A TOOL FOR CREATING HIGH-SPEED, MEMORY EFFICIENT DERIVATIVE CODES FOR LARGE SCALE APPLICATIONS

A Thesis Presented To

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology
Ohio University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by

Alexei Stovboun
August, 2000
THIS THESIS ENTITLED
“A TOOL FOR CREATING HIGH-SPEED, MEMORY EFFICIENT DERIVATIVE CODES FOR LARGE SCALE APPLICATIONS”

by Alexei Stovboun

has been approved

for the School of Electrical Engineering and Computer Science
and the Russ College of Engineering and Technology

Dr. David Juedes
Associate Professor of Electrical Engineering and Computer Science

Jerrel R. Mitchell, Dean
Fritz J. and Dolores H. Russ
College of Engineering and Technology
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Differentiation Techniques</td>
<td>2</td>
</tr>
<tr>
<td>1.2 LogAD System</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Thesis Organization</td>
<td>9</td>
</tr>
<tr>
<td>2. AUTOMATIC DIFFERENTIATION METHODS: MATHEMATICAL BACKGROUND</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Forward Mode</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Reverse Mode</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Adjoint Equations</td>
<td>14</td>
</tr>
<tr>
<td>2.4 Reverse Mode with Checkpointing: Griewank's Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>3. LOGAD ARCHITECTURE</td>
<td>22</td>
</tr>
<tr>
<td>3.1 LogAD Compiler</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1 Expressions and Index Expressions</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2 Function Calls</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 <em>if-elseif-else</em> statements</td>
<td>27</td>
</tr>
<tr>
<td>3.1.4 <em>while</em> loop</td>
<td>28</td>
</tr>
<tr>
<td>3.1.5 <em>for</em> loop</td>
<td>29</td>
</tr>
<tr>
<td>3.2 LogAD Linker</td>
<td>31</td>
</tr>
<tr>
<td>3.3 LogAD Shared Library</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1 Execution Environment</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2 Checkpointing</td>
<td>34</td>
</tr>
<tr>
<td>3.3.3 Tape</td>
<td>35</td>
</tr>
</tbody>
</table>
3.4 Memory Pooling

4. LOGAD RESULTS
   4.1 Chain Matrix Multiplications
   4.2 Helmholtz Energy Function

5. CONCLUSIONS AND FUTURE RESEARCH

BIBLIOGRAPHY

APPENDIX

A. LOGAD CONFIGURATION AND COMMAND LINE ARGUMENTS
   A.1 Configuration
   A.2 LogAD Compiler
   A.3 LogAD Linker

B. LOGAD INPUT LANGUAGE GRAMMAR
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Computing Gradient of $F$ in Reverse Mode</td>
</tr>
<tr>
<td>4.1</td>
<td>Number of Multiplications and Computation Sizes for Matrix Multiplications</td>
</tr>
<tr>
<td>4.2</td>
<td>Number of Independent Variables and Computation Sizes for Helmholtz Energy</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Files product.m and cube.m</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Producing Shared Library from Octave Source Code</td>
<td>7</td>
</tr>
<tr>
<td>1.3 File cube.logad</td>
<td>8</td>
</tr>
<tr>
<td>1.4 Results of Running product</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Griewank's Algorithm Using Bisectioning</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Pseudo-code of LogAD Checkpoint Manager</td>
<td>20</td>
</tr>
<tr>
<td>3.1 LogAD Structure</td>
<td>22</td>
</tr>
<tr>
<td>3.2 GNU Extension: Labels as Values</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Compiled Expression Octave Expression</td>
<td>26</td>
</tr>
<tr>
<td>3.4 Compiled Function Call</td>
<td>28</td>
</tr>
<tr>
<td>3.5 Octave if-elseif-else Statement</td>
<td>28</td>
</tr>
<tr>
<td>3.6 Compiled if-elseif-else Statement</td>
<td>29</td>
</tr>
<tr>
<td>3.7 Octave while Statement</td>
<td>30</td>
</tr>
<tr>
<td>3.8 Compiled while Statement</td>
<td>30</td>
</tr>
<tr>
<td>3.9 Octave for Statement</td>
<td>31</td>
</tr>
<tr>
<td>3.10 Compiled for Statement</td>
<td>31</td>
</tr>
<tr>
<td>3.11 LogAD Execution Stacks</td>
<td>33</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>3.12 LogAD Tape Structure</td>
<td>36</td>
</tr>
<tr>
<td>4.1 LogAD and ADOL-C Time Results for Matrix Multiplications</td>
<td>38</td>
</tr>
<tr>
<td>4.2 ADOL-C Space Results for Matrix Multiplications</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Number of Checkpoints for Matrix Multiplications</td>
<td>40</td>
</tr>
<tr>
<td>4.4 LogAD and ADOL-C Time Results for Helmholtz Energy</td>
<td>42</td>
</tr>
<tr>
<td>4.5 ADOL-C Space Results for Helmholtz Energy</td>
<td>43</td>
</tr>
<tr>
<td>4.6 Number of Checkpoints for Helmholtz Energy</td>
<td>44</td>
</tr>
<tr>
<td>4.7 <em>Tape Size - Time</em> Dependency for Helmholtz Energy</td>
<td>45</td>
</tr>
<tr>
<td>4.8 <em>Tape Size - Number of Checkpoint</em> Dependency for Helmholtz Energy</td>
<td>46</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The exact and efficient calculation of first-order derivatives is important to many scientific applications ranging from the very large scale such as weather forecasting [BPK96] to medium and small scale application such as fluid dynamics [GNH93, CGBN94, MMRS96], beam physics [Ber91], neural networks [SBC91, JB96, BC96], and robotics applications that use inverse kinematics. Most existing differentiation tools [BCKM96, GJU96, etc.] work best for small or medium sized applications, but perform poorly on large problems. In this thesis, we present a tool for computing partial derivatives for large-scale applications, called LogAD v.2, which uses and expands the ideas of Heller’s LogAD tool \(^1\) [Hel98].

