IMPLEMENTATION OF FORWARD AND REVERSE MODE AUTOMATIC DIFFERENTIATION FOR GNU OCTAVE APPLICATIONS

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

Yixiu Kang

March 2003

This thesis entitled

IMPLEMENTATION OF FORWARD AND REVERSE MODE AUTOMATIC DIFFERENTIATION FOR GNU OCTAVE APPLICATIONS

BY

YIXIU KANG

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

David W. Juedes

Associate Professor of Electrical Engineering and Computer Science

Dennis Irwin

Dean, Russ College of Engineering and Technology

KANG, YIXIU M.S. March 2003. EECS

Implementation of Forward and Reverse Mode Automatic Differentiation for GNU Octave Applications (71pp.)

Director of Thesis: David W. Juedes

In this work, we present two C/C++ implementations of general purpose automatic differentiation (AD) for GNU Octave applications: FAD for forward mode AD and LogAD for reverse mode AD with bisection checkpointing. Both FAD and LogAD accept functions written in the GNU Octave language and work in the Octave environment via dynamically linked functions. FAD evaluates the product of the Jacobian of the input function and an arbitrary vector in time and space that are proportional to the time and space used by the original function. LogAD evaluates the product of an arbitrary vector and the Jacobian of the input function via a checkpointing approach first proposed by Griewank in 1992.

Approved: David W. Juedes

Associate Professor of EECS

Acknowledgments

I want to acknowledge those whose help and support made this thesis possible. My sincere thanks go to my thesis advisor Dr. David W. Juedes for his guidance and helps. My thanks also go to Dr. Wen-Jia R. Chen for being the college representative on my thesis committee, and Dr. Mehmet Celenk and Dr. Cynthia R. Marling for being on my thesis committee.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The mathematical description of any system with continuous changes in one or more of its properties involves either differentiation or integration. The differentiation of a function is a fundamental problem in scientific and engineering world. Modern computer made it possible to lend the complex computation work to computer with high automation, efficiency, and precision. The application of the chain rule to obtain derivatives performed automatically and accurately by computers, automatic differentiation (AD), started to play a significant role in solving scientific and engineering problems, such as weather forecasting [6, 10], fluid dynamics [11, 7, 23], beam physics [3], and neural networks [29, 18, 4], during the last few decades. In this work, we present two C/C++ implementations of automatic differentiation (AD), FAD for forward mode AD and LogAD for reverse mode AD with bisection checkpointing, for GNU Octave applications.

1.1 Introduction

The forward mode AD was introduced around late 1950s and early 1960s by Beta Korolev, Sukkikh, and Frolova [2], and Wengert [33], with further investigations by Wilkins [35], Kedem [20], and Rall [26]. The forward mode AD involves the decomposition of a given function written in a programing language, such as C or FORTRAN, into a series of elementary functional steps, and automatic evaluation and differentiation of the elementary functions through the same forward sequence.

The efficient evaluation of all partial derivatives of a single dependent variable with respect to many independent variables, reverse mode AD, was independently discovered by Werbos [34], Linnainmaa [22], Speelpenning [30], and Baur and Strassen [1]. The checkpointing of the computation to improve space complexity was discussed by Volin and Ostrovskii [32] and Griewank [13].

The general purpose AD tools usually implement one or more preprocessors to translate the codes of a function in a programming language into codes in the same or another language to differentiate the function automatically. The first powerful general purpose AD tool might be GRESS [24] developed at Oak Ridge National Laboratory in the 1980s. Some early AD software implementations were listed and classified by Juedes in [17]. More software have been developed since then, just to name a few, which include ADIFOR [5], Odyssée [28], ADOL-C [15], PADRE2

[21] and TAMC [9].

So far, we know of two AD tools can be called from Matlab. The first one is GRAD, which is a C language implementation of forward mode AD in Matlab reported by Rich and Hill in 1992 [27], and the second one is ADMIT, which was reported by Coleman and Verma in 2000 [8]. ADMIT is a Matlab interface built on top of ADOL-C, which also includes the graph-coloring algorithms to compute sparse Jacobian and Hessian. The implementation of ADOL-C involves operating system interventions through Unix forks and pipes, which is what we are avoiding in this work.

More detailed discussions can be found in books [25, 19] for forward AD, in paper [12] for reverse AD, and in book [14] for both.

1.2 A little bit history of FAD and LogAD

FAD and LogAD are C/C++ AD software tools developed in Prof. David W. Juedes's group at Ohio University. The first version of LogAD was developed by Richard Heller [16], which is a compiler that accepts functions in a subset of C language, and generate C code that implements reverse mode AD with Griewank's bisection checkpointing algorithm [13] without using operating system intervention. The second version of LogAD, LogAD v.2, was developed by Alexei Stovboun [31]. Unlike the former version, LogAD v.2 accepts functions written in GNU Oc-

tave language, which is compatible to the language used by Matlab, and works in the Octave environment by calling dynamically linked function gdiff to evaluate the gradient of scalar functions. The primary contribution of this work is the development of LogAD v.2.1, which implemented the general computation of vector-Jacobian products in the reverse mode of AD, and FAD v.1, which implemented general purpose computation of Jacobian-vector products in the forward mode of AD. LogAD v.2.1 is the improvement and further development of LogAD v.2 to a working version with more functionalities. To make it a more complete AD tool, we also implemented FAD v.1 for forward mode AD in this thesis. FAD and LogAD could be an additional tool in Matlab/Octave's toolkit for engineers to solve engineering problems.

1.3 Organization of this thesis

The thesis is organized as follows: in Chapter 2, mathematical fundamentals of AD will be briefly introduced; in Chapter 3, the implementation details of the general forward mode AD, FAD, will be given; in Chapter 4, the implementation details of general reverse mode AD, LogAD, will be given; in Chapter 5, some examples are provided for both LogAD and FAD to test the software performance; in Chapter 6, we summarize the test results for FAD and LogAD.

Chapter 2

Automatic differentiation

Automatic differentiation is a chain rule based technique for evaluating the derivatives analytically without any truncation errors with respect to input variables of functions defined by a high-level language computer program. The automatic differentiation algorithms fall into two major categories, forward mode and reverse mode. In this section, we will review the fundamentals of the forward and reverse automatic differentiation, following the concepts in [14].

2.1 Fundamentals of forward and reverse AD

Consider a differentiable function:

$$F : R^k \rightarrow R^d \tag{2.1}$$

with independent variables $(x_1, x_2, \cdots, x_k)$ and dependent variables $(y_1, y_2, \cdots, y_d)$:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{pmatrix} = \begin{pmatrix} f_1(x_1, x_2, \cdots, x_k) \\ f_2(x_1, x_2, \cdots, x_k) \\ \vdots \\ f_d(x_1, x_2, \cdots, x_k) \end{pmatrix} \tag{2.2}$$

The Jacobian of F

$$F' = \left(\frac{\partial y_i}{\partial x_j}\right)_{d \times k} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_k} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_k} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y_d}{\partial x_1} & \frac{\partial y_d}{\partial x_2} & \cdots & \frac{\partial y_d}{\partial x_k} \end{pmatrix} \tag{2.3}$$

is a $d \times k$ matrix of first-order partial derivatives.

A program evaluating the function F can be viewed as a sequence of continuously differentiable elemental functions $v_i = \varphi_i(v_j)_{j \prec i}$ with $j < i$, where v is a set of ordered variables such that v_j is computed before v_i. The ordered variables can be partitioned into three vectors evaluated sequentially as listed in Table 2.1.

v_{i-k}	$=$	x_i	$\} \; i = 1 \cdots k$	(independent)
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	$\} \; i = 1 \cdots l$	(intermediate)
y_{d-i}	$=$	v_{l-i}	$\} \; i = d-1 \cdots 0$	(dependent)

Table 2.1: General evaluation procedure for a function.

In this section we calculate first derivative vectors, called tangents and gradients, in the forward and reverse modes of AD, respectively. In general, the numerical values of derivative vectors obtained by AD depend on certain seed directions and weight functions rather than representing derivatives of particular dependent variables y_i with respect to particular independent variables x_j which we call Cartesian derivatives. In other words we obtain directional derivatives of scalar functions that are defined as weighted averages of vector function F.

The forward propagation of tangents can be viewed as differentiating each single elemental functions in the same order as the functions are evaluated. Let $\dot{x}$ be the selected direction vector, according to the chain rule we have

$$\dot{y} = F'(x)\dot{x} \tag{2.4}$$

The tangent evaluation procedure Table 2.2 can be obtained by simply differentiation Table 2.1, where $u_i = (v_j)_{j \prec i}$ is the argument vector on which φ_i depends. Let the time to evaluate the original function F be T, the matrix-vector product

v_{i-k}	$\equiv$	x_i	$\left.\begin{array}{l}\\\\\end{array}\right\} 1 \le i \le k$
$\dot{v}_{i-k}$	$\equiv$	$\dot{x}_i$	
v_i	$\equiv$	$\varphi_i(v_j)_{j \prec i}$	$\left.\begin{array}{l}\\\\\end{array}\right\} 1 \le i \le l$
$\dot{v}_i$	$\equiv$	$\sum_{j \prec i} \frac{\partial}{\partial v_j}\varphi_i(u_i)\dot{v}_j$	
y_{d-i}	$\equiv$	v_{l-i}	$\left.\begin{array}{l}\\\\\end{array}\right\} d > i \ge 0$
$\dot{y}_{d-i}$	$\equiv$	$\dot{v}_{l-i}$	

Table 2.2: Tangent recursion for general evaluation procedure.

