operator, `l` is the left operand, and `r` is the right operand. `l` and `r` can be variables in the code list or numeric constants. `BinaryMap` returns a string literal by concatenation: `enc(l)` + `name(op)` + `enc(r)`.

2. `UnaryMap(op, g)`

   `op` is a unary operator, and `g` is the operand; `g` can be a variable in the code list or a constant number. `UnaryMap` returns a string literal by concatenation: `enc(g)` + `name(op)`.

Figure 3.3: Two routines that create string literal representations of symbols for variables: the first one for variables with binary operators and the second one for variables with unary operators.

3.2 Algorithms for CSE

Algorithm 3.2.1 describes how to index a variable in the code list and add new node to the computational graph accordingly. This algorithm returns true if and only if v_j represents a subexpression appearing before v_j.

Algorithm 3.2.1 INDEXING(v_j, T, V)

INPUT

variable v_j in a code list

symbol table T mapping symbols to unique indices

OUTPUT

vertex set V of computational graph $G(V, E)$

COMPUTE

% Create symbol s (string literal) for v_j

if v_j is an input variable ($j \leq 0$)

$s =$ "j"

else

if $v_j \equiv \texttt{op}(\texttt{l}, \texttt{r})$

$s = \texttt{BinaryMap}(\texttt{op}, \texttt{l}, \texttt{r})$

elseif $v_j \equiv \texttt{op}(\texttt{g})$

$s = \texttt{UnaryMap}(\texttt{op}, \texttt{g})$

% Setup index for v_j

v_j.index $= T.\texttt{map}(s)$

% Determine whether v_j represents a subexpression appearing before

if v_j.index $> |V|$

create a node $n_{v_j.\text{index}}$ and add it to V

return FALSE

else

return TRUE

The required time of $T.\texttt{map}(s)$ implemented by a ternary search trie is linearly proportional to the number of characters in a given symbol s. As every symbol consists of constant number of characters, mapping such symbol to an index by $T.\texttt{map}(s)$ runs in O(1) time and so does Algorithm 3.2.1.

Algorithm 3.2.2 presents the whole common subexpression elimination method. Since each variable directly depends on at most two variables, and $|T|$ is of $O(N)$, where N denotes the number of variables in the input code list, both the time complexity and space complexity of Algorithm 3.2.2 are $O(N)$.

Algorithm 3.2.2 COMMON SUBEXPRESSION ELIMINATION

INPUT

code list of given expressions

symbol table T mapping symbols to unique indices

OUTPUT

computational graph $G(V, E)$ with common subexpressions eliminated

COMPUTE

for each variable v_j in the code list

if INDEXING$(v_j, T, V) \neq$ TRUE

for each $i \prec j$

add edge $(n_{v_i.\mathsf{index}}, n_{v_j.\mathsf{index}})$ to E

Table 3.1: Process of building computational graph $G(V, E)$ for the expressions in Example 3.0.1 using Algorithm 3.2.2. The column $|V|$ represents the current size of V. The last two columns stand for the nodes and edges being added to $G(V, E)$, respectively; "$-$" denotes that no nodes or edges are created. Indices marked by ▒ indicate where the algorithm identifies a common subexpression.

code list	expression	symbol	index	$\|V\|$	node	edge(s)
$v_{-3} = x$	x	`"-3"`	$1 > 0$		n_1	$-$
$v_{-2} = y$	y	`"-2"`	$2 > 1$		n_2	$-$
$v_{-1} = z$	z	`"-1"`	$3 > 2$		n_3	$-$
$v_0 = \lambda$	λ	`"0"`	$4 > 3$		n_4	$-$
$v_1 = v_{-3} + v_{-2}$	$x + y$	`"1+2"`	$5 > 4$		n_5	$(n_1, n_5),\ (n_2, n_5)$
$v_2 = v_{-1} * v_1$	$z * (x + y)$	`"3*5"`	$6 > 5$		n_6	$(n_3, n_6),\ (n_5, n_6)$
$v_3 = v_{-3} + v_{-2}$	$x + y$	`"1+2"`	$5 < 6$		$-$	$-$
$v_4 = v_{-3} + v_{-2}$	$x + y$	`"1+2"`	$5 < 6$		$-$	$-$
$v_5 = v_4 + v_0$	$x + y + \lambda$	`"5+4"`	$7 > 6$		n_7	$(n_5, n_7),\ (n_4.n_7)$
$v_6 = v_3 * v_5$	$(x + y) * (x + y + \lambda)$	`"5*7"`	$8 > 7$		n_8	$(n_5, n_8),\ (n_7, n_8)$

Example 3.2.1 Using Algorithm 3.2.2 to build the computational graph for expressions in Example 3.0.1, we illustrate in Table 3.1 the process of this common subexpression elimination algorithm. At termination, it constructs a computational graph that is the same as the one in Figure 3.1(b).

It is important to point out that the proposed CSE technique is different from symbolic simplification. We do not consider whether they are symbolically equivalent such as $(a - b)^2 = a^2 + b^2 - 2ab$. However, we do recognize the commutativity of multiplication and addition in CSE by always putting the operand with smaller indices or constant number in the first place for the string concatenation in Figure 3.3.

Unlike symbolic simplifications, CSE is only of linear time complexity, which makes CSE more suitable as a preprocessing step before computing. All successive overloaded computing including automatic differentiation using either the forward mode or the reverse mode can save time and memory by processing a compact computational graph with fewer nodes.

As we shall see in Chapter 4 and Chapter 5, with CSE and topological ordering, we not only remove duplicate storage of nodes that represent same common subexpressions, but we also eliminate duplicate evaluations or visits of common subexpressions during automatic differentiation. For all the examples in Chapter 4 and Chapter 5, we assume CSE has been applied in the first place.

Chapter 4

Computing Taylor Series

Denote the pth Taylor series coefficient (TC) of a function $u(t)$ at some point t^* by

$$(u)_p = u^{(p)}/p!$$

Based on Taylor arithmetic [10], TCs of any variable in a code list (where all variables are differentiable) can be computed through a propagation of formal power series over the variables that this variable depends on (both directly and indirectly) [21, §13.2]. For example, if TCs of x and y are known to order p, the pth TCs for $x + y$, $x * y$ and x/y can be computed by

$$\begin{aligned}
(x+y)_p &= x_p + y_p, \\
(x*y)_p &= \sum_{r=0}^{p} (x)_r (y)_{p-r}, \quad \text{and} \\
(x/y)_p &= \frac{1}{(y)_0}\Big(x_p - \sum_{r=0}^{p-1} (y)_{p-r} (x/y\,)_r\Big).
\end{aligned}$$

Similar formulas can be established for every elementary function ϕ_r in (2.9) (see [21, §13.2]). There are existing AD packages, for example, TADIFF[1] [8] and ADOL-C [19], that implement these Taylor rules for computing TCs. The goal of this Chapter is to derive a scheme to compute incrementally TCs by using such formulas across all stages of the DAE solution scheme outlined in §2.1.

4.1 Topological ordering

Applying rules of Taylor arithmetic to expressions represented by nodes in the computational graph, TCs can be obtained through a graph traversal in topological order (see §2.3).

At a given stage, each $f_i^{(k+c_i)}$ in (2.3) may be evaluated up to different order due to components in $\mathbf{c}$ are not necessarily equal in DAE solution scheme (see Example 2.1.1). Also, at negative stages, only a subset of equations is activated, cf. (2.3). Hence, shared nodes (see Definition 2.3.2) should be evaluated up to the maximum possible order at a given stage. That is, for a shared node reachable from multiple output nodes representing f_i's in the transpose of the computational graph of (2.3), this node should be evaluated up to the maximum order required among all these f_i's at a given stage. In contrast to the top-down traversal from each output node, i.e., the TADIFF way [8], topological ordering guarantees all shared nodes are visited exactly once.

Definition 4.1.1 *At stage k, the* active nodes *in the computational graph G for computing TCs are those nodes reachable from the nodes representing the output variables f_i's, where $k + c_i \geq 0$, in G^T.*

[1]By TADIFF, we refer to the Taylor series implementation in FADBAD++ [7], which is derived from its predecessor with the same name TADIFF [8].

Algorithm 4.1.1 TOPOLOGICAL ORDERING OF COMPUTATIONAL GRAPH

INPUT

computational graph $G(V, E)$

equation sets $\mathcal{F}_k = \{f_i \mid k + c_i = 0\}$ for all $k = k_d, \ldots, 0$

OUTPUT

array $A[0, \ldots, |V| - 1]$ of references to nodes in topological order

array $B[0, \ldots, -k_d]$, where $B[-k]$ records a position in A such that

$A[0, \ldots, B[-k] - 1]$ stores references to nodes activated at stage $k \leq 0$

COMPUTE

for stages $k = k_d, \ldots, 0$

for each node $v_i \in V$ corresponding to $f \in \mathcal{F}_k$

DFS-VISIT(G^T, v_i, A)

$B[-k] =$ current size of A

DFS-VISIT(G, v_i, A)

mark node v_i visited

for each v_j s.t. $(v_i, v_j) \in E$

if node v_j is not visited

DFS-VISIT(G, v_j, A)

append the reference to node v_i at the end of array A

Algorithm 4.1.1 depicts the method to sort the nodes of a DAE's computational graph G in topological order using depth-first search (DFS). This algorithm puts

references (addresses in memory) to nodes of G into an output array A. Postorder numbering in DFS on G^T sorts the nodes into reverse topological order with respect to G^T [35]. Since the edges in G^T are in opposite directions to those in G, the nodes in A are in topological order with respect to G.