LogAD v.2 uses the reverse mode of automatic differentiation with checkpointing, which allows creating derivative codes for large scale applications. To improve the performance, the tool does not use operating system intervention – checkpoints are created by compiler generated code. Unlike most current differentiation tools that accept algorithms written in traditional programming languages like C [GJU96] or Fortran [BCKM96], LogAD v.2 uses the high-level GNU Octave \(^2\) language, which is similar to the language used by Matlab \(^3\), and works in the Octave environment.

\(^1\) stands for “Logarithmic Automatic Differentiation”
\(^2\) available from ftp://ftp.che.wisc.edu/pub/octave, additional information available from http://www.che.wisc.edu/octave
\(^3\) ©Mathworks, Inc.
1.1 Differentiation Techniques

This section describes several common approaches to compute a gradient of a function. One technique for calculating a partial derivative of a continuous differentiable function $f$ at a point $x_i$ is through using the definition of derivation:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x + he_i) - f(x)}{h},$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is a continuously differentiable function and $e_i$ is the $i$th unit vector. By setting $h$ to some sufficiently "small" number, each value of the gradient can be accurately approximated at any point in the function domain. This method is straightforward, however it introduces a number of problems. First, it is inaccurate since the result depends on a choice of $h$. Second, it is difficult to choose an appropriate value of $h$, since this choice greatly depends on the behavior of the function around the point $x$. Usually, to get a good estimate of a derivative, $h$ must be very small, and this may cause problems due to finite precision arithmetic in the computer. Another obvious problem of this method is inefficiency – to find a gradient, the equation (1.1) must be evaluated for each variable, which makes this method computationally expensive.

There are several other techniques for computing derivatives, but each has its own problems [Gri89, Gri91]. One of them is symbolic differentiation, which is implemented in MathCad 4, Mathematica 5, and some other mathematical packages. Another technique is hand-coding derivatives.

Although quite common, the mentioned methods may be very slow or difficult to implement in certain situations. The key for the automatic and efficient evaluation of derivatives lies in the group of techniques, called automatic (or computational)
differentiation. These techniques work by splitting the function into a sequence of elementary operations with predefined (and well-known) derivative codes and applying the chain rule (see Chapter 2).

There are two major directions in automatic differentiation: the forward mode and the reverse mode. The forward mode works by propagating derivatives from independent variables to dependent variables; i.e., in the same "direction" in which the original function is computed. This approach is straightforward, but at times is quite inefficient. The general implementations require $n$ passes for computing a gradient for $n$ independent variables, which is approximately $n$ times more expensive than computing the original function; however, using the linear combinations for sparse Jacobians [AMB+94, CV96, BH91] can improve the running time significantly.