$\dot{y} = F'(x)\dot{x}$ calculated by employing the evaluation procedure Table 2.2 can clearly be done in time proportional to T. One may want to calculate p tangents simultaneously in several directions $\dot{x}$ from the same base point x, the cost of evaluating derivatives by propagating them forward through the evaluation procedure will increase linearly with p.

The reverse propagation of gradients (or adjoints) evaluates the vector-matrix product

$$\bar{x} \equiv \bar{y}F'(x) \tag{2.5}$$

where $\bar{y}$ is a fixed row vector. The reverse mode of AD is in some sense a backward application of the chain rule. The evaluation of the vector-matrix product can be rewritten as the adjoint evaluation procedure listed in Table 2.3. As is apparent in

$\bar{v}_i$	$\equiv$	0	$1-k \le i \le l$
v_{i-k}	$\equiv$	x_i	$1 \le i \le k$
v_i	$\equiv$	$\varphi_i(v_j)_{j \prec i}$	$1 \le i \le l$
y_{d-i}	$\equiv$	v_{l-i}	$d > i \ge 0$
$\bar{v}_{l-i}$	$\equiv$	$\bar{y}_{d-i}$	$0 \le i < d$
$\bar{v}_j$	$+\equiv$	$\bar{v}_i \frac{\partial}{\partial v_j}\varphi_i(u_i)$ for $j \prec i$	$l \ge i \ge 1$
$\bar{x}_i$	$\equiv$	$\bar{v}_{i-k}$	$k \ge i \ge 1$

Table 2.3: Incremental adjoint recursion evaluation procedure.

Table 2.3, the calculation of the adjoint vector $\bar{x} = \bar{y}F'(x)$ proceeds in two stages: a forward sweep up to the single line and a return sweep below it. Throughout the thesis we will refer to the combination of a forward sweep and a subsequent return

sweep as a reverse sweep. The cost for evaluating the adjoint is proportional to the cost to evaluate the function. When q gradients is evaluated simultaneously, the total cost will increase linearly with q. Therefore, AD not only can benefit from a small number of independent variables by performing forward propagation, but also can benefit from a small number of dependent variables by performing reverse propagation.

In reverse mode AD, in order to perform the return sweep, the forward sweep must be recorded. Therefore, the memory needed to record the computation is proportional to the run time of the program, which could be prohibitively expensive. A partial solution to the memory problem of the reverse mode AD was proposed by Griewank [13], which achieves logarithmic growth of temporal and spatial complexity in reverse AD by checkpointing the computation at appropriate points.

2.2 A simple example for forward and reverse AD

Let us consider the function $F : R^2 \to R^2$ with independent variables $x = (x_1, x_2)$ and dependent variables

$$y_1 = x_1 \cos(x_2) \quad \text{and} \quad y_2 = x_1 \sin(x_2) \tag{2.6}$$

The Jacobian of the function

$$F' = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{pmatrix} = \begin{pmatrix} cos(x_2) & -x_1 \sin(x_2) \\ sin(x_2) & x_1 \cos(x_2) \end{pmatrix} \tag{2.7}$$

Let the direction vector be $\dot{x} = \begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix}$, then

$$\dot{y} = F'(x)\dot{x} = \begin{pmatrix} \dot{x}_1 \cos(x_2) - \dot{x}_2 x_1 \sin(x_2) \\ \dot{x}_1 \sin(x_2) + \dot{x}_2 x_1 \cos(x_2) \end{pmatrix} \tag{2.8}$$

The procedure for the forward propagation of tangents is listed in Table 2.4. Let

v_{-1}	=	x_1;	$\dot{v}_{-1}$	=	$\dot{x}_1$
v_0	=	x_2;	$\dot{v}_0$	=	$\dot{x}_2$
v_1	=	$\cos(v_0)$;	$\dot{v}_1$	=	$-\sin(v_o)\dot{v}_0$
v_2	=	$\sin(v_0)$;	$\dot{v}_2$	=	$\cos(v_o)\dot{v}_0$
v_3	=	$v_{-1} * v_1$;	$\dot{v}_3$	=	$\dot{v}_{-1} * v_1 + v_{-1} * \dot{v}_1$
v_4	=	$v_{-1} * v_2$;	$\dot{v}_4$	=	$\dot{v}_{-1} * v_2 + v_{-1} * \dot{v}_2$
y_1	=	v_3;	$\dot{y}_1$	=	$\dot{v}_3$
y_2	=	v_4;	$\dot{y}_2$	=	$\dot{v}_4$

Table 2.4: Forward propagation of tangents of the function.

$x_1 = 5$, $x_2 = \pi/4$, $\dot{x}_1 = 1$, and $\dot{x}_2 = 1$, by following the evaluation procedure Table 2.4 we will obtain that $y_1 = 3.53553$, $\dot{y}_1 = -2.82842$, $y_2 = 3.53553$, and $\dot{y}_2 = 4.24264$.

Now let the adjoint vector be $\bar{y} = (\bar{y_1} \quad \bar{y_2})$, then

$$\bar{x} = \bar{y}F'(x) = \begin{pmatrix} \bar{y_1}\cos(x_2) + \bar{y_2}\sin(x_2) & -\bar{y_1}x_1\sin(x_2) + \bar{y_2}x_1\cos(x_2) \end{pmatrix} \tag{2.9}$$

The reverse propagation of the adjoints is listed in Table 2.5, with $\bar{v_i} = 0$ for $i =$

v_{-1}	$=$	x_1
v_0	$=$	x_2
v_1	$=$	$\cos(v_0)$
v_2	$=$	$\sin(v_0)$
v_3	$=$	$v_{-1} * v_1$
v_4	$=$	$v_{-1} * v_2$
y_1	$=$	v_3
y_2	$=$	v_4
$\bar{v_4}$	$=$	$\bar{y_2}$
$\bar{v_3}$	$=$	$\bar{y_1}$
$\bar{v_{-1}}$	$+=$	$\bar{v_4} * v_2$
$\bar{v_2}$	$+=$	$\bar{v_4} * v_{-1}$
$\bar{v_{-1}}$	$+=$	$\bar{v_3} * v_1$
$\bar{v_1}$	$+=$	$\bar{v_3} * v_{-1}$
$\bar{v_0}$	$+=$	$\bar{v_2} * \cos(v_0)$
$\bar{v_0}$	$+=$	$-\bar{v_1} * \sin(v_0)$
$\bar{x_2}$	$=$	$\bar{v_0}$
$\bar{x_1}$	$=$	$\bar{v_{-1}}$

Table 2.5: Reverse propagation of gradients of the function.

$-1, \cdots, 4$ being initialized before the evaluation. Let $x_1 = 5$, $x_2 = \pi/4$, $\bar{y_1} = 1$, and $\bar{y_2} = 1$, by following the evaluation procedure Table 2.5 we will obtain that $y_1 = 3.53553$, $\bar{x_1} = 1.41421$, $y_2 = 3.53553$, and $\bar{x_2} = 0.00000$.

2.3 Proposed work of this thesis

In this thesis, we implement the forward mode AD (FAD v.1) which evaluate the product of Jacobian and an arbitrary vector with a cost proportional to the cost of evaluating the original function, and the reverse mode AD (LogAD v.2.1) which evaluate the product of an arbitrary vector and Jacobian with bi-section checkpointing algorithm [13].

FAD v.1 implements general forward mode AD procedure listed in Table 2.2, which evaluate the value of a function, either a scalar or a matrix, and the product of the Jacobian of the function and an arbitrary vector. The temporal and spatial complexity of the algorithm are proportional to that of the evaluating the value of the function only.

LogAD v.2.1 implements general reverse mode AD procedure listed in Table 2.3 with bisection checkpointing algorithms by revising and expanding the function of LogAD v.2 [31], which developed to evaluate partial derivatives for scalar functions only, to evaluate the product of an arbitrary vector and the Jacobian of a function, which can be either a scalar or a matrix function. The checkpointing algorithm implemented the bi-section checkpointing algorithm proposed by Griewank [13] so that the large size computations can be performed when space is limited.

In this work, the following differentiable elementary functions are implemented for both forward and reverse mode automatic differentiation:

- $v = c$;
- $v = -u$;
- $v = u \pm w$;
- $v = u * w$;
- $v = u/w$;
- $v = \sin(u),\ v = \cos(u)$;
- $v = \log(u),\ v = \log 10(u)$;
- $v = exp(u)$;
- $v = u^w$.

Where u, v, and w are variables (scalar or matrix), and c is a constant.

Both FAD and LogAD accept algorithms written in a subset of Octave/Matlab language, and can be called in Octave environment by calling dynamically linked functions fdiff and gdiff.

Chapter 3

General forward mode automatic differentiation

3.1 Algorithm

The general forward mode automatic differentiation in this work implements the algorithm listed in Table 2.3, which evaluates the input function and the product of its Jacobian and an arbitrary vector. The spatial and temporal complexity of the general forward mode automatic differentiation is proportional to the spatial and temporal complexity of the original algorithm for evaluating the input function only.

3.2 Implementation

The input functions to FAD v.1 are functions in GNU Octave language, which is compatible with the language used in Matlab by MathWorks, Inc.