At stage k, we refer to an f_i with $k + c_i \geq 0$ as an *active* equation and an f_i with $k + c_i = 0$ as an *immediately active* equation, respectively. Equation set $\mathcal{F}_k$ contains immediately active equations at stage k such that $\mathcal{F}_k = \{f_i \mid k + c_i = 0\}$. We obtain $\mathcal{F}_k$ by Pryce's SA of a DAE [32] (see also §2.1).

In addition, Algorithm 4.1.1 forms a partition of G separated by $B[-k]$ for $k = k_d : 0$ (see Example 4.1.1). References to active nodes at stage $k \leq 0$ are stored in $A[0]$ through $A[B[-k] - 1]$. These nodes constitute a complete set of nodes needed to compute required TCs of active equations in (2.3) at stage $k \leq 0$. For positive stages, all the nodes in array A are activated and hence required for computing TCs.

Algorithm 4.1.1 also organizes the nodes with respect to an increasing order of stages where they are activated. Thus, the TCs of nodes computed at earlier stages can be reused for computing TCs of higher orders at later stages. This is because the required order of TCs for a node in the computational graph increases as the stage increases. Based on this fact, we can incrementally compute TCs from lower order to higher order across all the stages. Details are derived in the next section.

Example 4.1.1 Consider the simple pendulum DAE in Example 2.1.1. There are 3 non-positive stages staring from $k_d = -2$ with $\mathcal{F}_{-2} = \{h\}$, $\mathcal{F}_{-1} = \{\emptyset\}$, and $\mathcal{F}_0 = \{f, g\}$. Applying Algorithm 4.1.1 to this DAE's computational graph generates the array of node references in Figure 4.1. The first six nodes in array A are active through all stages, but the TCs of them are evaluated up to an increasing order as

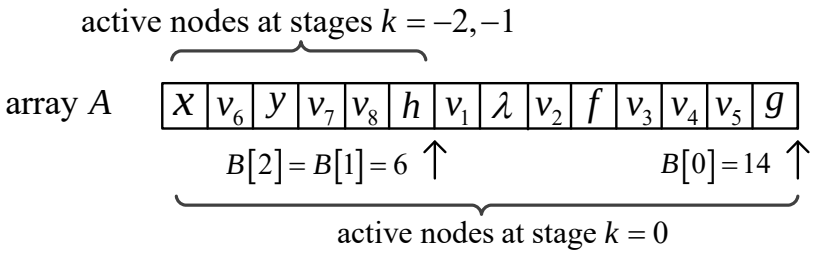


Figure 4.1: An array of node references in topological order as the result of applying Algorithm 4.1.1 to the computational graph of the simple pendulum DAE in Example 2.1.1. $B[-k]$ records the ending position of active nodes at a stage $k \leq 0$.

the stage increases. At stage $k = -2$, TCs of order 0 are computed, then order 1 at stage $k = -1$, order 2 at stage $k = 0$, etc. As we will see in the next section, the TCs of these nodes computed at previous stages can be reused to compute TCs of higher orders in the next stages.

4.2 Reduction for incremental computing

In the DAE solution scheme presented in §2.1, TCs are generated from lower order to higher order. That is we generate TCs of a variable in the code list following

$$(v_j)_0 \to (v_j)_1 \to \cdots \to (v_j)_p. \tag{4.1}$$

As addressed in [20] and [21, §13.5], if the $(v_j)_s$ for $s \leq p$ are reevaluated from scratch every time, the cost of computing TCs for each variable will end up with $O(p^3)$ floating point operations for every Taylor arithmetic of ϕ_r that involves a convolution. In [20], Griewank points out two ways to reduce the computational cost to $O(p^2)$. One approach is to use the so called coefficient doubling [21, §13.5]. Another approach

is to store and reuse TCs of each v_j already computed before in a code list. This approach is adopted, e.g., by ATOMFT [11] and TADIFF [8].

Definition 4.2.1 *The Taylor expansion point of each variable v_j in the code list is constituted by the input variables $v_{-n} = t$ and $v_{j-n} = x_j$, for $j = 1 : n$.*

Observe that any $(v_j)_p$ computed through a convolution is determined uniquely by all TCs $\{ (v_i)_s \mid i \prec j \text{ and } s \leq p \}$. Thus, if TCs in (4.1) are expanded at same point, they can be computed in a batch with $O(p^2)$ floating point operations by reusing the TCs already calculated along the way.

In TADIFF, a variable called *length* is also used to record how many TCs of a variable v_j have already been computed. When computing higher order TCs from successive propagations, these lower order TCs will be retrieved directly without recalculating. However, this storing approach works only if TCs are expanded at the same point, which is not always the case in our DAE solution scheme. This is because when solving (2.3) from stage to stage, we derive new TCs in (2.4). That is, TCs of input variables, $\left\{ (x_j)_{k+d_j} \mid 0 \leq j \leq n \right\}$, have changed from trial values for (2.3) to its solution, the length of each intermediate/output variable has to be reset to zero. Then, TCs of intermediate/output variables are calculated from scratch again. Therefore, $O(p^3)$ floating point operations are still required due to this resetting of *length*.

Since we store the computational graph in memory, it is natural to use the second approach. Here we introduce a new technique, which we name reduction, based on the storing approach. We formulate it in a way such that lower order TCs can be reused to compute higher order TCs even across stages of the DAE solution scheme, where the Taylor expansion point changes from stage to stage and from iteration

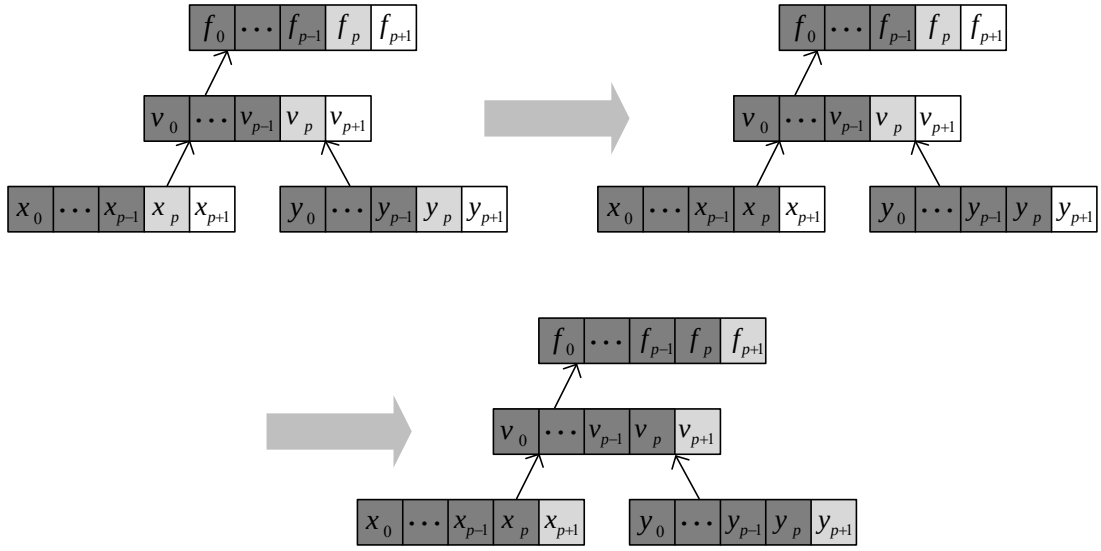


Figure 4.2: Computing Taylor series by reduction (TCs are written without parentheses for brevity). The top left graph presents the progress in propagating TCs at beginning of current stage with trial values trying to equate $(f)_p$ to zero; The top right graph reaches to the moment that we plug the solution back. After reduction, a new propagation of TCs in next stage leads to the bottom graph.

to iteration in solving (2.3). An illustration of this reduction method is shown in Figure 4.2. For the sake of brevity, we describe our method based on a simple code list

$$[x, y, v, f]$$

with only one intermediate variable v, two input variables x and y, and one output variable f, where $f = v = x \times y$. To keep track of the progress in propagating TCs, we color each square representing a TC dark gray, light gray, or white. A TC in dark gray already satisfies (2.3) for stages solved so far. A square in light gray means the corresponding TC is a trial value or an intermediate value just computed in an iteration for solving (2.3) at current stage. A TC corresponding to a square in white stands for the TC to be solved in next stage.

In Figure 4.2, the graph at top left shows TCs of x, y, v, and f at the beginning of a stage in solving (2.3) with trial values. At end of this stage, new TCs $(x)_p$ and $(y)_p$ equating $(f)_p$ to zero are obtained. Accordingly, we plug these TCs back so their colors become dark gray, which corresponds to the graph at top right in Figure 4.2. Now, as $(x)_p$ and $(y)_p$ have been updated, the expansion point (x, y) of $(f)_p$ has changed but the squares corresponding to $(v)_p$ and $(f)_p$ are still in light gray. This is because $(v)_p$ and $(f)_p$ have not been updated yet. In next stage, if we apply the storing approach directly for computing $(f)_{p+1}$, we end up using wrong $(v)_p$ and $(f)_p$ that are not updated.

To fix this problem, we need to reduce by one the length recording how many TCs have been computed for v and f so far (hence the name reduction). We reduce the length by one since only the last computed TC of each variable has changed in current stage, while the TCs up to length-1 stay unchanged. After a new propagation of TCs from x and y in next stage, we compute two new TCs for v and f, respectively. This computation runs in $O(p)$. At termination, $(v)_p$ and $(f)_p$ become dark gray while $(v)_{p+1}$ and $(f)_{p+1}$ become light gray, which leads to the bottom graph in Figure 4.2.