The alternative to the forward mode is the reverse mode of automatic differentiation, which was independently discovered by Werbos [Wer74], Linnainmaa [Lin76], Speelpenning [Spe80], and Baur and Strassen [BS83]. This mode works by backpropagating partial derivatives from dependent variables to independent variables and can compute all partial derivatives in one "reverse" pass. Hence, this mode is particularly suitable for computing gradients of functions with a large number of independent variables. In fact, computing derivatives of all independent variables with respect to one dependent variable is no more than 5 times more expensive as computing the original function [Gri89].

General implementations of the reverse mode are efficient, but, unfortunately, require that the program be available "in reverse," i.e., the whole computation must be somehow recorded. Therefore, the amount of memory needed is proportional to the run time of the original function. For example, ADOL-C requires approximately 30 megabytes for one minute of evaluation time in the reverse mode on a Sun 3 workstation [GJU96]. Clearly, this memory requirements can limit the applicability of this method for large-scale applications.
In [Gri92], Griewank gives a solution to the memory problem of the reverse mode. He observes that there is no need to record the entire computation. Instead, the computation can be checkpointed at designated points, and a fragment that is about to be reversed can be recomputed from the most recent checkpoint and recorded. Therefore, at any given time only one fragment of the computation is kept in the memory. Although this approach makes high speed differentiation feasible for many large scale problems, it involves a time-space trade off where intermediate results are recomputed instead of being stored. Griewank showed that if the original problem runs in $O(T)$ time, requiring $O(S)$ memory cells, then for the reverse computation, the time requirement is $O(T \log T)$ and the space requirement is $O(S \log T)$ (all logarithms in this thesis are base 2).

The checkpointing scheme was first implemented in the C++ package ADOL-C [Gri92] and later improved by Mauer-Oats in the Checkpoint 1 package [MO96b, MO96a]. These implementations used the UNIX forking and piping facilities for checkpointing the computation. Although Griewank mentioned that checkpointing by forking is “surprisingly simple and even elegant,” it has serious drawbacks. First, the `fork()` system call is expensive in terms of time and system resources because the OS creates an exact copy of the calling process; however, this is only partially true on some systems that implement copy-on-write approach (e.g., Linux). Second, the operating system usually places a limit on the number of processes running simultaneously, which might create inconveniences using system-assisted checkpointing. Third, this solution is not completely portable – it cannot be ported on the systems that have no forking facilities (e.g., Win32).

1.2 LogAD System

The first version of LogAD differentiator implemented by Heller [Hel98] used the reverse mode of automatic differentiation with compiler generated checkpoints and
accepted algorithms written in a subset of C. Although LogAD v.1 was not fully functional, it established many of the implementation specific details. This thesis presents the fully functional second version of LogAD, which also employs checkpointing without operating system intervention. For the performance and portability reasons mentioned above, our implementation uses a compiler-generated code to create checkpoints. At designated points in the computation the control transfers to a decision making code, which is responsible for managing the computation and inserting checkpoints.

LogAD v.2 uses the Octave language as its input language. It is a high-level interpreted language, convenient for engineering and mathematical computations. It is compatible with the language used in Matlab [BB95] by MathWorks, Inc. As of version 2.0.14, Octave implements four built-in data types: real scalar, real matrix, complex scalar, and complex matrix. Octave uses run time operation resolution, which makes it fully polymorphic, i.e., a variable does not have any particular data type, rather it can store any data type. One of the consequences of this is that variables are not declared before their use. Another Octave feature that differs from other languages is function calls. An Octave function can accept and return any number of arguments and the function behavior can change depending on how many arguments have been passed or how many return values are expected by the calling function. Since the calling function generally does not know the number of arguments expected by the called function, arguments are not placed directly on stack but are passed via a special buffer. Although, as of now, LogAD implements only real scalar and real matrix data types and a subset of the Octave language, new language and functional facilities can be added without rewriting the existing code.

For the user convinience, LogAD is integrated into the Octave environment and can be invoked by calling dynamically linked 6 function \texttt{gdiff}. This function takes a name of the function to differentiate and values of its independent variables. It

\footnote{6for information about dynamic linking in Octave see Octave manual}
returns values of the dependent variables and the derivatives. In order to use LogAD the system must support dynamic linking \(^7\).