FAD v.1 implemented the following functions:

1. FAD compiler, fadc, which decomposes a function written in Octave language into elemental operations, and translates the code into a fragment of C++ code. It implements two Octave data types, real scalar and real matrix, and accepts a subset of Octave language constructs:

   - expressions and index expressions;
   - *if* - *elseif* - *else* statements;
   - *for* loops;
   - *while* loops;
   - function calls.

2. FAD linker, fadl, which generates a special shared library, <fname>.fso, that contains the function to be differentiated.

3. FAD shared library, fad.so, which provides the execution environment for the compiled Octave function.

The FAD v.1 is integrated into the octave environment and can be invoked by calling dynamically linked function fdiff. Prior to call fdiff for function <fname>,

the Octave function need to be compiled and linked by FAD compiler and FAD linker. There are two FAD run time parameters need to be set through Octave variables before calling fdiff:

- FAD_STACK_SIZE (in slots): the size of FAD execution stack;
- FAD_ARG_BUF_SIZE (in slots): the size of FAD argument buffer.

The prototype of the fdiff function is:

```
fdiff ("<fname>", <a>, <v>, <x1>, <x2>, ...)
```

where the first argument is the name of the function to be differentiated, the second argument is the active status, the third argument is the direction vector for the independent variables, and the rest arguments are the independent variables. If the active status evaluates to be TRUE, both the function and the tangent will be evaluated, otherwise only the function will be evaluated. When the active status is TRUE, the return values of fdiff are the values of the input function, either scalar or matrix, and the product of the Jacobian of the function and the direction vector; when the active status is FALSE, the return value is the value of the function only.

3.3 Two simple examples

In this section we will examine two simple examples for the forward mode automatic differentiation.

3.3.1 Example 1

In this section, a simple example, which evaluates the example in Chapter 2, will be given. Figure 3.1 lists the Octave program for the problem. Let $x =$

```
# simple_example1.m
function [y1, y2] = simple_example1 (x)
  y1 = x(1) * cos(x(2));
  y2 = x(1) * sin(x(2));
endfunction
```

Figure 3.1: File simple_example1.m in Octave language.

$(5.00000\ 0.78540)$. The computation can be done by running the dynamically linked function fdiff in Octave environment:

```
[a, b, c, d] = fdiff("simple_example1", 1, [1, 1], x)
```

The output of this computation is listed in figure 3.2.

```
a = 3.5355

b = 3.5355

c = -2.8284

d = 4.2426
```

Figure 3.2: Results of the computation for example 1.

3.3.2 Example 2

In this section, a simple example, which evaluates the product of matrices, will be given. Figure 3.3 lists the Octave program for the problem. Let $x = ones(2)$,

```
# simple_example2
function result = simple_example2 (matrix, n)
  result = 1;
  for i = 1:n
    result = result * matrix;
  end
end
```

Figure 3.3: File simple_example2.m in Octave language.

which is a 2 *by* 2 matrix filled with 1*s*. The computation can be done by running the dynamically linked function fdiff in Octave environment:

```
[a, b] = fdiff("simple_example2", 1, x, x, 4)
```

The output of this computation is shown in Figure 3.4.

```
a =

  8  8
  8  8

b =

  32  32
  32  32
```

Figure 3.4: Results of the computation for example 2.

Chapter 4

General reverse mode automatic differentiation

4.1 Algorithm

The general reverse mode AD in this work implements the algorithm listed in Table 2.2. In general, the space complexity of the general reverse mode AD is proportional to the time complexity of the original algorithm. To break this dependency so that LogAD can be used for large scale applications, LogAD also implemented Griewank's algorithm [13], reverse mode with bisection checkpointing. If implemented efficiently, reverse AD with bisection checkpointing algorithm can compute the gradient of the function with temporal and spatial complexity that are proportional to the logarithm of the time complexity of the original problem. Assume the

time and space costs of the original algorithm are T and S respectively, then the temporal complexity and spatial complexity for reverse mode AD with bisection checkpointing will be $O(T \log_2 T)$ and $O(S \log_2 T)$ respectively. In this work, for the purpose of a simpler implementation, some functions, such as index assignment for elements of a matrix, do not achieve the logarithmic temporal and spatial time complexity.

4.2 Implementation

LogAD v.2.1 is a revising and expanding of LogAD v.2, which is the work of Stovboun's master's degree thesis [31].

4.2.1 A brief review of LogAD v.2

In LogAD v.2, the reverse mode with bisection checkpointing algorithm was implemented to calculate the value and the gradient of a single input scalar function. The input functions to LogAD v.2 are functions in GNU Octave language, which is compatible with the language used in Matlab by MathWorks, Inc.

The LogAD v.2 was intended to implement the following functions:

1. LogAD compiler, logadc, which decomposes a function written in Octave language into elemental operations, inserts checkpointing algorithms, and translates the code into a fragment of C++ code. It implements two Oc-

tave data types, real scalar and real matrix, and accepts a subset of Octave language constructs:

- expressions and index expressions;
- *if* - *elseif* - *else* statements;
- *for* loops;
- *while* loops;
- function calls.

2. LogAD linker, logadl, which generates a special shared library, <fname>.lso, that contains the function to be differentiated.

3. LogAD shared library, logad.so, which provides the execution environment for the compiled Octave function.

The LogAD v.2 is integrated into the octave environment and can be invoked by calling dynamically linked function gdiff. Prior to call gdiff for function <fname>, the Octave function need to be compiled and linked by LogAD compiler and LogAD linker. There are three LogAD run time parameters need to be set through Octave variables before calling gdiff:

- LOGAD_TAPE_SIZE (in bytes): LogAD tape size for recording the computation;
- LOGAD_STACK_SIZE (in slots): the size of LogAD execution stack;

- LOGAD_ARG_BUF_SIZE (in slots): size of LogAD argument buffer.

The prototype of the gdiff function is:

```
gdiff ("<fname>", <x1>, <a1>, <x2>, <a2>,...)
```

where the first argument is the name of the function to be differentiated, and the next pairs of the arguments are the independent variables and their active states. If the active status of an independent variable is evaluated to be TRUE, the partial derivative with respect to the variable will be calculated, otherwise it will not be calculated. The function returns the value and the gradient of the function with respect to the active independent variables.

4.2.2 LogAD v.2.1

In this work, LogAD v.2 is improved and expanded to calculate the value of the input function, either a scalar function or a matrix function, and the product of the Jacobian of the function and an arbitrary vector. The evaluation procedure is listed in Table 2.3. The changes to LogAD v.2 include:

- Fix errors of LogAD v.2 functions, to name a few, which include getting index assignment and several other functions to work and produce correct results.

- Change the original idea of recording integers on the tape as bytes which caused segmentation faults for large computations.

- Change the data type for computation size related numbers so that LogAD can work for larger computation sizes.
- LogAD v.2 accepts and evaluates scalar function(s) only; LogAD v.2.1 accepts and evaluates both scalar and matrix functions;
- LogAD v.2 internally initialize the adjoint(s) for the scalar function(s) to be 1; LogAD v.2.1 accepts user provided initialization adjoint values.

the prototype of the gdiff function in LogAD v.2.1 is:

```
gdiff ("<fname>", <v>, <x1>, <a1>, <x2>, <a2>, ...)
```

where the second argument is the initial adjoint value(s), and the rest arguments share the same meaning as that of the former version.

4.3 Two simple examples

In this section we will reexamine the two simple examples we examined in Chapter 3 for the reverse mode automatic differentiation.

4.3.1 Example 1

In this section, the first example in Chapter 3 will be reexamined for reverse mode AD. Let $x = (5.00000\ 0.78540)$. The computation can be done by running the dynamically linked function gdiff in Octave environment:

```
[a, b, c] = gdiff("simple_example1", [1, 1], x, 1)
```

The output of this computation is listed in figure 4.1.

```
a = 3.5355

b = 3.5355

c =

   1.4142e+00   4.4409e-16
```

Figure 4.1: Results of the computation for example 1.

4.3.2 Example 2

In this section, the second example in Chapter 3 will be reexamined for reverse mode AD. Let $x = ones(2)$, which is a 2 *by* 2 matrix with all elements being 1. The computation can be done by running the dynamically linked function gdiff in Octave environment:

```
[a, b] = gdiff("simple_example2", x, x, 1, 4, 0)
```

The output of this computation is listed in figure 4.2. Notice that though both forward mode AD and reverse mode AD reached same numerical values for '*b*', they carried completely different mathematical origins.

```
a =

   8   8
   8   8

b =

  32  32
  32  32
```

Figure 4.2: Results of the computation for example 2.

Chapter 5

Software performance

In this chapter we test the performance of FAD v.1 and LogAD v.2.1 by running three examples with different input sizes and computation lengths. The tests were performed on a 500 MHz Intel Pentium III processor with 192M of RAM. The operating system was the Red Hat Linux 7.0. The Octave 2.0.16 environment was used.

The three examples are the following functions: the first function is a Helmholtz energy function, which has multiple independent variables and one dependent variable; the second function is to calculate 1 to *n*th power of a scalar, which has one independent variable and multiple dependent variables; and the third function is to calculate *n*th power of an input matrix, which has multiple independent variables and multiple dependent variables.