This reduction strategy also applies from iteration to iteration, where only the last computed TC of each variable is changing. To extend the above method to a code list in general form, we should reduce by one the length of computed TCs corresponding to each v_j with $j > 0$ in a code list at end of every stage and each iteration in solving (2.3). Then, the TCs staying unchanged (in dark gray) can always be reused to compute higher order TCs in linear time without recalculating from order 0, which maintains the time of computing (4.1) being quadratic in p even across stages and iterations.

Assume that at each stage $k \leq 0$, w iterations are required to solve (2.3) (nonlinearly). Suppose a node representing v_j is activated at stage $-c_j$, and p is the maximum required order of input variables' TCs. Then, the time complexity of computing TCs for each node in a DAE's computational graph is $O(wc_j^2)$ through stages $k \leq 0$, and $O(p^2)$ through stages $k > 0$.

Chapter 5

Evaluating Jacobian

The definition of the System Jacobian $\mathbf{J}$ is given in §2.1. The goal of this Chapter is to derive methods for computing $\mathbf{J}$ efficiently. In the first section, we derive an incremental computing scheme to amortize the overhead in evaluating this Jacobian across all stages. In the second section, we derive a sparse gradient propagation algorithm in the forward mode of AD based on the sparsity structure of the given equations.

5.1 Amortizing overhead across stages

An important feature of the System Jacobian $\mathbf{J}$ is that its symbolic form is unchanged through all stages of the DAE solution scheme [27] presented in §2.1. At stage k of this scheme, denote by $\mathbf{J}_k$ the matrix of the linear system to be solved at this stage. If $k < 0$, $\mathbf{J}_k$ is a submatrix of $\mathbf{J}$ by keeping only rows where $k + c_i \geq 0$ and columns where $k + d_j \geq 0$; If $k \geq 0$, $\mathbf{J}_k = \mathbf{J}$. In addition, $\mathbf{J}_k$ is a submatrix of $\mathbf{J}_{k+1}$ for $k < 0$ as well. The purpose of this section is to derive a method for computing $\mathbf{J}_k$ incrementally

from stage to stage, such that the overhead in computing $\mathbf{J}$ is amortized across all stages.

Denote

$$\nabla_k = \left(\frac{\partial}{\partial x_1^{(k+d_1)}}, \ldots, \frac{\partial}{\partial x_n^{(k+d_n)}} \right). \tag{5.1}$$

If $k + d_j < 0$, the jth component of ∇_k is zero due to absence of x_j in (2.3) at stage k.

Lemma 5.1.1 *For computing* $\mathbf{J}$ *across all sages, the following holds.*

(i) For any $k > k^*$, $\nabla_k \left(f_i^{(k+c_i)} \right) = \nabla_{k^*} \left(f_i^{(k^*+c_i)} \right)$.

(ii) $\nabla_k \left(f_i^{(k+c_i)} \right)$ *is the* i*th row of* $\mathbf{J}$ *for all* $k + c_i \geq 0$ *and its components with* $k + d_j \geq 0$ *form that row of* $\mathbf{J}_k$.

Proof for Lemma 5.1.1 is given in Appendix A. Based on Lemma 5.1.1, once the non-zero components of a row in $\mathbf{J}$ are found at a previous stage, we do not need to recompute them, as their numerical values are fixed after the last iteration of the (nonlinear) solving for (2.3) at the previous stage. That is, we only need to evaluate $\nabla_k \left(f_i^{(k+c_i)} \right)$ at stage k^* where $k^* + c_i = 0$ and then reuse the non-zero components of this gradient in following stages $k > k^*$.

Definition 5.1.1 *At stage* k*, the* active nodes *in the computational graph* G *for computing* $\mathbf{J}_k$ *are those nodes reachable from the nodes representing the output variables* f_i*'s, where* $k + c_i = 0$*, in* G^T*, excluding the nodes already activated at previous stages.*

We can partition G according to the active nodes at different stages $k \leq 0$. The gradients of nodes activated at earlier stages are fixed once we pass the stage where

they are activated. This is because a gradient's symbolic form is consistent though all stages as the result of Lemma 5.1.1. It is important to exclude nodes activated at previous stages, when forming a partition, such that partitions will not overlap with each other. This indicates no shared nodes are visited more than once even they may be reachable from nodes activated at different stages in G^T.

From stage k to stage $k+1$, a traversal in topological order over the active nodes at stage $k+1$ computes the new rows in $\mathbf{J}_{k+1}$ that are different from $\mathbf{J}_k$. Combining these new rows and the same rows in $\mathbf{J}_k$, we obtain $\mathbf{J}_{k+1}$. Repeating this process across all stages forms an *incremental computing* scheme. In this process, we only compute the gradients corresponding to newly active nodes at a stage, and hence the overhead of computing $\mathbf{J}$ is amortized across all stages $k \leq 0$. Figure 5.1 shows the partitioning of the computational graph of the DAE in Example 2.1.1.

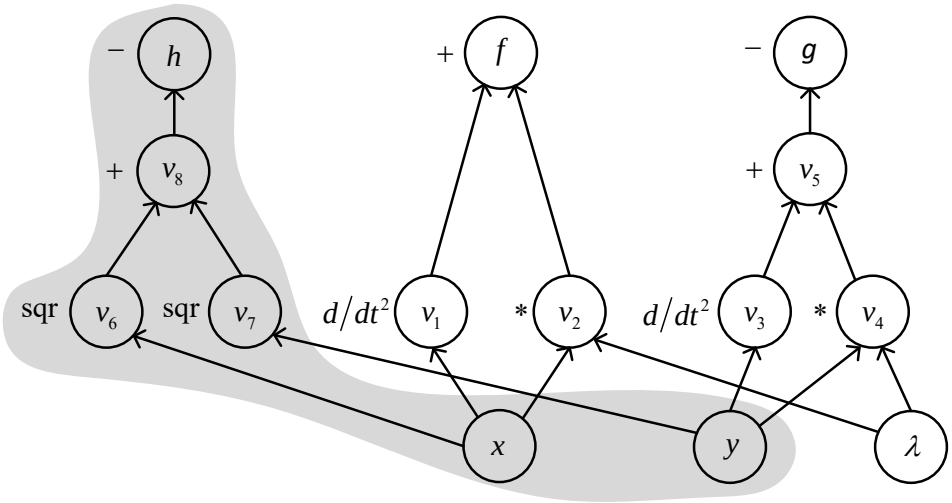


Figure 5.1: A partitioned computational graph of the simple pendulum DAE. The nodes in the gray region are active at stage $k = -2$, while the rest are active at stage $k = 0$. At stage $k = -1$, no nodes are active; hence no computations are required. Even though node f depends on node x, and node g depends on node y at stage $k = 0$, nodes x and y are visited only at stage $k = -2$ according to this partitioning.

Algorithm 5.1.1 PARTITIONING OF COMPUTATIONAL GRAPH

INPUT

computational graph $G(V, E)$

equation sets $\mathcal{F}_k = \{f_i \mid k + c_i = 0\}$ for all $k = k_d, \ldots, 0$

OUTPUT

array $A[0, \ldots, |V| - 1]$ of references to nodes in topological order

array $P[0, \ldots, -k_d + 1]$ such that $A[P[-k + 1]]$ through $A[P[-k] - 1]$ stores references to nodes activated at stage $k \leq 0$

COMPUTE

$P[-k_d + 1] = 0$

for stages $k = k_d, \ldots, 0$

for each node $v_i \in V$ corresponding to $f \in \mathcal{F}_k$

DFS-VISIT(G^T, v_i, A)

$P[-k] =$ current size of A

Algorithm 5.1.1 depicts the method to partition the computational graph of a DAE for computing $\mathbf{J}$. Similar to Algorithm 4.1.1, Algorithm 5.1.1 uses DFS to sort the references to nodes in G topologically into an array A. In addition, Algorithm 5.1.1 forms partitions of G for stages $k \leq 0$. References to nodes in the partition for stage k are stored in $A[P[-k+1]]$ through $A[P[-k]-1]$ with respect to a topological order. Using the chain rule for gradient calculation [21], a loop over $A[P[-k+1], \ldots, P[-k]-1]$ can compute the gradients required at stage k in the forward mode. Provided that

CSE has been applied to G, there are no references to duplicate nodes in array A. As a result, Algorithm 5.1.1 guarantees the gradients of common subexpression are evaluated exactly once in every gradient propagation in the forward mode.

Denote the cost of evaluating (2.1) by $\text{cost}(F)$, and assume that at each stage, w iterations are required to solve (2.3) nonlinearly. Usually, $w = 1$ or 2. The partitions at all stages $k \leq 0$ cover all the nodes in a DAE's computational graph exactly once. Therefore, the cost of traversing active nodes through all stages $k \leq 0$ in each iteration is $O(\text{cost}(F))$. Then the overall time complexity for computing all $\mathbf{J}_k$'s ($k \leq 0$) in the forward mode of AD is $O(nw \cdot \text{cost}(F))$, where n is the number of equations, independent of the number of stages. This complexity is achieved by the proposed incremental computing scheme that amortizes the overhead across stages.

5.2 Exploiting sparsity

As for computing the Jacobian using the forward mode or reverse mode, there are generally three approaches to exploiting sparsity [21].

- Static approach detects and exploits sparsity at compile time. For example, this approach has been implemented in the source-to-source transformation tool TAF [18].

- Pseudo-static approach relies on a priori analysis of the sparsity structure of a Jacobian or functions. Leveraging the structural information from a priori analysis, successive evaluations of Jacobian can be done efficiently. Curtis-Powell-Reid seeding [12] and Newsam-Ramsdell seeding [29] are two classical examples exploiting the sparsity structure of a Jacobian to compress it into an

equivalent one with fewer dimensions.