Consider the process of differentiating an Octave program. Figure 1.1 shows a very simple Octave program that calculates the product of cubes of a vector of values. For

```octave
# product.m
function result = product (v)
  result = 1.0;
  for i = v
    result = result * cube(i);
  endfor
endfunction

# cube.m
function a = cube (b)
  a = b * b * b;
endfunction
```

Figure 1.1 Files product.m and cube.m

the sake of the example, this operation is divided into two functions product and cube that reside in the files product.m and cube.m, respectively. In order to be differentiated, these functions must be compiled and placed into a special shared library product.lso, which then will be dynamically linked by the LogAD. These steps are performed by the LogAD linker logadl (see Figure 1.2). First, it calls the LogAD compiler logadc for each Octave source file to produce fragments of C++ code and the dependency information. The produced C++ code for the function cube is shown in Figure 1.3 (the macro _CHECKP contains the checkpoint manager code, which determines when to insert a checkpoint, start recording, or perform the reverse pass). Then all fragments are combined into one C++ function the_diff_func

\(^7\)check this by issuing `octave_config.info("dld")` command at Octave prompt
and compiled by g++ to produce a special shared library **product.lso**, which can be loaded by LogAD.

Figure 1.4 shows the results of differentiating the function **product** with the input vector [1,2,3,4]. The first argument to **gdiff** is the name of the function to differentiate and must be a string. The next pair of arguments defines the first independent variable $v$. The first element in the pair is the value of the independent variable. The second element in the pair defines an activity status - 1 (or any other
// Function: cube
_f_cube:
{
    static char *names[] = {"a", "b", "vt_0");
    enum {
        _v_a = 3,
        _v_b = 2,
        _vt_0 = 1
    };

    add_frame (3, 1, 1, names);
    _CHECKP(_c_cube_0);
    bi_mult (_vt_0, _v_b, _v_b);
    _CHECKP(_c_cube_1);
    bi_mult (_vt_0, _vt_0, _v_b);
    _CHECKP(_c_cube_2);
    assign (_v_a, _vt_0);

    _f_cube_end:
    _CHECKP(_c_cube_3);
    del_frame(3, 1);

    RETURN;
}

Figure 1.3 File cube.logad

value that evaluates to T R U E) means that the corresponding independent variable is active, and the partial derivatives with respect to this variable are calculated, 0 (or any other value that evaluates to F A L S E) means that the corresponding independent variable is passive, and the partial derivative with respect to this variable are not calculated.

In this example, gdiff returns two values that are stored in variables res and adj. The first value is the value of the independent variable result (see Figure 1.1)
and the second value is a partial derivative $\left( \frac{\partial \text{product}}{\partial v} \right)$. Since the computation is very small, LogAD uses only one checkpoint to differentiate it.

octave:1> [res, adj] = gdiff("product", [1, 2, 3, 4], 1)
Checkpoints used: 1

res = 13824

adj =

41472 20736 13824 10368

Figure 1.4 Results of Running \textbf{product}

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 discusses underlying mathematics and Griewank's algorithm. Chapter 3 details the internal structure of the LogAD system and relationships between its components. Evaluation of LogAD is presented in Chapter 4. Finally, Chapter 5 draws some conclusions and sketches the directions of the future research.
2. AUTOMATIC DIFFERENTIATION METHODS: MATHEMATICAL BACKGROUND

This chapter gives a more in-depth overview of the mathematical methods behind the LogAD system. First, we define the computational framework.

Consider a composite differentiable function

\[ F : \mathbb{R}^k \rightarrow \mathbb{R}^d, \]  
(2.1)

with independent variables \([x_1, x_2, \ldots, x_k]\) and dependent variables \([y_1, y_2, \ldots, y_d]\). The Jacobian of \(F\) is the \(k \times d\) matrix of first-order partial derivatives and defined as:

\[ J_F = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \vdots \\ \frac{\partial y_d}{\partial x_k} \end{bmatrix}_{k \times d}. \]  
(2.2)

If \(d = 1\), then the gradient of \(F\) is the \(k\) element vector of first order partial derivatives:

\[ \nabla F = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \vdots \\ \frac{\partial y}{\partial x_k} \end{bmatrix}_k, \]  
(2.3)

where \(y = F(\bar{x})\).

The function \(F\) is assumed to be a sequence of \(n\) elementary differentiable transformations \(f_i\):

\[ s_{i+1} \leftarrow f_i(s_i), \]  
(2.4)

where \(s\) denotes the computational state of the function \(F\). The computation state includes dependent and independent variables, intermediate variables, and other information. Typically, each \(f_i\) is an elementary operation and hence \(f_i\) only depends on a few elements of the state \(s_i\). If each elementary transformation has a Jacobian \(J_i\), then, using the chain rule, the Jacobian of the function \(F\) is defined as:

\[ J_F = J_{f_1} \times J_{f_2} \times \ldots \times J_{f_n} \]  
(2.5)
For example, consider a function

\[ F(X) : y = x_1 \sin x_2. \]  