5.1 Helmholtz energy function

In [12], Griewank gave some performance statistics on the Helmholtz function of a mixed fluid:

$$f(x) = RT \sum_{i=1}^{n} \log\left(\frac{x_i}{1-b^T x}\right) - \frac{x^T A x}{\sqrt{8} b^T x} \log\left(\frac{1+(1+\sqrt{2})b^T x}{1+(1-\sqrt{2})b^T x}\right) \tag{5.1}$$

where R is the universal gas constant and:

$$0 \leq x,\ b \in \mathbf{R}^n, \quad A = A^T \in \mathbf{R}^{n \times n}.$$

In this section, we will test FAD and LogAD by using this example. Figure 5.1 lists the Octave program for the problem, where RT is set to be 1, A is chosen to be the identity matrix, and b is chosen to be x for simplicity. Notice that $x^T x$ is calculated twice to get the computation longer.

5.1.1 Performance of FAD v.1

The tests were performed for the cases with $n = 1000 - 100000$, $\dot{x} = ones(1,n)$, and $x = 0.001 * ones(1,n)$, where $ones(1,n)$ is a matrix of 1 *by* n elements that are all 1*s*. The performance of FAD is listed in Table 5.1, where T_{fad} is the time costs to evaluate the function and the tangent, which is the summation of all partial derivatives $\sum_{i=1}^{n}(\partial y/\partial x_i)$ in this case, and T is the time to evaluate the function only.

```
#helm.m
function result = helm(x, n)
RT = 1;
A = 1;

bTx = 0;
for i=1:n
  bTx = bTx + x(i) * x(i);
end

xTAx = 0;
for i=1:n
  xTAx = xTAx + x(i) * x(i);
end

result = 0;
for i=1:n
  result = result + x(i) * log(x(i)/(1-bTx));
end

result = RT * result;
temp = xTAx/2.8284271/bTx;
temp = temp * log((1+(1+1.4142136)*bTx)/(1+(1-1.4142136)*bTx));
result = result - temp;
end
```

Figure 5.1: File helm.m in Octave language.

The experiments were run several times for each *n*, and the average time is used for better statistics.

Figure 5.2 plotted the time costs versus the number of independent variables *n*, and Figure 5.3 plotted the ratio T_{fad}/T versus number of independent variables *n*. Figure 5.3 shows that for relatively large *n*s, the ratio T_{fad}/T is approximately a constant, 1.25 ± 0.04 as shown in Figure 5.3 as a horizontal line , which implies

n		1000	5000	10000	20000	30000	40000
T_{fad}	(clock ticks)	2.0	9.6	19.5	37.4	52.0	76.8
T	(clock ticks)	1.7	7.6	15.7	28.8	42.5	60.0
n		50000	60000	70000	80000	90000	100000
T_{fad}	(clock ticks)	94.3	114.0	138.0	155.0	176.3	195.4
T	(clock ticks)	73.3	94.0	110.0	121.2	138.7	154.0

Table 5.2: Time costs by running FAD for Helmholtz function.

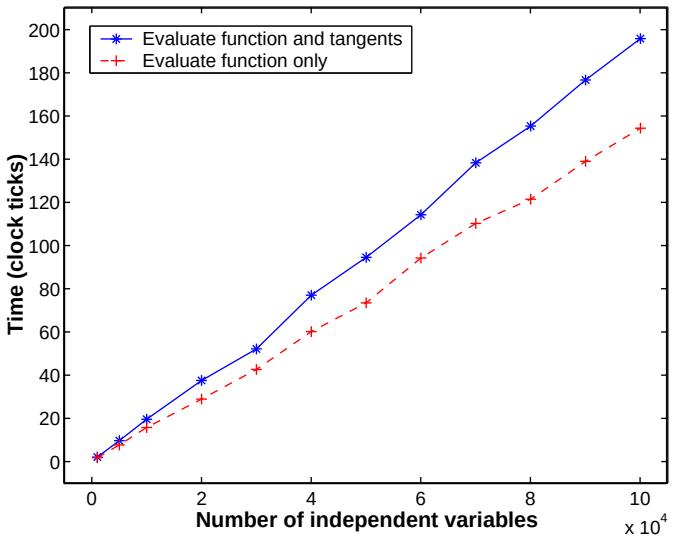

Figure 5.2: Time costs versus the number of independent variables n.

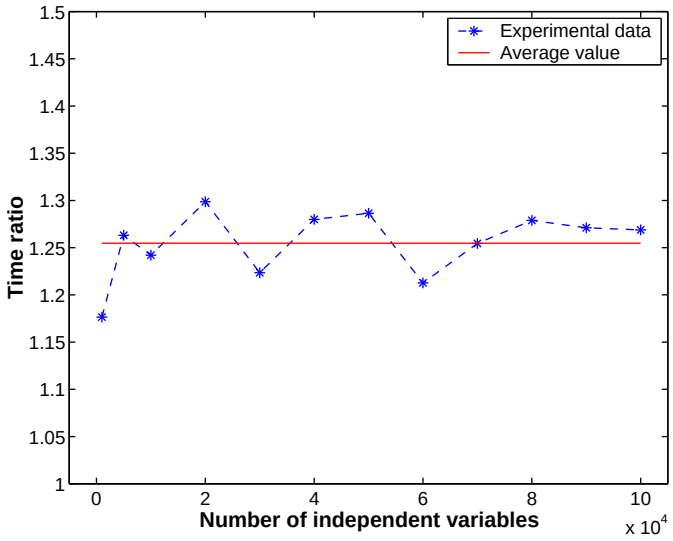

Figure 5.3: The ratio T_{fad}/T versus number of independent variables n.

that the FAD time cost to evaluate the Helmholtz function and the product of the Jacobian and a vector is approximately proportional to the time cost to evaluate the function only.

5.1.2 Performance of LogAD v.2.1

The tests were performed for the cases with $n = 1000 - 100000$, $\bar{y} = 1$, and $x = 0.001 * ones(1,n)$. Table 5.2 lists the computation sizes at different number of independent variables n. Two tape sizes used were: 60001036 bytes, which is the computation size of the largest computation tested, and 1Mb. The performances of LogAD are listed in Table 5.3 for 60001036 bytes tape size, and in Table 5.4 for 1M tape size, where T_{logad} is the time costs to evaluate the function and the gradient,

n	1000	5000	10000	20000
Computation size	601036	3001036	6001036	12001036
n	30000	40000	50000	60000
Computation size	18001036	24001036	30001036	36001036
n	70000	80000	90000	100000
Computation size	42001036	48001036	54001036	60001036

Table 5.2: Computation sizes by running LogAD for Helmholtz function.

n	1000	5000	10000	20000	30000	40000
T_{logad} (clock ticks)	6.6	33.6	69.0	135.8	202.0	271.3
CP	1	1	1	1	1	1
n	50000	60000	70000	80000	90000	100000
T_{logad} (clock ticks)	333.0	397.0	477.5	544.0	609.5	676.5
CP	1	1	1	1	1	1

Table 5.3: Time costs and check points used by running LogAD for Helmholtz function with 60001036 bytes size tape.

n	1000	5000	10000	20000	30000	40000
T_{logad} (clock ticks)	7.2	44.0	100.0	225.3	373.0	499.3
CP	1	4	8	16	32	32
n	50000	60000	70000	80000	90000	100000
T_{logad} (clock ticks)	620.7	815.0	956.0	1095.0	1231.5	1377.0
CP	32	64	64	64	64	64

Table 5.4: Time costs and check points used by running LogAD for Helmholtz function with 1M size tape.

which is the partial derivatives $(\partial y/\partial x_1, \partial y/\partial x_2, \cdots, \partial y/\partial x_n)$ in this case, and CP is the number of check points used by running LogAD. The experiments were run several times for each n, and the average time is used for better statistics.

Figure 5.4 plotted the time costs versus the number of independent variables n, and Figure 5.5 plotted the ratio T_{logad}/T versus the number of independent variables

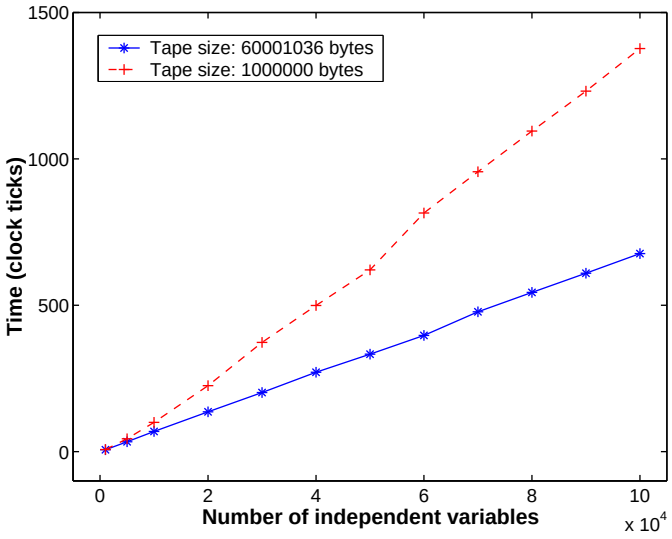

Figure 5.4: Time costs versus the number of independent variables n.

n, where T is the time cost to evaluate the input function only which was evaluated at previous section by FAD, and Figure 5.6 plotted the ratio $T_{logad}/(T\log_2 T)$ versus number of independent variables n. Figure 5.4 and Figure 5.5 show that for relatively large tape size, i.e. 60001036 bytes tape size, the the ratio T_{logad}/T is approximately a constant, 4.42 ± 0.22. When the computation size is checked, we can see that the tape size is large enough to record all the computation at once for all

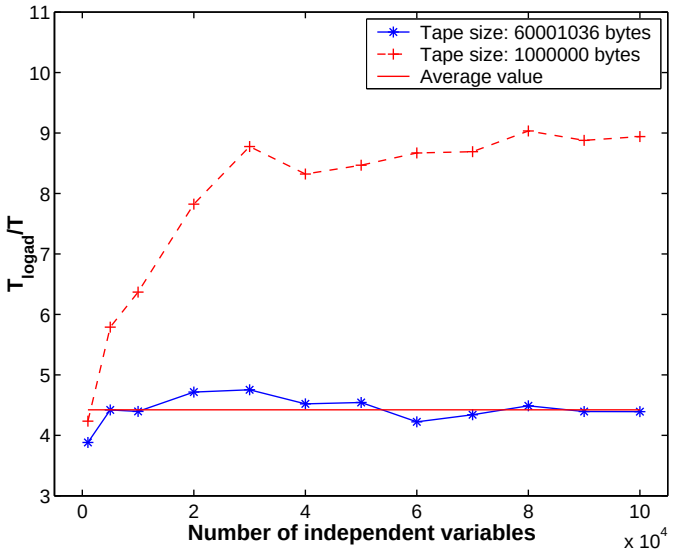

Figure 5.5: Time ratio T_{logad}/T versus the number of independent variables n.