- Dynamic approach uses dynamically allocated data structure at runtime to store gradients and process only the nonzero components of gradients. The implementation of this idea in the context of AD was introduced by [16] followed by [4] and [9]. It has been shown in [21] that, in theory, the dynamic approaches have lower time complexity than the aforementioned compression approaches. However, a dynamic approach may render a high overhead in processing the dynamically allocated structure at runtime [21].

To avoid the runtime penalty of the dynamic approach, we devise a new data representation called *compressed vector* to store gradients. Like the dynamic approach, our method computes only nonzero components in the gradient. Hence, our method has the same time complexity (see Algorithm 5.2.1) as the dynamic approach using the forward mode. However, by learning the structure of a given function, we derive the sizes and index mappings of all *compressed vectors* before gradient propagations.

We use sparsity analysis of gradients to gather such information based on a given function's computational graph. This preprocessing step sets up all necessary information for all successive gradient propagations using the forward mode. Based on the sparsity analysis of gradients, we can allocate memory for all *compressed vectors* at the beginning and free their memory after all successive gradient propagations are finished. Hence, the overhead of repeated memory allocations are circumvented. Since the preprocessing step needs to be done only once at beginning, we identify our method as a pseudo-static approach.

First, to help us learn the structure of a given vector function $F(t, \boldsymbol{x}) \in \mathbb{R}^n$, we define the following two index sets: index domain (Definition 5.2.1) [21], and index

span (Definition 5.2.2). The length of F's code list is $N = 2n + q$, cf. (2.10).

Definition 5.2.1 *The* index domain $\mathcal{X}_k$ *of a variable* v_k *in* F*'s code list (except for* t*) is the set of indices of input variables on which* v_k *depends* [21]*. That is,*

$$\mathcal{X}_k \equiv \left\{ 0 \le j \le n : j - n \prec^* k \right\} \qquad \textit{for} \quad k = 1 - n : q + n \ .$$

By definition, the index domain $\mathcal{X}_k$ contains all the indices of ∇v_k's components that can be nonzeros. In F's computational graph, $\mathcal{X}_k$ can be computed by taking the union of the index domains of v_k's children:

$$\mathcal{X}_k = \bigcup_{j \prec k} \mathcal{X}_j \quad \text{from} \quad \mathcal{X}_{j-n} = \{ j \} \quad \text{for} \quad 1 \le j \le n \text{ [21].} \tag{5.2}$$

We can use the index domain size $|\mathcal{X}_k|$ to quantify the sparsity of ∇v_k (the number of nonzero components in ∇v_k). We call

$$\bar{n} \equiv \frac{\sum_{k=1}^{q+n} |\mathcal{X}_k|}{q+n}$$

the average domain size of F. We say F is sparse if $\bar{n}$ is small relative to n.

Definition 5.2.2 *The* index span $\mathcal{S}_k$ *of a variable* v_k *in* F*'s code list (except for* t*) is the union of index domains of* v_k*'s parents in* F*'s computational graph. That is,*

$$\mathcal{S}_k \equiv \begin{cases} \bigcup_{i \succ k} \mathcal{X}_i & \textit{for} \quad k = 1 - n : q \qquad \text{(non-output variables)}, \\ \mathcal{X}_k & \textit{for} \quad k = q + 1 : q + n \quad \text{(output variables)}. \end{cases} \tag{5.3}$$

After $\mathcal{X}_k$ of all variables in F's code list are obtained, $\mathcal{S}_k$ can be collected in F's

computational graph by applying the above definition formula in reverse topological order. We call

$$\bar{s} \equiv \frac{\sum_{k=1-n}^{q+n} |\mathcal{S}_k|}{N-1}$$

the average span size of F.

Definition 5.2.3 *A jth component in ∇v_k is a* placeholder *in ∇v_k, if j is in $\mathcal{S}_k$ but not in $\mathcal{X}_k$.*

By Definition 5.2.1 and Definition 5.2.2, we have $\mathcal{X}_k \subseteq \mathcal{S}_k$. The trivial case is $\mathcal{S}_k = \mathcal{X}_k$. In this case, $\mathcal{S}_k$ is the set of indices of nonzero components in ∇v_k. Otherwise, the nonempty set $\mathcal{S}_k \setminus \mathcal{X}_k$ contains the indices of placeholders in ∇v_k. The components corresponding to these placeholders are always zeros. Their purpose is to pad the sparse gradient so their entries are aligned in the vector form. This idea is illustrated on the following example.

Example 5.2.1 Consider the following two possible scenarios when $\mathcal{S}_k \neq \mathcal{X}_k$.

1. Assume that v_k has a single parent v_g and v_g has another child v_h in F's computational graph and $\mathcal{X}_k \neq \mathcal{X}_h$. Then $\mathcal{X}_k \subset \mathcal{S}_k = \mathcal{X}_g$ and $\mathcal{X}_h \subset \mathcal{S}_h = \mathcal{X}_g$. Both hold since $\mathcal{X}_k \neq \mathcal{X}_h$. The indices of placeholders in ∇v_k are all in $\mathcal{S}_k \setminus \mathcal{X}_k = \mathcal{X}_g \setminus \mathcal{X}_k$. The indices of placeholders in ∇v_h are all in $\mathcal{S}_h \setminus \mathcal{X}_h = \mathcal{X}_g \setminus \mathcal{X}_h$. These placeholders must exist when carrying out

$$\nabla v_g = \frac{\partial v_g}{\partial v_k} \nabla v_k + \frac{\partial v_g}{\partial v_h} \nabla v_h$$

in the vector form as illustrated by the following example

$$\begin{bmatrix} \times \\ \\ \times \\ \\ \times \end{bmatrix} = \begin{bmatrix} \times \\ \\ \times \\ \\ \bigcirc \end{bmatrix} + \begin{bmatrix} \bigcirc \\ \\ \times \\ \\ \times \end{bmatrix},$$

where $\times$ represents a nonzero entry and $\bigcirc$ represents a placeholder, respectively, while other zero entries are left blank. Here, $\mathcal{S}_k = \mathcal{S}_h = \mathcal{X}_g = \{1, 3, 5\}$, $\mathcal{X}_k = \{1, 3\}$, and $\mathcal{X}_h = \{3, 5\}$.

2. If v_k has multiple parents in F's computational graph, then

$$\mathcal{S}_k \setminus \mathcal{X}_k = \bigcup_{p \succ k} \mathcal{X}_p \setminus \mathcal{X}_k \ .$$

Thus, $\mathcal{S}_k \setminus \mathcal{X}_k$ contains all the indices of those placeholders in ∇v_k that must *exist* to carry out ∇v_p for any $p \succ k$ in the vector form.

As illustrated in Example 5.2.1, the index span $\mathcal{S}_k$ contains all the indices of ∇v_k's components that must exist as either nonzero or placeholder to carry out propagation of gradients in the vector form using the forward mode. Thus, the index span size $|\mathcal{S}_k|$ is the least number of components in ∇v_k that are needed to propagate gradients in our method.

Observe that $|\mathcal{X}_k| \leq |\mathcal{S}_k| \leq n$. For propagating ∇v_k's nonzero components using the forward mode, $|\mathcal{X}_k|$ is a measure of computational effort, while $|\mathcal{S}_k|$ is a measure of storage requirements. Both $\bar{n}$ and $\bar{s}$ would be much smaller than n if F is very sparse.

The objective is to exploit these structure properties of F to save both time and memory when computing gradients in the forward mode, when F is of considerable sparsity.

Denote by $(\nabla v_k)_j$ the jth component of ∇v_k. Algorithm 5.2.1 describes how to derive ∇v_k's nonzero components for all variables in F's code list using each variable's $\mathcal{X}_k$. The ith row of F's Jacobian $\mathbf{J}$ is ∇v_{q+i}.

Algorithm 5.2.1 Basic Sparse Gradient Propagation

Input

F's code list $\left[v_{-n}, v_{1-n}, \ldots, v_0,\ v_1, \ldots, v_q,\ v_{q+1}, \ldots, v_{q+n}\right]$

index span $\mathcal{X}_k$ of each variable v_k in F's code list

Output

∇v_{q+i} for $i = 1 : n$

Compute

for $k = 1 - n$ **to** 0

$$(\nabla v_k)_{n+k} = 1$$

for $k = 1$ **to** $q + n$

for each $j \in \mathcal{X}_k$

$$(\nabla v_k)_j = \sum_{i \prec k} \frac{\partial v_k}{\partial v_i} \times (\nabla v_i)_j$$

In Algorithm 5.2.1, the data representation for each ∇v_k is a vector of size n. Hence, $(\nabla v_k)_j$ is the jth element of that vector. The gradient calculation in the last line runs in $O(1)$ as v_k directly depends on at most two variables. This line is

executed $\sum_{k=1}^{q+n} |\mathcal{X}_k| = \bar{n} \cdot (n+q)$ times, where $n+q$ is the number of non-input variables. Denote the cost of evaluating F by $\text{cost}(F)$, which is of $O(n+q)$. Therefore, the time complexity of Algorithm 5.2.1 is $O(\bar{n} \cdot \text{cost}(F))$, while the space complexity of this algorithm is $O(nN)$ due to storing each variable's gradient in a vector of size n.

If F is sparse, then $\bar{n}$ is small relative to n, which leads to a lower time complexity than $O(n \cdot \text{cost}(F))$ of finite differences and the dense gradient propagation in the forward mode. The space complexity is, however, the same as that of the dense gradient propagation in the forward mode. This preliminary data presentation not only wastes memory when F is sparse, it also fails to preserve data locality, especially when ∇v_k's nonzero components are scattered throughout the size n vector. In such scenario, cache efficiency is downgraded.