This function can be computed by applying the following sequence of elementary transformations:

\[ f_1 : t_1 = \sin x_2; \]  \hspace{1cm} (2.7)

\[ f_2 : y = x_1 \times t_1. \]  \hspace{1cm} (2.8)

These functions use four variables \( y, x_1, x_2, \) and \( t_1, \) hence each elementary operation can be viewed as a \( \mathbb{R}^4 \rightarrow \mathbb{R}^4 \) mapping. Each of the elementary transformations has a well-defined Jacobian:

For \( f_1 \):

\[
J_{f_1} = \begin{bmatrix}
y \\
x_1 \\
x_2 \\
t_1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & \cos x_2 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (2.9)

For \( f_2 \):

\[
J_{f_2} = \begin{bmatrix}
y \\
x_1 \\
x_2 \\
t_1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
t_1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
x_1 & 0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (2.10)

To compute the derivatives of interest these Jacobians are multiplied together to produce the Jacobian of the original Equation 2.6:

\[
J_F = \begin{bmatrix}
y \\
x_1 \\
x_2 \\
t_1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
t_1 & 1 & 0 & 0 \\
x_1 \cos x_2 & 0 & 1 & \cos x_2 \\
x_1 & 0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (2.11)
By substituting Equation 2.7, we get

\[
J_F = x_1 \begin{bmatrix}
1 & 0 & 0 & 0 \\
\sin x_2 & 1 & 0 & 0 \\
x_1 \cos x_2 & 0 & 1 & \cos x_2 \\
x_1 & 0 & 0 & 1
\end{bmatrix}
\]  \hspace{1cm} (2.12)

Given this Jacobian, evaluation of the gradient in question can be done in several ways. Two commonly used approaches are the forward mode and the reverse mode.

2.1 Forward Mode

The forward mode of automatic differentiation is the most straightforward way of obtaining derivatives. To extract the \(i\)th row of the function Jacobian, we can multiply a unit column vector times the function Jacobian. Since the function Jacobian is the product of the Jacobians of the elementary operations, this process is equivalent to:

\[
(l_i)^T \times J_F = (((l_i)^T \times J_{f_1}) \times J_{f_2}) \times \ldots \times J_{f_n}),
\]  \hspace{1cm} (2.13)

where \(l_i\) is the \(i\)th unit vector. Each matrix-vector multiplication can be done in \(O(1)\) steps because Jacobians of elementary transformations are very sparse (see matrices 2.9 and 2.10). Using the above example from this chapter, the 2nd row of the resulting Jacobian can be obtained in the following way:

\[
\left( \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \times J_{f_1} \right) \times J_{f_2} = \begin{bmatrix} t_1 & 1 & 0 & 0 \end{bmatrix}.
\]  \hspace{1cm} (2.14)

Substituting the right part of equation 2.7 in place of \(t_1\) the following vector is obtained (with a derivative in question underlined):

\[
\begin{bmatrix}
\sin x_2 & 1 & 0 & 0
\end{bmatrix}.
\]  \hspace{1cm} (2.15)

Forward mode is easy to program and does not require recording the computation. One forward pass can be done in roughly the same time as the original function is
computed. In order to compute the whole gradient, this approach requires one pass for each independent variable. Hence, this approach is inefficient for functions with many independent variables. However, the efficiency of the forward mode can be improved using sparse vector techniques as done in ADIFOR [AMB+94, BH91].

2.2 Reverse Mode

The full gradient is a column in the Jacobian, and thus can be computed by multiplying the Jacobian by an appropriate column unit vector:

\[ \nabla F = J_F \times \vec{l} = (J_{f_1} \times (J_{f_2} \times \ldots \times (J_{f_n} \times \vec{l}))), \tag{2.16} \]

where \( \vec{l} \) is a unit vector. Again, since Jacobians of elementary transformations are very sparse, each matrix-vector product can be computed in \( O(1) \) steps and the whole product can be computed right to left roughly in the same time as the original function.

This approach can calculate the whole gradient in one pass, and hence it is more efficient for functions with many independent variables and few dependent variables; however, some a priori knowledge is required in order to calculate (2.16). Specifically, since it uses the last Jacobian first, the computation must be somehow recorded and be available “in reverse.” The storage space needed for recording the computation is proportional to the running time. Therefore, the space complexity can become prohibitively large for big volumes of data and iterative routines [Gri92].