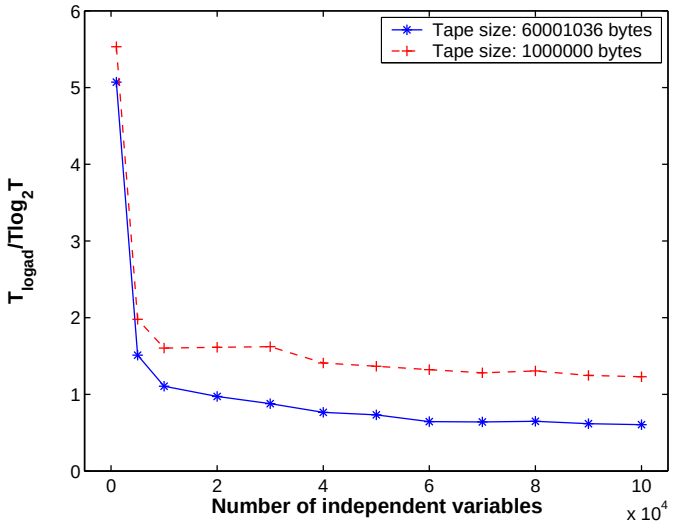

Figure 5.6: The ratio $T_{logad}/(T\log_2 T)$ versus number of independent variables n.

tested *n*s, therefore no checkpointing is needed. For this case, the space used is the computation size, which is approximately proportional to the time T as shown in Figure 5.7. These results indicate that when enough space is available, LogAD has

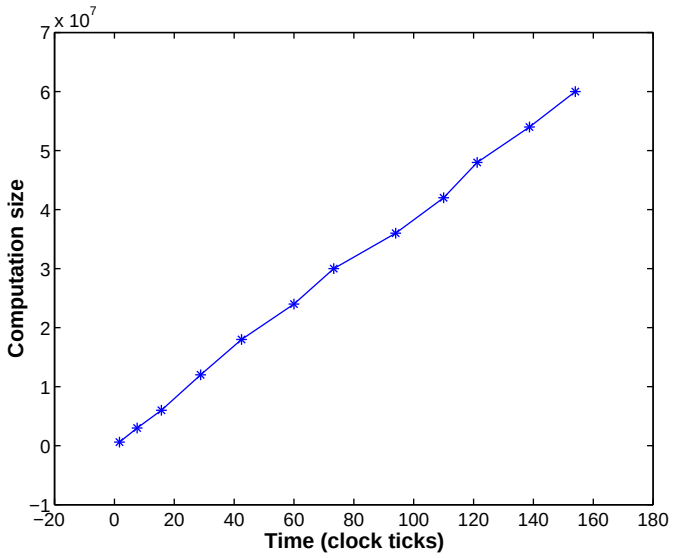

Figure 5.7: The computation size versus time cost for evaluation the original function.

a time complexity and a space complexity which is proportional to the time complexity and space complexity, respectively, of the original algorithm for evaluating function only for this example. The time increment as a function of the input size for tape size 1M, when checkpointing is needed, is more rapid than a linear relation. Figure 5.6 shows that the ratio $T_{logad}/(T\log_2 T)$ decreases when the number of independent variables *n* increases, therefore the LogAD has a better performance than logarithmic increment for temporal complexity for this example, which is due

to, in this case, that the available space is larger than the $S \log_2 T$.

5.2 Power function

In this section, we will evaluate 1 to *n*th power of a scalar variable *x*:

$$y = (y_1, y_2, \cdots, y_n) \quad \text{where } y_i = x^i,\ i = 1, 2, \cdots, n \tag{5.2}$$

Figure 5.8 lists the Octave program for the problem, where *z* is the initialization values for *y*.

```
# power.m
function y = power (x, z, n)
y = z;
y(1) = x;
for i = 2:n
  y(i) = y(i-1) * x;
end
endfunction
```

Figure 5.8: File power.m in Octave language.

5.2.1 Performance of FAD v.1

The tests were performed for the cases with $n = 5000 - 100000$, $x = 1$, $\dot{x} = 1$, and $z = ones(1, n)$, where $ones(1, n)$ is a matrix of 1 *by n* elements that are all 1*s*. The performance of FAD is listed in Table 5.5, where T_{fad} is the time costs

n		5000	10000	20000	30000	40000	50000
T_{fad}	(clock ticks)	2.7	5.1	10.5	14.1	20.5	26.0
T	(clock ticks)	2.0	4.2	8.0	11.1	15.9	19.5
n		60000	70000	80000	90000	100000	
T_{fad}	(clock ticks)	30.2	40.0	43.2	48.5	54.0	
T	(clock ticks)	23.8	30.4	33.0	37.2	41.4	

Table 5.5: Time costs by running FAD for power function.

to evaluate the function and the tangent, which is the derivatives of the function $(\partial y_1/\partial x, \partial y_2/\partial x, \cdots, \partial y_n/\partial x)$ in this case, and T is the time to evaluate the function only. The experiments were run several times for each n, and the average time is used for better statistics.

Figure 5.9 plotted the time costs versus the number of multiplications n, and Figure 5.10 plotted the ratio T_{fad}/T versus number of multiplications n. Figure 5.10 shows that for relatively large ns, the ratio T_{fad}/T is approximately a constant, 1.30 ± 0.04, which implies that the FAD time cost to evaluate the power function and the product of the Jacobian and a vector is approximately proportional to the time cost to evaluate the function only.

5.2.2 Performance of LogAD v.2.1

The tests were performed for the cases with $n = 5000 - 100000$, $x = 1$, $\bar{y} = ones(1,n)$, and $z = ones(1,n)$. Table 5.6 lists the computation sizes at different number of multiplications n. The size of the tape used was 10M. The performance

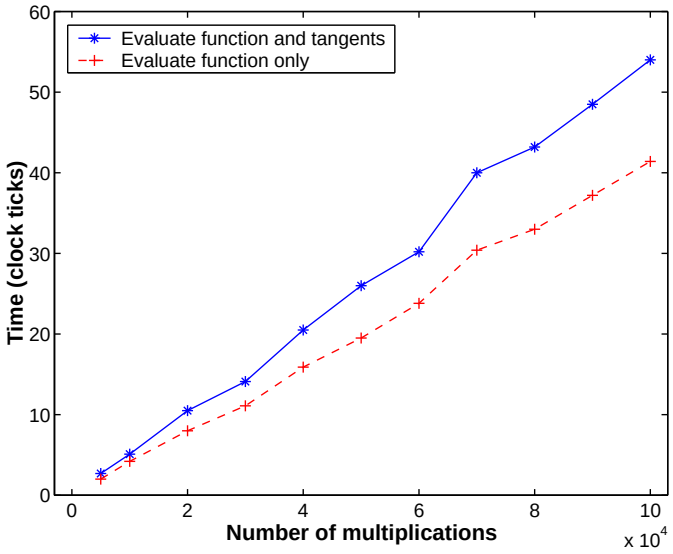

Figure 5.9: Time costs versus the number of multiplications n.