We can overcome the foregoing disadvantages of Algorithm 5.2.1 and reduce its space complexity by further utilizing each variable's $\mathcal{S}_k$. Recall that $|\mathcal{S}_k|$ is the least number of components in ∇v_k required to propagate gradients of the vector form (cf., Example 5.2.1) in the forward mode. Based on this property, we propose a new data representation which we call *compressed vector* $G_k[1 \dots |\mathcal{S}_k|]$ to store only necessary components (nonzero components and placeholders) of ∇v_k for gradient propagations. For a variable v_k, consider $\mathcal{S}_k$ as an ordered set of indicies in an increasing order. Then we can build the mapping

$$\gamma_k : \mathcal{S}_k \longmapsto \{\, 1 \dots |\mathcal{S}_k| \,\} \tag{5.4}$$

such that $\gamma_k(\mathcal{S}_{k,j}) = j$, where $\mathcal{S}_{k,j}$ denotes the jth element in $\mathcal{S}_k$. Like $\mathcal{X}_k$ and

$\mathcal{S}_k$, γ_k for each variable can be constructed in advance through a traversal on F's computational graph before the gradient propagation.

We summarize our method to set up $\mathcal{X}_k$, $\mathcal{S}_k$, and γ_k based on F's computational graph in Algorithm 5.2.2. We refer to it as *sparsity analysis* for gradient propagations. Since $\mathcal{X}_k$, $\mathcal{S}_k$, and γ_k will not change throughout all successive propagations, this sparsity analysis needs to be done only once as a preprocessing step. Moreover, Algorithm 5.2.2 does not require the user to specify the sparsity pattern, hence exploiting the sparsity transparently.

Algorithm 5.2.2 SPARSITY ANALYSIS

INPUT

F's computational graph G

OUTPUT

$\mathcal{X}_k$, $\mathcal{S}_k$, and γ_k, for $k = 1 - n : q + n$

COMPUTE

1. Visit each node v_k in G in topological order to compute $\mathcal{X}_k$ by (5.2).
2. Visit each node v_k in G in reverse topological order to compute $\mathcal{S}_k$ by (5.3).
3. Traverse each node v_k in G (by either DFS or BFS) to compute γ_k by (5.4).

For $j \in \mathcal{S}_k$, we store the component $(\nabla v_k)_j$ in $G_k[\gamma_k(j)]$. Recall that $\mathcal{X}_k \subseteq \mathcal{S}_k$. Also, for $i \prec k$, $\mathcal{X}_k \subseteq \mathcal{S}_i$ by Definition 5.2.2. Accordingly, we have $(\nabla v_k)_j = G_k[\gamma_k(j)]$ and $(\nabla v_i)_j = G_i[\gamma_i(j)]$ for any $j \in \mathcal{X}_k$ and $i \prec k$. Based on this, we derive Algorithm 5.2.3 using compressed vectors to store gradients. The nonzero components in ith row of $\mathbf{J}$ are all in vector G_{q+i}.

Algorithm 5.2.3 Improved Sparse Gradient Propagation

Input

F's code list $\left[v_{-n}, v_{1-n}, \ldots, v_0,\ v_1, \ldots, v_q,\ v_{q+1}, \ldots, v_{q+n}\right]$

index span $\mathcal{X}_k$ of each variable v_k in F's code list

mapping γ_k for each variable v_k in F's code list

Output

G_{q+i} for $i = 1 : n$

Compute

for $k = 1 - n$ to 0

$$G_k[\gamma_k(n+k)] = 1$$

for $k = 1$ to $q + n$

for each $j \in \mathcal{X}_k$

$$G_k[\gamma_k(j)] = \sum_{i \prec k} \frac{\partial v_k}{\partial v_i} \times G_i[\gamma_i(j)]$$

Like Algorithm 5.2.1, Algorithm 5.2.3 also executes the gradient calculation in the last line for $\sum_{k=1}^{q+n} |\mathcal{X}_k| = \bar{n} \cdot (n+q)$ times. So these two algorithms have the same time complexity $O(\bar{n} \cdot \text{cost}(F))$. However, since algorithm 5.2.3 uses compressed vectors to store gradients, these vectors hold $\sum_{k=1-n}^{q+n} |\mathcal{S}_k| = \bar{s} \cdot (N-1)$ elements instead of $n(N-1)$. The space complexity is reduced to $O(\bar{s}N)$. Contrary to the size n vector, the nonzero components in the compressed vector try to stay close to each other as it has smaller size in general. Thus, the gradient calculations in Algorithm 5.2.3 have a better data locality than that of Algorithm 5.2.1.

It is expected that $\bar{n}$ and $\bar{s}$ are small relative to n when F is sparse. In such

a scenario, Algorithm 5.2.3 saves both time and memory in contrast to the dense gradient propagation method.

Definition 5.2.4 *The* offset *of a variable v in F's code list is defined* [27] *as*

$$\alpha(v) = \min_j \big(d_j - \sigma_j(v)\big),$$

where

$$\sigma_j(v) = \begin{cases} \textit{the highest order derivative of } x_j \textit{ on which } v \textit{ formally depends, or} \\ -\infty \quad \textit{if } v \textit{ does not depend on } x_j. \end{cases}$$

To extend Algorithm 5.2.3 for computing System Jacobian instead of general Jacobian, we need to incorporate offsets of variables [27]. Using Algorithm 5.1 in [27], we can obtain the offset of each variable in F's code list. Carrying these offsets, we derive Algorithm 5.2.4 to compute System Jacobian through sparse gradient propagation in the forward mode of AD. The nonzero components in ith row of System Jacobian are all in vector G_{q+i}.

Algorithm 5.2.4 Compute Sparse System Jacobian

Input

F's code list $\big[v_{-n}, v_{1-n}, \ldots, v_0,\ v_1, \ldots, v_q,\ v_{q+1}, \ldots, v_{q+n}\big]$

index span $\mathcal{X}_k$ of each variable v_k in F's code list

mapping γ_k for each variable v_k in F's code list

offset $\alpha(v_k)$ of each variable v_k in F's code list

OUTPUT

G_{q+i} for $i = 1 : n$

COMPUTE

for $k = 1 - n$ to 0

$G_k[\gamma_k(n+k)] = 1$

for $k = 1$ to $q + n$

if $v_k \equiv du^p/dt^p$ for a $u \prec v_k$

for each $j \in \mathcal{X}_k$

$G_k[\gamma_k(j)] = G_u[\gamma_u(j)]$

else

for each $j \in \mathcal{X}_k$

$$G_k[\gamma_k(j)] = \sum_{\substack{i \prec k \\ \alpha(v_k)=\alpha(v_i)}} \frac{\partial v_k}{\partial v_i} \times G_i[\gamma_i(j)]$$

Chapter 6

Computational Results

In this chapter, we assess the proposed AD techniques and the overall performance of our AD schemes in solving several dense and sparse DAEs. All tests are conducted on a 2016 MacBook Pro laptop with a 2.9 GHz Intel Core i5 processor and 8 GB RAM, running `macOS` Sierra 10.12. The `C++` compiler is GCC 5.4.0.

6.1 Case studies of CSE

In this section, we apply the proposed CSE techniques on equations of six practical problems. The first three problems are drawn from the Minpack-2 test suite [3]. The last three problems are: the Spring-Mass-Pendulum (with 30 pendulum), which is a mechanics modeling system taken from [38], the Distillation Column, which is an chemical engineering problem presented in [36], and the Simulated Moving Bed, which is a dynamic optimization problem developed in [15]. For the Spring-Mass-Pendulum, CSE is performed when differentiating the original system by AD in the backward mode [33].

Table 6.1: Effect of CSE on equations of six practical problems. $n \to m$ denotes n input variables and m output variables. "#operators w/o CSE" and "#operators w/ CSE" stand for the number of operators required for AD without and with applying CSE, respectively.

Case	$n \to m$	#operators w/o CSE	#operators w/ CSE	Elimination Ratio
`Elastic-Plastic Torsion`	$2,500 \to 1$	63,043	46,581	.26
`Driven Cavity`	$14,400 \to 14,400$	1,024,567	420,154	.59
`2-D Ginzburg-Landau`	$3,600 \to 1$	52,930	24,748	.53
`Spring-Mass-Pendulum`	$90 \to 90$	16,038	5,211	.68
`Distillation Column`	$207 \to 207$	10,799	6,216	.42
`Simulated Moving Bed`	$8,580 \to 8,570$	107,936	49,456	.54

In the equations of these problems, the operators are overloaded for AD. By CSE, we eliminate the variables standing for duplicate subexpressions in the computational graph, thereby reducing the operators needed to carry out AD. We assess the effect of CSE by the elimination ratio, which is defined as the number of operators reduced by CSE divided by the number of operators required without applying CSE. The higher the eliminations ratio is, the less the time and memory are required to carry out successive operator overloading computations.

The results are given in Table 6.1. The elimination ratio ranges from 26% to 68%. This indicates that a good portion of common subexpressions can be eliminated when we have nontrivial modeling equations. The CSE technique handles the elimination automatically instead of requiring manually rearranging the code of modeling equations, which can be cumbersome and error-prone when the complexity of equations is

high. It also supports the case when the system of equations is the result of differentiating another system (see details in [33]), for example, the Spring-Mass-Pendulum system used here.