Using the example above, the whole gradient \( \nabla F \) can be extracted in one pass:

\[ \nabla F = F_{f_1} \times (F_{f_2} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} y \\ x_1 \\ x_2 \\ t_1 \end{bmatrix} \begin{bmatrix} 1 \\ t_1 \\ x_1 \cos x_2 \\ x_1 \end{bmatrix}). \tag{2.17} \]
Substituting equation 2.7 in equation 2.17, the following gradient is obtained (with derivatives in question underlined):

\[
\nabla F = \begin{bmatrix}
    y \\
    x_1 \\
    x_2 \\
    t_1
\end{bmatrix}
\begin{bmatrix}
    1 \\
    \sin x_2 \\
    \frac{x_1 \cos x_2}{x_1} \\
    x_1
\end{bmatrix}.
\]

(2.18)

Note that computing the gradient is actually a special case of a more general technique. By using an appropriate choice for the vector \( \vec{l} \), linear combinations of the columns of the Jacobian can be computed simultaneously using a single reverse pass. This approach can be used to compute the full Jacobian very quickly when the Jacobian is sparse [CV96].

2.3 Adjoint Equations

As mentioned above, each matrix-vector multiplication can be done in \( O(1) \) steps because the Jacobians of elementary transformations are very sparse. Noticing that only arguments of elementary functions change their values, a set of equations for “reversing” every elementary operation can be found. These equations calculate only the elements that change their values as a result of matrix-vector multiplications and are often referred to as the adjoint equations.

As an example consider the addition operator:

\[
x = y + z.
\]

(2.19)

It has the following well defined derivatives:

\[
\frac{\partial x}{\partial y} = 1,
\]

(2.20)

\[
\frac{\partial x}{\partial z} = 1,
\]

(2.21)
and the Jacobian

\[
\begin{array}{c}
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 0 \\
1 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{z}
\end{bmatrix}
\end{array}
\]

The adjoint vector is a column vector of the structure:

\[
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{z}
\end{bmatrix}
\]

Multiplication of the Jacobian by the adjoint vector yields the column vector:

\[
\begin{bmatrix}
1 & 0 & 0 \\
1 & 1 & 0 \\
1 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\dot{x} \\
\dot{y} \\
\dot{z}
\end{bmatrix}
= 
\begin{bmatrix}
\dot{x} + \dot{y} \\
\dot{x} + \dot{z}
\end{bmatrix}
\]

This gives the following adjoint equations:

\[
\dot{y} + = \dot{x},
\]

\[
\dot{z} + = \dot{x}.
\]

Since the adjoint equations 2.25 and 2.26 do not use the actual values of the variables, reversing the operation addition does not require any recorded information.

As another example consider the division operator:

\[
x = \frac{y}{z}.
\]

The derivatives are:

\[
\frac{\partial x}{\partial y} = \frac{1}{z},
\]

\[
\frac{\partial x}{\partial z} = -\frac{y}{z^2}.
\]

This gives the following Jacobian matrix:

\[
\begin{array}{c}
\begin{bmatrix}
1 & 0 & 0 \\
\frac{1}{z} & 1 & 0 \\
-\frac{y}{z^2} & 0 & 1
\end{bmatrix}
\end{array}
\]
Multiplying this Jacobian by the adjoint vector yields:

\[
\begin{bmatrix}
1 & 0 & 0 \\
\frac{1}{z} & 1 & 0 \\
-\frac{y}{z^2} & 0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
\hat{x} \\
\hat{y} \\
\hat{z}
\end{bmatrix}
= \begin{bmatrix}
\hat{x} \\
\hat{y} + \frac{\hat{x}}{z} \\
\hat{z} - \frac{y\hat{x}}{z^2}
\end{bmatrix}.
\]  

(2.31)

The above vector gives the following adjoint equations:

\[
\hat{y} = \frac{\hat{x}}{z},
\]

(2.32)

\[
\hat{z} = \frac{y\hat{x}}{z^2}.
\]

(2.33)

Notice that the equations 2.32 and 2.33 use the values of \(y\) and \(z\) that are obtained during the forward pass through the computation.