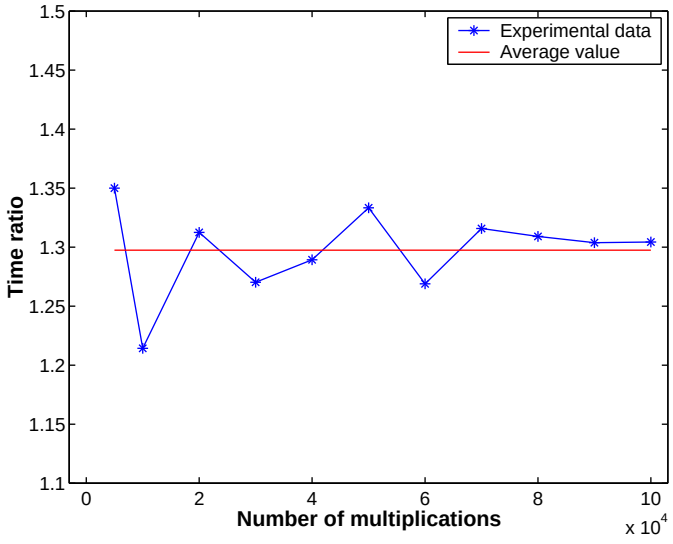

Figure 5.10: The ratio T_{fad}/T versus number of multiplications n.

n	5000	10000	20000	30000
Comput. size	200800044	801600044	3203200044	7204800044
n	40000	50000	60000	70000
Comput. size	12806400044	20008000044	28809600044	39211200044
n	80000	90000	100000	
Comput. size	51212800044	64814400044	80016000044	

Table 5.6: Computation sizes by running LogAD for power function.

of LogAD is listed in Table 5.7. where T_{logad} is the time costs to evaluate the

n	5000	10000	20000	30000	40000	50000
T_{logad}	329.0	1334.3	5207.8	13458.0	25925.0	41822.5
CP	32	128	512	1024	2048	2897
n	60000	70000	80000	90000	100000	
T_{logad}	60948.0	81822.0	108477.5	137302.5	168719.0	
CP	4096	4465	8192	8192	9889	

Table 5.7: Time costs and check points used by running LogAD for the power function, where T_{logad} is measured in clock ticks.

function and the gradient, which is the summation of the derivatives $\sum_{i=1}^{n}(\partial y_i/\partial x)$, and CP is the check points used by running LogAD. The experiments were run several times for each n, and the average time is used for better statistics.

Figure 5.11 plotted the time costs versus the number of multiplications n, Figure 5.12 plotted the time ratio T_{logad}/T versus number of multiplications n, where T is the time cost to evaluate the input function only which was evaluated at previous section by FAD, Figure 5.13 plotted the ratio $T_{logad}/(T\log_2 T)$ versus number of

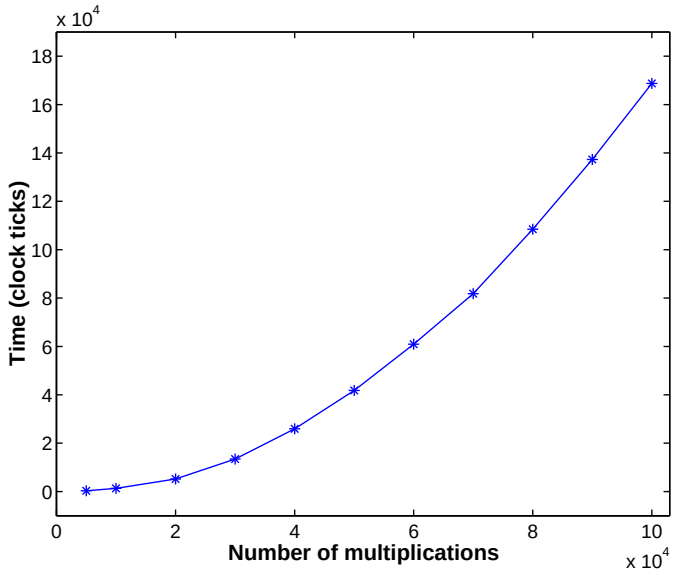

Figure 5.11: Time costs versus the number of multiplications n.

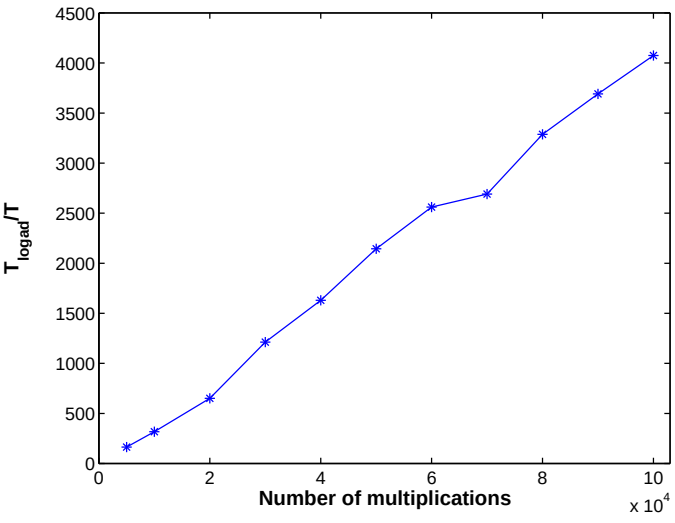

Figure 5.12: Time ratio T_{logad}/T versus the number of multiplications n.

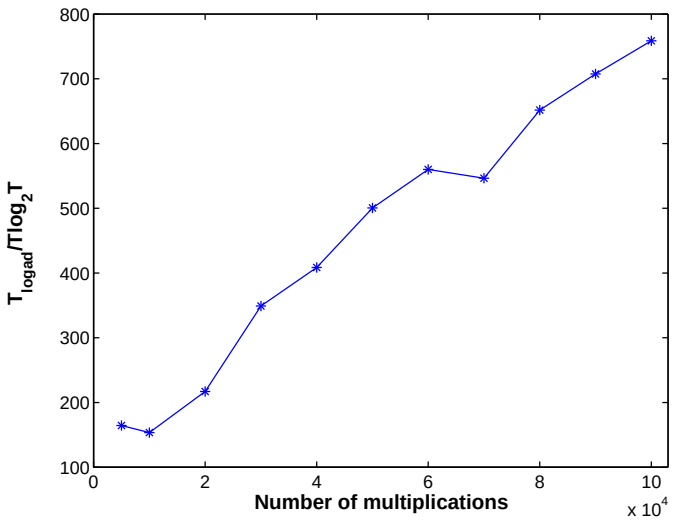

Figure 5.13: The ratio $T_{logad}/(T\log_2 T)$ versus number of multiplications n.

multiplications n, and Figure 5.14 plotted the ratio T_{logad}/T^2 versus number of multiplications n. The above figures show that the time costs as a function of the number of multiplications n increases more rapidly than a linear relation and a logarithmic relation. Figure 5.14 shows that the ratio T_{logad}/T^2 is approximately a constant, 96 ± 12. Therefore the performance of LogAD for this example is worse than logarithmic increment but close to T^2 for temporal complexity. When the ratio of computation size to T^2 as a function of n is checked, which is plotted in Figure 5.15, we found that the implementation of LogAD, which is due to the implementation of indexed assignment, for this example has a time complexity which is approximately proportional to T^2. Therefore the more rapid time increment than logarithmic relation is not due to the checkpointing algorithm, but the implementa-

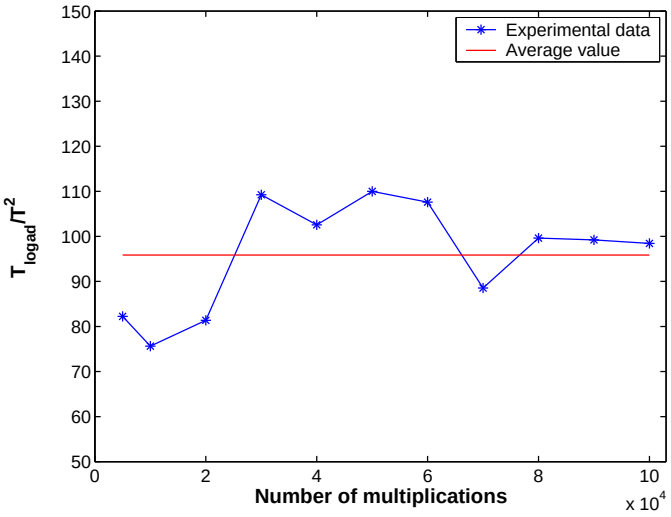

Figure 5.14: The ratio T_{logad}/T^2 versus the number of multiplications n.

tion of general reverse sweep in this case.

5.3 Matrix multiplication

In this section, we will evaluate the nth power of a matrix x:

$$y = \prod_{i=1}^{n} x \tag{5.3}$$

The Octave program for the problem is listed in Figure 3.3 in Chapter 3.

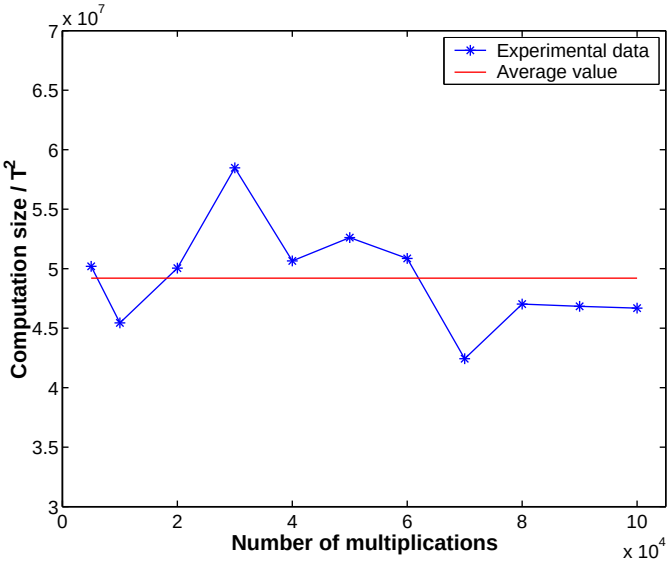

Figure 5.15: The ratio T_{logad}/T^2 versus the number of multiplications n.