6.2 Speed test for computing Taylor series

In this section, we compare the speed of computing Taylor series by our AD with the speed of that by other operator overloading AD tools including CppAD [6], ADOL-C [19], Sacado [31], and FADBAD++ [7]. We build the speed tests based on the `poly` speed benchmark in CppAD [6]. The speed comparisons are assessed by the rates of computing $2rd$ order TCs of polynomials in two forms:

1. nested form (also known as Horner scheme [30])

$$p(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x(a_n)) \cdots)),$$

2. simplified form

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n.$$

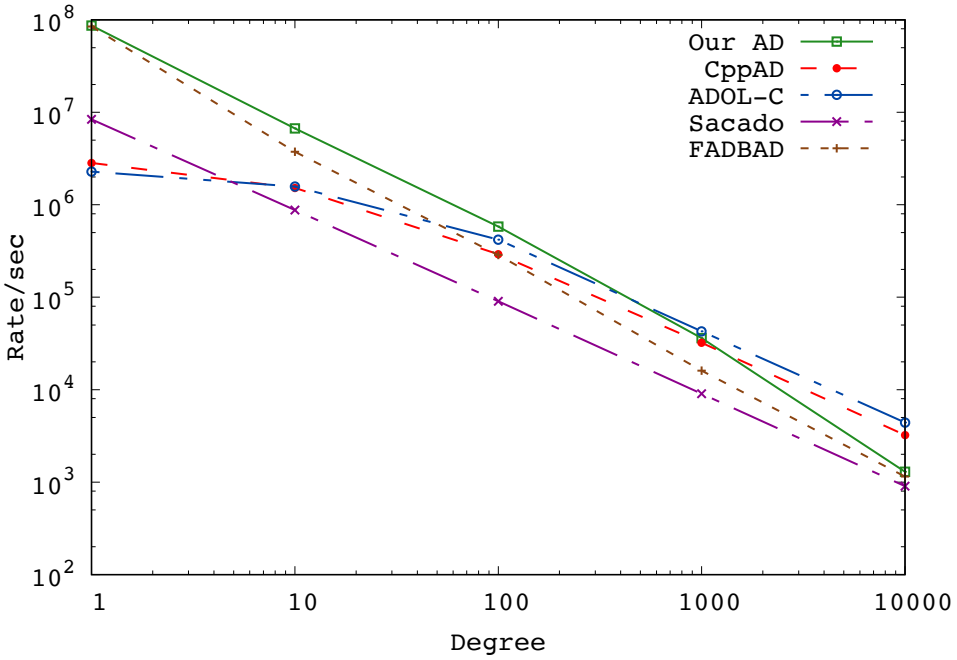
The rates are the number of times per second that the TCs of the corresponding polynomials are computed. The original `poly` benchmark from CppAD uses the nested form, while we add the second form to test the effect of CSE. Observe that every power term in a polynomial of simplified form is a subexpression of all the following power terms. Thus, if a AD tool can recognize the common subexpressions, by evaluating only the last power term, the AD also obtains TCs of all the power terms before the last power term.

Table 6.2: Comparison of setup time with other AD tools. Since CPU time less than one microsecond is not very accurate for being recorded and not of much interest, we use "−" to denote it instead of recording it.

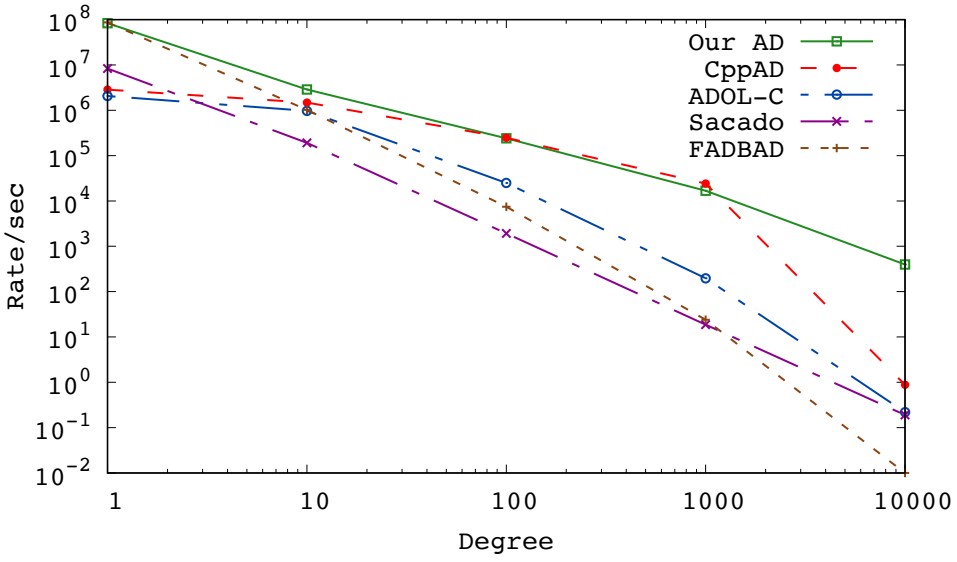
Polynomial (simplified)		Setup Time (CPU second)				
degree	#operators	Our AD	CppAD	ADOL-C	Sacado	FADBAD++
`1`	1	-	-	-	-	-
`10`	55	-	-	-	-	-
`100`	5,050	4×10^{-3}	-	-	-	-
`1,000`	500,500	0.4	5.2×10^{-2}	4×10^{-3}	-	1.6×10^{-2}
`10,000`	50,005,000	45.1	6.5	0.4	-	2.0

In these speed tests, we do the setup work only once before computing TCs. That is, at the beginning of these tests, we record tapes for CppAD and ADOL-C, or build computational graphs for our AD and FADBAD++. For Sacado, no setup work other than copying coefficients of polynomials is required. Also, in the case of CppAD, the `optimize` option is used to optimize the operation sequence before performing computations [6].

For polynomials in the nested form, the setup time for all used AD tools is negligible (within tens of microseconds); for polynomials in the simplified forms, the setup time of the used AD tools is given in Table 6.2. Since the setup work is done only once, we do not take the setup time into account when calculating the rates for all the AD tools. This is reasonable as in the practical applications, the setup work (recording tapes or building computational graphs) is needed only once before computations as long as the equations do not change, while the computations of TCs are carried out repeatedly during the whole process.



(a) Nested Polynomials



(b) Simplified Polynomials

Figure 6.1: Speed tests of computing Taylor series. (a) Evaluating $2nd$ order TCs of polynomials in the nested forms. (b) Evaluating $2nd$ order TCs of polynomials in the simplified forms.

The major portion of setup time in our AD tool comes from CSE, of which the cost grows linearly as the number of overloaded operators increases (see the time complexity in §3.2). Nevertheless, CSE manages to process around 50 million operators within one minute as shown in the last row of Table 6.2.

In Figure 6.1, we illustrate the rates for polynomials of degree 1 up to degree $10,000$. Judging from these rates, our AD has the best performance overall among all the AD tools used here in the speed test for computing Taylor series. The advantage of CSE is revealed when considering polynomials in the simplified form. As shown in Figure 6.1(b), only our AD manages to maintain the rate of computing TCs of the simplified polynomials at the same magnitudes as that of computing TCs of the nested polynomials. This is because the CSE technique in our AD eliminates all the common power terms in the simplified polynomials. The CppAD also uses the `optimize` option to optimize the operation sequence. However, as shown in the red slope between degree 1,000 and 10,000 in Figure 6.1(b), the rate decreases sheer when the operations count reaches 50 millions (at degree 10,000).

6.3 Performance in solving DAEs

We have integrated all the proposed methods into the Daets solver. In this section, we evaluate the overall performance of our AD methods in solving DAEs. In the following assessments, sparse gradient propagation (for computing System Jacobian, see §5.2) and sparse linear algebra are used for sparse DAE, while dense gradient propagation and dense linear algebra are used for dense DAE. In the Daets solver, tolerance and the Taylor series order are set to 10^{-8} and 15, respectively.

6.3.1 DAEs used in the benchmarking

Dense DAE

The Layne Watson [37] (LW for short) is an index-1 DAE of 60 equations.

sparse DAEs

The distillation column [36] (DC for short) is an index-2 DAE of 189 equations. The Spring-Mass-Pendulum [38] with 30 pendulum (SMP-30 for short) is converted to an index-3 DAE of 90 equations by differentiating the Euler Lagrange equation in the backward mode of AD.

Except for profiling in §6.3.4, we integrate LW until DAETS find the solution to the problem, and we integrate DC and SMP-30 over the interval $[0, 100]$.

6.3.2 Incremental computing

Here, we show the performance improvement gained from the incremental computing schemes alone. For this purpose, we have set an option in the DAETS solver to disable/enable the incremental computing in AD, while all other components are the same. Table 6.3 presents the results.

Table 6.3: Comparison of CPU time spent in the DAE solving with the incremental computing disabled/enabled in the DAETS solver.

DAE	CPU Time (seconds)		
	INRC disabled	INRC enabled	Time saved
LW	45.3	11.2	75.3%
DC	47.7	13.6	71.5%
SMP-30	203.4	68.7	66.2%

6.3.3 Comparison against other AD tools

FADBAD++ was the AD package used by the DAETS solver. We also have built interfaces for DAETS to use ADOL-C. Since ADOL-C does not support d^p/dt^p operator for dependent variables in the equations, we cannot use ADOL-C to solve SMP-30. Table 6.4 shows that our AD methods achieve multiple-fold speedups over these two popular AD packages in solving DAEs.

Table 6.4: Comparison of CPU time in solving DAEs using our AD methods against using FADBAD++ and ADOL-C, where "−" denotes that ADOL-C does not support solving that DAE for the reason explained above.