To clarify the details, the gradient of the following function will be computed:

\[
F : y = x_1 \sin x_2.
\]

(2.34)

\(F\) can be calculated by applying the following sequence of elementary transformations:

\[
f_1 : y = \sin x_2;
\]

(2.35)

\[
f_2 : y = x_1 \times y.
\]

(2.36)

Let

\[
x_1 = 10,
\]

(2.37)

\[
x_2 = 2.
\]

(2.38)

Then, running this computation forward yields:

1. \(y = \sin x_2 = \sin 2 = .9092\)

2. \(y = x_1 t_1 = 10 \times .9092 = 9.092\)
Table 2.1 Computing Gradient of $F$ in Reverse Mode

<table>
<thead>
<tr>
<th>Initial State</th>
<th>State 2</th>
<th>State 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>9.092</td>
<td>.9092</td>
</tr>
<tr>
<td>$x_1$</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>$x_2$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$\hat{y}$</td>
<td>1</td>
<td>$\hat{y}' + x_1 \hat{y} = 10$</td>
</tr>
<tr>
<td>$\hat{x}_1$</td>
<td>0</td>
<td>$\hat{x}_1' + y \hat{y} = .9092$</td>
</tr>
<tr>
<td>$\hat{x}_2$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

During this forward sweep all the old values of the variables are recorded to be restored during the reverse sweep. In our example, the value of $y$ is redefined in the second step, and hence must be stored. Before each operation in the reverse sweep, the "old" value of operation resulting variable must be restored and its adjoint set to 0; however for the adjoint of the result of elementary operation, the current value must be used (denoted by '). Table 2.1 illustrates all 3 step of the process. Initially, all adjoints are set to 0, except the adjoint of the dependent variable, which is set to 1. Then, adjoints for every elementary operation are calculated in reverse order.

The resulting gradient with respect to $y$ is:

$$ x_1 \begin{bmatrix} .9092 \\ -4.162 \end{bmatrix} $$  \hspace{1cm} (2.39)

The same results are given by direct symbolic differentiation:

$$ \frac{\partial y}{\partial x_1} = \sin x_2 = \sin 2 = .9092, $$  \hspace{1cm} (2.40)

$$ \frac{\partial y}{\partial x_2} = x_1 \cos x_2 = 10 \cos 2 = -4.162 $$  \hspace{1cm} (2.41)
2.4 Reverse Mode with Checkpointing: Griewank’s Algorithm

As mentioned earlier, the space complexity of the general reverse mode of automatic differentiation is proportional to the time complexity of the original algorithm. This dependency can be prohibitive for large iterative computations. In [Gri92], Griewank proposed a solution, which “destroys the proportionality between the temporal complexity of the original evaluation program and the spatial complexity of the reverse mode” by introducing checkpointing in the reverse mode. He observed that the reverse mode can be implemented by checkpointing the computation at designated points, recording only the last portion of the computation, backpropagating derivatives on this portion, and then restarting the computation from the most recent checkpoint, i.e., by increasing the time by a logarithmic factor the gradient can be computed with much less space.

Griewank considers two partitioning algorithms bisection and binomial. He showed that the bisection partitioning strategy yields increases in both time and space complexity that are equal to the logarithm of the time complexity of the original problem. This factor is reduced by 2 if one uses binomial partitioning, which is optimal.

Figure 2.1 shows a simplified diagram of Griewank’s algorithm that uses the bisection strategy. The vertical bars depict the computation steps that are considered in an iteration. The very first iteration involves the whole computation. The shaded bars designate the part of the computation that are recorded and reversed. The thick horizontal lines show the points where the checkpoints are created. Each successive checkpoint is created in the half-way \( p_i \) from the current checkpoint \( p_{i-1} \) to the “end” of the computation \( p_n \):

\[
p_i = p_{i-1} + \frac{p_n - p_{i-1}}{2}.
\] (2.42)

Every iteration begins with the most recent checkpoint except the very first iteration. As the picture shows, the adjoints are propagated from the “end” to the “beginning” of the computation.
Although the bisection strategy is not optimal, it is used in LogAD for its simplicity. Checkpoint scheduling according to the bisection algorithm is done by the checkpoint manager code. Since LogAD uses run time operation resolution and a recording size of an elementary operation is not known at compile time, the checkpoint manager code needs to be inserted in front of each elementary operation (which may degrade performance in case of scalar operations). The pseudo-code of the manager code is given in Figure 2.2.
if \( p[i] \geq \frac{p[i-1] + (p[n] - p[i-1])}{2} \) then
\[\text{if already recording then} \]
\[\text{start reverse pass}\]
\[\text{else}\]
\[\text{record the size of the current computation in data space}\]
\[\text{of previous checkpoint}\]
\[\text{if } p[i] - p[i-1] \leq \text{bound then}\]
\[\text{start recording}\]
\[\text{else}\]
\[\text{make checkpoint}\]
endif
endif
endif

Figure 2.2 Pseudo-code of LogAD Checkpoint Manager

As mentioned earlier, Griewank's algorithm was first implemented in ADOL-C [GJU96] using piping and forking facilities of UNIX and employed manual placing of checkpoints. Checkpointing was implemented as a tree of processes. When a process determines that the checkpoint is needed it forks out a child, which inherits all data from its parent process. Meanwhile the parent process freezes its execution until the child completes its part of the job. When the child is finished with its part of the computation the adjoint values are sent back to the parent through a UNIX pipe. Then the parent continues its execution until it forks out another child process, or its part of the computation has been completely recorded and reversed.