5.3.1 Performance of FAD v.1

The tests were performed for the cases with $n = 1000 - 1000000$, x being a 9 *by* 9 identity matrix, and $\dot{x} = ones(9)$, where $ones(9)$ is a 9 *by* 9 matrix with all elements being 1. The performance of FAD is listed in Table 5.8, where T_{fad} is

n	1000	10000	50000	100000
T_{fad}	8.6	86.5	434.2	872.0
T	3.8	35.3	174.5	349.8
n	250000	500000	750000	1000000
T_{fad}	2162.3	4345.3	6509.0	8644.0
T	870.0	1746.8	2610.8	3462.3

Table 5.8: Time costs by running FAD for matrix multiplication function, where time is measured in clock ticks.

the time costs to evaluate the function and the tangent, and T is the time costs to evaluate the function only. The experiments were run several times for each n, and the average time is used for better statistics.

Figure 5.16 plotted the time costs versus the number of matrix multiplications n, and Figure 5.17 plotted the ratio T_{fad}/T versus number of matrix multiplications

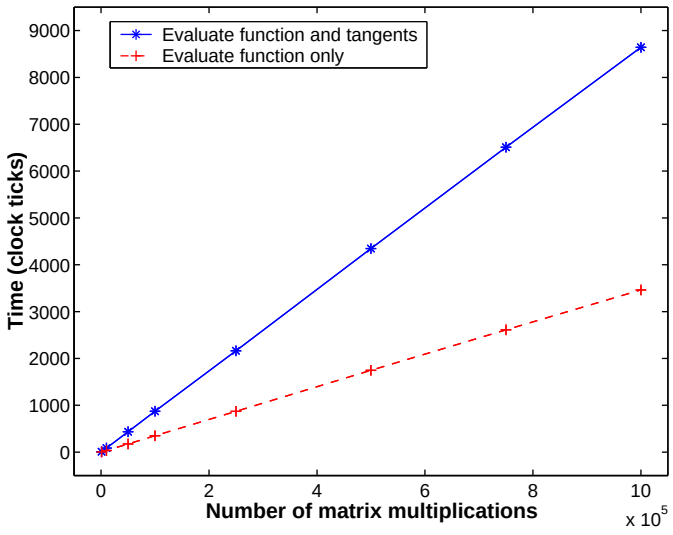

Figure 5.16: Time costs versus the number of matrix multiplications n.

n. Figure 5.17 shows that for relatively large ns, the ratio T_{fad}/T is approximately a constant, 2.489 ± 0.003, which implies that the FAD time cost to evaluate the matrix multiplication function and the product of the Jacobian and a vector is approximately proportional to the time cost to evaluate the function only.

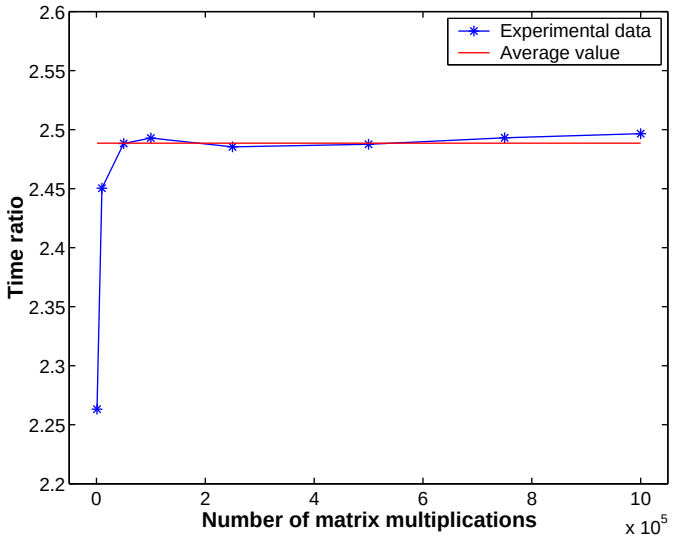

Figure 5.17: The ratio T_{fad}/T versus number of matrix multiplications n.

5.3.2 Performance of LogAD v.2.1

The tests were performed for the cases with $n = 1000 - 1000000$, $\bar{y} = ones(9)$, and x being the 9 *by* 9 identity matrix, where $ones(9)$ is a 9 *by* 9 matrix with all elements being 1. Table 5.9 lists the computation sizes at different number of matrix multiplications n. Two tape sizes used were: 10M and 1M. The performances of

n	1000	10000	50000	100000
Comput. size	1368152	13680152	68400152	136800152
n	250000	500000	750000	1000000
Comput. size	342000152	684000152	1026000152	1368000152

Table 5.9: Computation size by running LogAD for matrix multiplication function.

LogAD are listed in Table 5.10 for 10M tape size, and in Table 5.11 for 1M tape size, where T_{logad} is the time costs to evaluate the function and the gradients, and

n	1000	10000	50000	100000
T_{logad} (clock ticks)	18.4	204.5	1220.3	2628.0
CP	1	2	8	16
n	250000	500000	750000	1000000
T_{logad} (clock ticks)	7521.5	15983.5	23927.5	34424.0
CP	64	128	128	256

Table 5.10: Time costs and check points used by running LogAD for matrix multiplication function with 10M size tape.

CP is the number of check points used by running LogAD. The experiments were run several times for each n, and the average time is used for better statistics.

Figure 5.18 plotted the time costs versus the number of independent variables n, Figure 5.19 plotted the ratio T_{logad}/T versus the number of independent variables n, where T is the time cost to evaluate the input function only which was evaluated at previous section by FAD, and Figure 5.20 plotted the ratio $T_{logad}/(T\log_2 T)$ versus number of matrix multiplications n. The above figures show that the time increment for both tape sizes are more rapid than linear relation and close to logarithmic relation for large input sizes.

n	1000	10000	50000	100000
T_{logad} (clock ticks)	20.5	260.0	1579.8	3344.8
CP	2	16	128	256
n	250000	500000	750000	1000000
T_{logad} (clock ticks)	8844.5	18668.5	29327.5	39171.5
CP	512	1024	2048	2048

Table 5.11: Time costs and check points used by running LogAD for matrix multiplication function with 1M size tape.

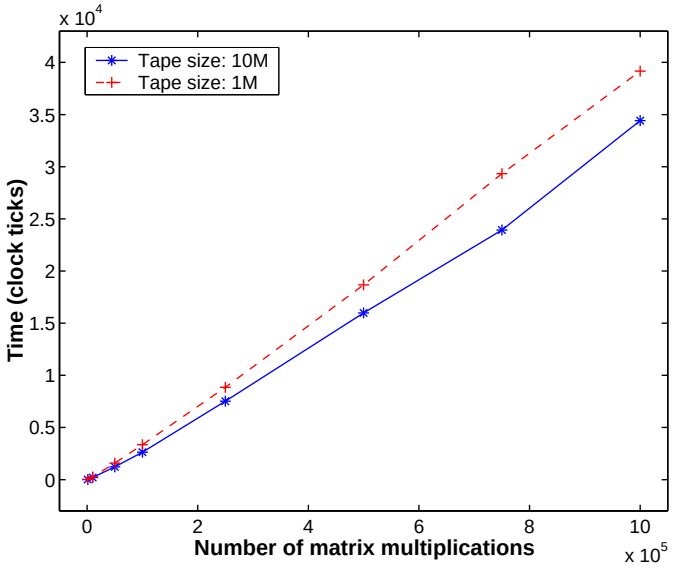

Figure 5.18: Time costs versus the number of matrix multiplications n.

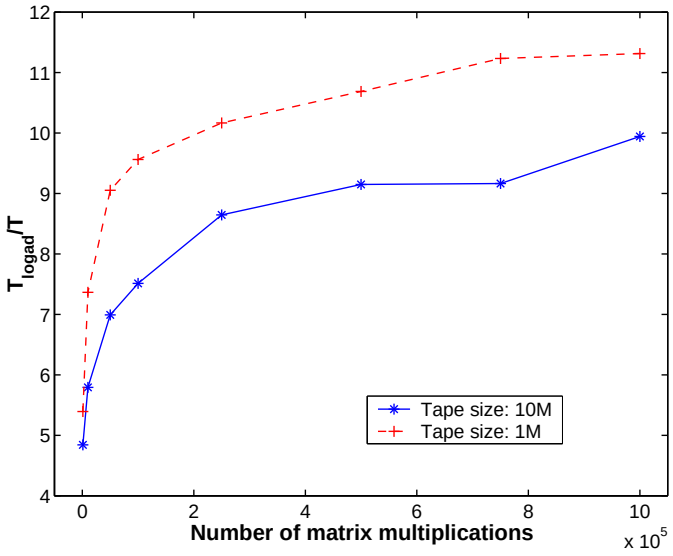

Figure 5.19: The ratio T_{logad}/T versus the number of matrix multiplications n.

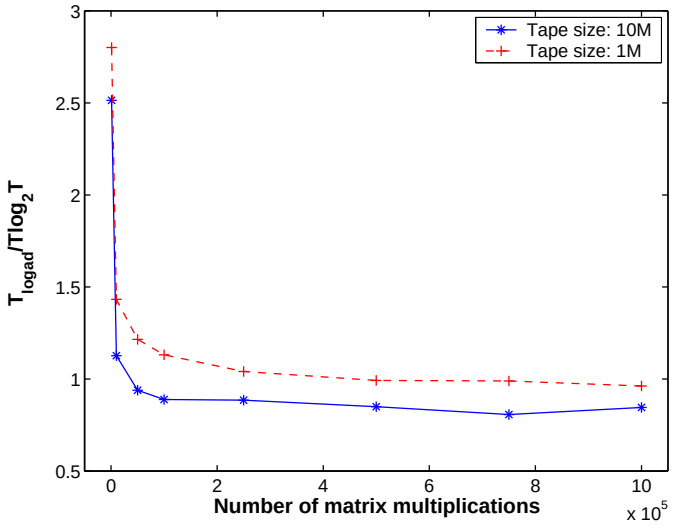

Figure 5.20: The ratio $T_{logad}/(T\log_2 T)$ versus number of matrix multiplications n.