DAE	CPU Time (seconds)			Speedup Over (times)	
	FADBAD++	ADOL-C	Ours	FADBAD++	ADOL-C
LW	138.0	24.2	11.2	12.3	2.2
DC	135.1	93.8	13.6	9.9	6.9
SMP-30	1041.0	-	68.7	15.2	-

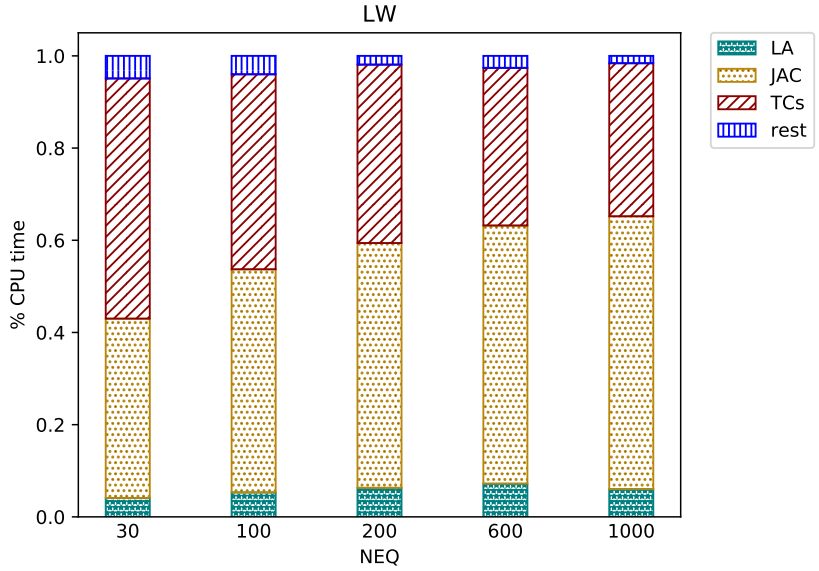
Besides the incremental computing strategy, our AD schemes have other two advantages over FADBAD++ and ADOL-C. That is, we exploit sparsity in computing System Jacobian and we use CSE to build compact computational graphs leading to fewer required operations in successive AD computations (see §6.1 and §6.2). Although ADOL-C exploits sparsity in computing general Jacobians by compression and coloring [17], it does not have the sparsity support for computing System Jacobian to date that this thesis is being written.

6.3.4 Scaling of computations: work breakdown

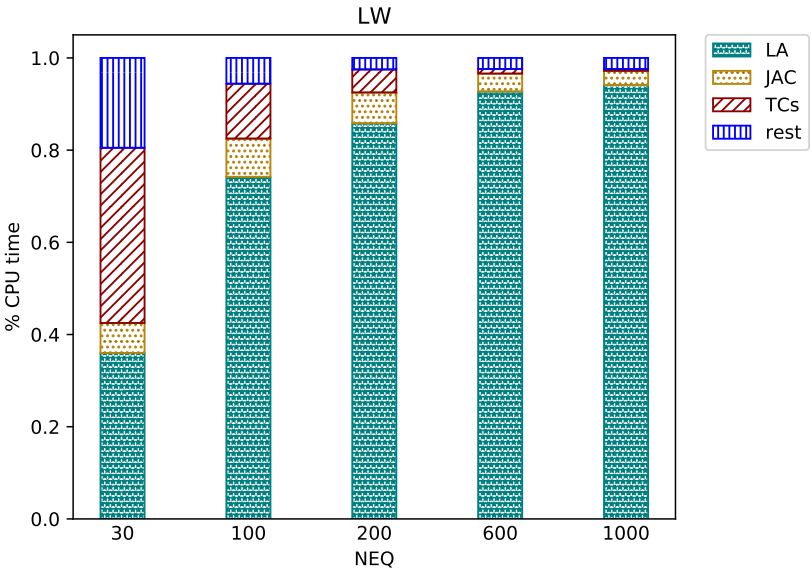
To compare the cost of AD computations and linear algebra for solving the underlying linear systems as the size of DAE increases, we use profiling results from Intel VTune Amplifier [2] to generate the work breakdown in Figure 6.2 and Figure 6.3 for LW (dense DAE) and DC (sparse DAE), respectively. We integrate each DAE of a specific size for at least one minute so the profiler gathers enough information for analysis.

It is well known that the factorization of unsymmetric matrices in dense linear algebra grows approximately $O(n^3)$, where n is the number of equations. Even so, if DAETS uses FADBAD++ to compute TCs and System Jacobian, the computing time is dominated by AD computations as shown in Figure 6.2(a). However, after incorporating our AD schemes into DAETS, dense linear algebra dominates the CPU time very soon as n increases as shown in Figure 6.2(b). This in return proves that our AD methods have much slower growth rates than that of dense algebra for solving linear systems (see time complexity bounds of our AD methods in §4.2 and §5.1).

For solving a sparse DAE like DC, we observe that almost all computing time is spent in AD, if using FADBAD++ to compute TCs and System Jacobian in DAETS as shown in Figure 6.3(a). The situation has been much improved after substituting our AD schemes for FADBAD++, especially for computing System Jacobian as shown in Figure 6.3(b). We note that computing TCs now takes the major portion of the solution time. This is because both linear algebra and computing System Jacobian (see §5.2) can take advantage of sparsity to gain efficiency while computing TCs cannot. Nevertheless, our AD methods have comparable growth rates as that of sparse linear algebra for solving linear systems, as evidenced by the similar CPU time breakdowns as the size of DAE increases in Figure 6.3(b).

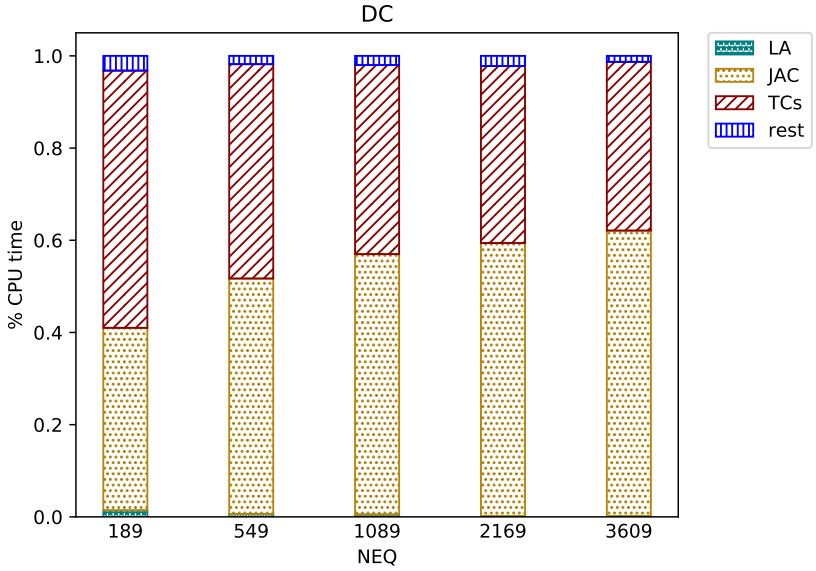


(a) Scaling of DAETS with FADBAD++

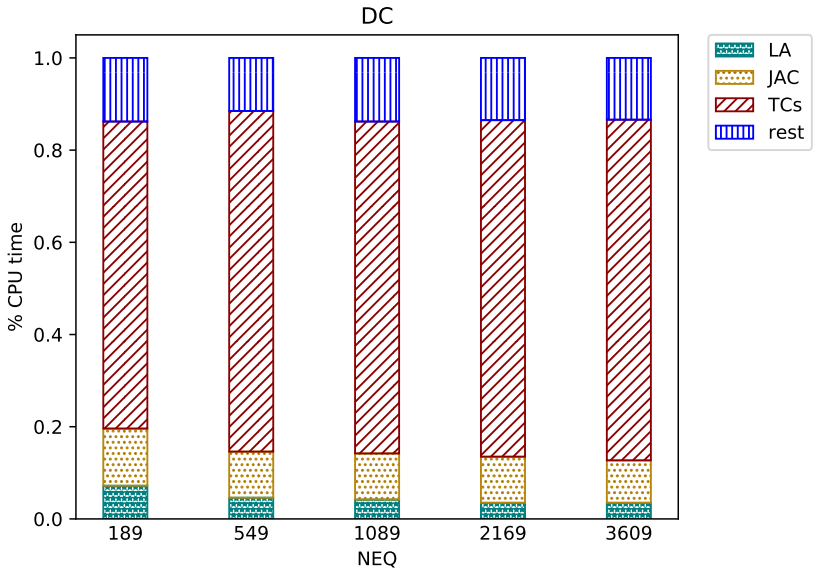


(b) Scaling of DAETS with our AD schemes

Figure 6.2: Proportions of computations by DAETS solver (with dense linear algebra) in solving a dense DAE, LW, as its size grows. The fractions of CPU time are denoted by `LA` for linear algebra, `JAC` for computing System Jacobian, `TCs` for computing Taylor series coefficients, and `rest` for all other computations involved.



(a) Scaling of DAETS with FADBAD++



(b) Scaling of DAETS with our AD schemes

Figure 6.3: Proportions of computations by DAETS solver (with sparse linear algebra) in solving a sparse DAE, DC, as its size grows. The fractions of CPU time are denoted by `LA` for linear algebra, `JAC` for computing System Jacobian, `TCs` for computing Taylor series coefficients, and `rest` for all other computations involved.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have developed a common subexpression elimination (CSE) method in §3 to build a compact computational graph through the operator overloading approach. CSE is of linear complexity both in time and space. Given a DAE function described by a computer program, CSE eliminates repeated subexpressions behind the scenes, leading to fewer required operations in successive AD computations.

In the context of solving DAE by Taylor series, we have derived two efficient AD schemes to compute incrementally Taylor series in §4 and System Jacobian in §5, respectively. For computing System Jacobian, we have also exploited the sparsity structure of equations to devise a sparse gradient propagation method in the forward mode of AD.