Chapter 6

Conclusions

In this work, we have implemented the general forward mode AD and general reverse mode AD with bisection checkpointing for GNU Octave applications.

The implementation of general forward mode AD, FAD, evaluates the product of the Jacobian of a function and an arbitrary vector. When the function is a set of the scalar functions with one common independent variable, by setting all the elements of the vector to be 1, the result is the derivatives of the scalar functions with respect to the common independent variable (such as the power function in Chapter 5). The test results in Chapter 5 showed that the FAD temporal complexity is proportional to the time complexity of the original algorithm for evaluating the input function only. Therefore, when only one independent variable is presented, the derivatives of a number of functions respect to the only independent variable can be evaluated with a cost which is proportional to that of the cost for evaluating

the original functions.

The implementation of the general reverse mode AD, LogAD, evaluates the product of an arbitrary vector and the Jacobian of a function. For a scalar function with multiple independent variables, by setting all the elements of the vector to be 1, the program returns the gradient of the function (such as the Helmholtz energy function in Chapter 5). When no space limit is presented, the test results in Chapter 5 showed that the LogAD temporal complexity could be proportional to that of the original function (such as running LogAD for the Helmholtz function with large tape size). When the space is limited, LogAD can also perform the task by checkpointing the computation with the price of longer computation time.

Bibliography

[1] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[2] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. Programs for automatic differentiation for the machine BESM. Technical report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, 1959.

[3] M. Berz. Forward algorithms for higher derivatives in many variables with applications to beam physics. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 147–156. SIAM, Philadelphia, Penn., 1991.

[4] C. H. Bischof and A. Carle. Users' experience with ADIFOR 2.0. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 385–392. SIAM, Philadelphia, Penn., 1996.

[5] C. H. Bischof, C. F. Corliss, L. Green, A. Griewank, K. Haigler, and P. Newman. Automatic differentiation of advanced CFD codes for multidisciplinary design. *Journal on Computing Systems in Engineering*, 3:625–638, 1992.

[6] C. H. Bischof, G. Pusch, and R. Knoesel. Sensitivity analysis of the MM5 weather model using automatic differentiation. *Computers in Physics*, 0:605–612, 1996.

[7] A. Carle, L. Green, C. H. Bischof, and P. Newman. Applications of automatic differentiation in CFD. In *Proceedings of the 25th AIAA Fluid Dynamics Conference, AIAA Paper 94-2197*. American Institute of Aeronautics and Astronautics, 1994.

[8] T. F. Coleman and A. Verma. ADMIT-1: Automatic differentiation and matlab interface toolbox. *ACM Transactions on mathematical Software*, 26(1):150–175, 2000.

[9] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, 1998.

[10] V. Goldman and G. Cats. Automatic adjoint modeling within a program generation framework: A case study for a weather forecasting grid-point model. In M. Berz, C. Bischof, G. Corliss, and Andreas Griewank, editors, *Compu-*

tational Differentiation: Techniques, Applications, and Tools, pages 185–194. SIAM, Philadelphia, Penn., 1996.

[11] L. Green, P. Newman, and K. Haigler. Sensitivity derivatives for advanced CFD algorithm and viscous modeling parameters via automatic differentiation. In *Proceedings of the 11th AIAA Computational Fluid Dynamics Conference, AIAA Paper 93-3321*. American Institute of Aeronautics and Astronautics, 1993.

[12] A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–107. KTK Scientific Publishers, Tokyo, 1989.

[13] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[14] A. Griewank. *Evaluating Derivatives: principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, Penn., 2000.

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

[16] R. Heller. Checkpointing without operating system intervention: Implementing griewank's algorithm. Master's thesis, Ohio University, August 1998.

[17] D. W. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991.

[18] D. W. Juedes and K. Balakrishnan. Generalized neural networks, computational differentiation, and evolution. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 273–285. SIAM, Philadelphia, Penn., 1996.

[19] H. Kagiwada, R. Kalaba, N. Rasakhoo, and K. Spingarn. *Numerical derivatives and nonlinear analysis*. Math. Concepts Methods Sci. Engrg. Plenum Press, New Yok, London, 1986.

[20] G. Kedem. Automatic differentiation of computer programs. *ACM transactions on mathematical software*, 6(2):150–165, 1980.

[21] K. Kubota. PADRE2 – Fortran precompiler for automatic differentiation and estimates of rounding error. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 367–374. SIAM, Philadelphia, Penn., 1996.

[22] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT*, 16(1):146–160, 1976.

[23] B. Mohammadi, J. Male, and N. Rostaig-Schmidt. Automatic differentiation in direct and reverse modes: Application to optimum shapes design in fluid mechanics. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 309–318. SIAM, Philadelphia, Penn., 1996.

[24] E. M. Oblow. An automated procedure for sensitivity analysis using computer calculus. Technical Report ORNL/TM-8776, Oak Ridge national laboratory, Oak Ridge, TN, 1983.

[25] L. B. Rall. *Automatic differentiation: Techniques and applications*, volume 120 of *Lecture Notes in Comput. Sci.* Springer-Verlag, Berlin, 1981.

[26] L. B. Rall. Differentiation in pascal-SC: Type GRADIENT. *ACM transactions on mathematical software*, 10:161–184, 1984.

[27] L. C. Rich and D. R. Hill. Automatic differentiation in matlab. *Applied numerical mathematics*, 9(1):33–43, 1992.

[28] N. Rostaing-Schmidt. *Différentiation automatique: Application à un problème d'optimisation en météorologie*. PhD thesis, Université de Nice, Sophia Antipolis, France, 1993.

[29] S. Saarinen, R. B. Bramley, and G. Cybenko. Neural networks, backpropagation, and automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 31–42. SIAM, Philadelphia, Penn., 1991.

[30] B. Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, Ill., January 1980.

[31] A. Stovboun. A tool for creating high-speed, memory efficient derivative codes for large scale applications. Master's thesis, Ohio University, August 2000.

[32] Y. M. Volin and G. M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating techniques – I: Algorithmic basis. *Comput. math. Appl.*, 11:1099–1114, 1985.

[33] R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

[34] P. Werbos. *Beyond Regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, Cambridge, Mass., November 1974.

[35] R. D. Wilkins. Investigation of a new analytical method for numerical derivative evaluation. *Communications of the ACM*, 7(8):465–471, 1964.

Appendix A

A brief guide to use FAD/LogAD

System requirement:

Linux/Unix operating system, GNU make, GNU C++ compiler, GNU Octave, and other necessary utilities.

Step 1: Compile and install the FAD/LogAD

1. Edit ’config.mk’
2. Make the package as follows:

- Install: gmake install

  gmake install INSTALLDIR=<DIR>

- Uninstall: gmake uninstall

gmake uninstall INSTALLDIR=<DIR>

- Just compile: gmake

  gmake debug

- Clean: gmake clean

  gmake cleanall

3. Copy fdiff.oct/gdiff.oct from the directory src/engine into the directory where Octave looks for dynamically linked functions, or you can specify the path to search for function files using command line options -p or –path when start Octave.

Step 2: Compile the functions in Octave language, the .m files, and link the function

1. Compile each related <fname>.m files: fadc/logadc <fname>.m
2. Link the function: fadl/logadl <fname>

Step 3: Run the dynamically linked function fdiff/gdiff in Octave environment

1. Start Octave

2. Set FAD/LogAD run time parameters:

   FAD_STACK_SIZE = <value1>

   FAD_ARG_BUF_SIZE = <value2>

   or

   LOGAD_TAPE_SIZE = <value1>

   LOGAD_STACK_SIZE = <value2>

   LOGAD_ARG_BUF_SIZE = <value3>

3. Call the dynamically linked function fdiff/gdiff with the following format:

   [r1, r2, ...] = fdiff ("<fname>", <a>, <v>, <x1>, <x2>, ...)

   or

   [r1, r2, ...] = gdiff ("<fname>", <v>, <x1>, <a1>, <x2>, <a2>, ...)

Example: Helmholtz energy function

In this section, I will list the steps to test Helmholtz energy function which was given in Chapter 5 Figure 5.1.

1. Install FAD/LogAD and copy fdiff.oct/gdiff.oct from the directory src/engine into the directory where Octave looks for dynamically linked functions.

2. Compile and link files:

   fadc/logadc helm.m

   (do this for all dependency .m files if there is any)

fadl/logadl helm

3. Start Octave and do the following in Octave environment

Run FAD:

```
FAD_ARG_BUF_SIZE = 100;

FAD_STACK_SIZE = 100;

[a, b] = fdiff("helm", 1, [1, 1], 0.001*[1, 1], 2)
```

Run LogAD:

```
LOGAD_TAPE_SIZE = 1000;

LOGAD_ARG_BUF_SIZE = 100;

LOGAD_STACK_SIZE = 100;

[a, b] = gdiff("helm", 1, 0.001*[1, 1], 1, 2, 0)
```

Note:

By the time this thesis has been written, FAD/LogAD has been tested only on a computer running Red Hat Linux 7.0, with GNU make version3.79.1, GNU gcc version 2.96, and GNU Octave version 2.0.16.