We have implemented the proposed methods in the DAETS solver. Computational results in §6 have shown the effectiveness of our methods and substantial performance improvement in solving sparse and dense DAEs.

7.2 Future work

As shown in the last section of §6, the computation of Taylor series is dominating the solving time when the DAE is sparse. This is because no sparsity is known for computing Taylor series, as opposed to linear algebra and gradient propagations. This observation suggests a possible direction is to improve the performance of computing Taylor series by exploiting parallelism or combining other strategy, for example, the coefficient doubling in [21].

The methods developed in this thesis benefit from careful study of the underlying computational graph of given equations. By the time this thesis is written, the author is developing an interactive web-based computational graph viewer. It reads the output from the Daets solver and visualizes graph information on a website canvas. We hope to gain insights from the visualization for systems at scale in order to further improve our AD schemes in the future.

Appendix A

Proof of Lemma 5.1.1

Let the notation be as at the start of §5.1.

Proof. We show the proof for two items of Lemma 5.1.1 as follows.

(i) The jth components of $\nabla_k \left({f_i}^{(k+c_i)} \right)$ and $\nabla_{k^*} \left({f_i}^{(k^*+c_i)} \right)$ are $\partial f_i^{(k+c_i)}/\partial x_j^{(k+d_j)}$ and $\partial f_i^{(k^*+c_i)}/\partial x_j^{(k^*+d_j)}$, respectively. Since $\partial f_i^{(k+c_i)}/\partial x_j^{(k+d_j)} = \partial f_i/\partial x_j^{(d_j-c_i)} = \partial f_i^{(k^*+c_i)}/\partial x_j^{(k^*+d_j)}$ by (2.6) for $j = 1:n$, where d_j and c_i are constants throughout all stages, we have $\nabla_k \left({f_i}^{(k+c_i)} \right) = \nabla_{k^*} \left({f_i}^{(k^*+c_i)} \right)$ for any stage $k > k^*$.

(ii) Recall from §2.1 that the jth component in the ith row of System Jacobian $\mathbf{J}$ is defined as $\mathbf{J}_{ij} = \partial f_i/\partial x^{(d_j-c_i)}$, if this derivative is present in f_i and 0 otherwise. Thus, for $d_j - c_i > 0$, we have $\mathbf{J}_{ij} = \partial f_i/\partial x_j^{(d_j-c_i)} = \partial f_i^{(k+c_i)}/\partial x_j^{(k+d_j)}$ by (2.6), and $\partial f_i^{(k+c_i)}/\partial x_j^{(k+d_j)}$ is also the jth component of $\nabla_k \left({f_i}^{(k+c_i)} \right)$, where $k+c_i \geq 0$, by (5.1); For $d_j - c_i \leq 0$, both $\mathbf{J}_{ij}$ and the jth component of $\nabla_k \left({f_i}^{(k+c_i)} \right)$ are zeros. Hence, we conclude that $\nabla_k({f_i}^{(k+c_i)})$ is the ith row of $\mathbf{J}$ at stages where $k + c_i \geq 0$. Further, since the ith row of $\mathbf{J}_k$ consists of components in the ith

row of $\mathbf{J}$ where $k + d_j \geq 0$, these components are the same as the components in $\nabla_k \left({f_i}^{(k+c_i)} \right)$ where $k + d_j \geq 0$.

□

Bibliography

[1] The Community Portal for Automatic Differentiation: autodiff web page. http://www.autodiff.org

[2] Intel VTune Amplifier: VTune 2017 web page. https://software.intel.com/en-us/intel-vtune-amplifier-xe

[3] Averick, B., Carter, R., Moré, J., Xue, G.: The MINPACK-2 test problem collection. Mathematics and Computer Science Division, Argonne National Laboratory. Preprint MCS-P153-0692 (1992)

[4] Bartholomew-Biggs, M.C., Bartholomew-Biggs, L., Christianson, B.: Optimization & automatic differentiation in Ada: some practical experience. Optimization Methods and Software **4**(1), 47–73 (1994)

[5] Bauer, F.L.: Computational graphs and rounding error. SIAM Journal on Numerical Analysis **11**(1), 87–96 (1974)

[6] Bell, B.M.: CppAD: a package for C++ algorithmic differentiation. Computational Infrastructure for Operations Research (2012)

[7] Bendsten, C., Stauning, O.: FADBAD++, version 2.1 (2007) http://www.fadbad.com

[8] Bendsten, C., Stauning, O.: TADIFF, a flexible C++ package for automatic differentiation using Taylor series. Tech. Rep. 1997-x5-94, Department of Mathematical Modelling, Technical University of Denmark, DK-2800, Lyngby, Denmark (1997)

[9] Bischof, C.H., Khademi, P.M., Buaricha, A., Alan, C.: Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. Optimization Methods and Software **7**(1), 1–39 (1996)

[10] Brent, R.P., Kung, H.T.: Fast algorithms for manipulating formal power series. Journal of the ACM (JACM) **25**(4), 581–595 (1978)

[11] Chang, Y., Corliss, G.: ATOMFT: solving ODEs and DAEs using Taylor series. Computers & Mathematics with Applications **28**(10-12), 209–233 (1994)

[12] Curtis, A., Powell, M.J., Reid, J.K.: On the estimation of sparse Jacobian matrices. IMA Journal of Applied Mathematics **13**(1), 117–119 (1974)

[13] Davis, T.A., et al.: SuiteSparse: a suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html

[14] Demmel, J.W.: Applied Numerical Linear Algebra. SIAM (1997)

[15] Diehl, M., Walther, A.: A test problem for periodic optimal control algorithms. Tech. rep., MATH-WR-01-2006, TU Dresden (2006)

[16] Dixon, L.C., Maany, Z., Mohseninia, M.: Automatic differentiation of large sparse systems. Journal of Economic Dynamics and Control **14**(2), 299–311 (1990)

[17] Gebremedhin, A.H., Pothen, A., Walther, A.: Exploiting sparsity in Jacobian computation via coloring and automatic differentiation: a case study in a simulated moving bed process. In: Advances in Automatic Differentiation, pp. 327–338. Springer (2008)

[18] Giering, R., Kaminski, T.: Automatic sparsity detection implemented as a source-to-source transformation. In: International Conference on Computational Science, pp. 591–598. Springer (2006)

[19] Griewank, A., Juedes, D., Utke, J.: Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. ACM Transactions on Mathematical Software (TOMS) **22**(2), 131–167 (1996)

[20] Griewank, A., Walther, A.: On the efficient generation of Taylor expansions for DAE solutions by automatic differentiation **3732**, 1103–1111 (2006)

[21] Griewank, A., Walther, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation–Second Edition. SIAM, Philadelphia, PA, USA (2008)

[22] Lam, M., Sethi, R., Ullman, J.D., Aho, A.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (2006)

[23] LAPACK Project: LAPACK — Linear Algebra PACKage. http://www.netlib.org/lapack/

[24] Liu, T., Bargteil, A.W., O'Brien, J.F., Kavan, L.: Fast simulation of mass-spring systems. ACM Transactions on Graphics (TOG) **32**(6), 214 (2013)

[25] Nedialkov, N., Pryce, J.: DAETS — Differential-Algebraic Equations by Taylor Series. http://www.cas.mcmaster.ca/~nedialk/daets

[26] Nedialkov, N.S., Pryce, J.D.: Solving differential-algebraic equations by Taylor series (I): Computing Taylor coefficients. BIT Numerical Mathematics **45**(3), 561–591 (2005)

[27] Nedialkov, N.S., Pryce, J.D.: Solving differential-algebraic equations by Taylor series (II): Computing the System Jacobian. BIT Numerical Mathematics **47**(1), 121–135 (2007)

[28] Nedialkov, N.S., Pryce, J.D.: Solving differential-algebraic equations by Taylor series (III): the DAETS code. JNAIAM J. Numer. Anal. Indust. Appl. Math **3**, 61–80 (2008)

[29] Newsam, G.N., Ramsdell, J.D.: Estimation of sparse Jacobian matrices. SIAM Journal on Algebraic Discrete Methods **4**(3), 404–418 (1983)

[30] Pankiewicz, W.: Algorithms: Algorithm 337: calculation of a polynomial and its derivative values by horner scheme. Communications of the ACM **11**(9), 633 (1968)

[31] Phipps, E., Pawlowski, R.: Efficient expression templates for operator overloading-based automatic differentiation. In: Recent Advances in Algorithmic Differentiation, pp. 309–319. Springer (2012)

[32] Pryce, J.D.: A simple structural analysis method for DAEs. BIT Numerical Mathematics **41**(2), 364–394 (2001)

[33] Pryce, J.D., Nedialkov, N.S., Tan, G., Li, X.: How AD can help solve differential-algebraic equations. arXiv preprint arXiv:1703.08914 (2017)

[34] Sedgewick, R.: Algorithms in C++, Parts 1-4: Fundamentals, Data Structures, Sorting, Searching–Third Edition. Addison Wesley Longman (1998)

[35] Sedgewick, R.: Algorithms in C++, Part 5: Graph Algorithms–Third Edition. Addison Wesley Longman (2002)

[36] Washington, I., Swartz, C.: On the numerical robustness of differential-algebraic distillation models. In: 61st Canadian Chemical Engineering Conference. London, Ontario, Canada (2011)

[37] Watson, L.T.: A globally convergent algorithm for computing fixed points of C^2 maps. Appl. Math. Comput. **5**, 297–311 (1979)

[38] Zhu, Y., Westbrook, E., Inoue, J., Chapoutot, A., Salama, C., Peralta, M., Martin, T., Taha, W., O'Malley, M., Cartwright, R., et al.: Mathematical equations as executable models of mechanical systems. In: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems, pp. 1–11. ACM (2010)