Although using \texttt{fork()} simplifies the task of programming, it can be extremely expensive on some systems that do not employ "copy-on-write" strategy. The facts that
the OS limits the number of processes that can run simultaneously can reduce the availability of this method for large problems on some machines. Furthermore, this implementation cannot be ported to the operating systems that do not implement fork-like facilities. Griewank mentions that implementing of reverse automatic differentiation with checkpointing at the compiler level would increase user convenience and computational efficiency.
3. LOGAD ARCHITECTURE

This chapter gives an overview of LogAD v.2, explains its architecture, and provides some examples.

LogAD functional structure is shown in Figure 3.1. The library `gdiff.oct` is “glue” between Octave and LogAD, and is compiled in the form suitable for dynamic linking by Octave. The sole function of this library is to load the LogAD shared library `logad.so` with flag `RTLD_GLOBAL` making all its objects global (this is required by `.lso` files, since they use functions from `logad.so`). The shared library `logad.so` provides the LogAD execution environment. It contains implementations of “variable” classes, tape, checkpointing algorithm, etc. An `.lso` file that is usually created by the LogAD linker contains only a function that is being differentiated, and is used by the LogAD shared library.

![Figure 3.1 LogAD Structure](image-url)
As was mentioned above, our implementation accepts algorithms written in a subset of the Octave language, which is compatible with the language used in Matlab. This subset includes the following language constructs:

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

Currently, LogAD v.2 implements two Octave data types: real scalar and real matrix. The other data types, such as complex scalar or complex matrix, can be added without rewriting any of the existing code.

To reduce the interpretation overhead LogAD translates Octave source code into equivalent fragment of C++ code before executing it. All fragments of C++ code generated from relevant Octave functions are placed in one C++ function, and this function is compiled by `g++` to produce a shared library `function.lso` (see Figure 1.2). Since our system does not use operating system assistance to snapshot the computation, checkpoints are produced by the compiler-generated code called the `checkpoint manager`. This code is responsible for calculating computational steps, creating snapshots according to the bisection algorithm described in Chapter 2, and initiating the reversing process. To be able to resume the computation from the point where it was interrupted, LogAD uses a GNU extension of the C language called “labels as values”. As seen in Figure 3.2 the address of a label defined in the current function can be obtained by applying the unary operator `&&` and assigning this value to a variable of type `void*`. Later, this variable can be used in the computed `goto` statement to jump to the address it points to. When the checkpoint manager
void func () {
    void *addr = &&my_label;  // assign address of my_label to addr
    ...
    goto *addr;               // jump to my_label
    ...
    my_label:
    ...
}

Figure 3.2 GNU Extension: Labels as Values

determines that it is time to make a snapshot of the current computation state, it
stores all necessary information as well as the address of the next label (each fragment
of the manager code contains a label). Later, when this checkpoint is restored, a
computed goto statement is used to jump to the stored address, and the computation
resumes from the point where it was interrupted.

3.1 LogAD Compiler

The LogAD compiler logadc translates a function, written in the Octave language
into a fragment of C++ code. This code almost entirely consists of function calls
and cannot be well optimized by the compiler (however, some functions calls can be
replaced by the inline code using the C keyword inline). Although the generated
C++ code mimics the behavior of assembler code, generating actual assembler code
would make the implementation machine-dependent, which is not desirable.

At run time each instance of an Octave function (i.e., C++ code that simulates
an Octave function) is given its stack frame. All Octave variables (i.e., instances
of C++ classes that simulate Octave variables) are addressed by their offset in the current stack frame and not by their names. However, each variable object stores the name of the corresponding Octave variable, so that meaningful error messages can be produced if the program tries to use an uninitialized variable or perform other violations. In front of each code fragment that computes an Octave function, the compiler produces the offset table, which defines an offset of each variable used in the function with respect to the current frame pointer.

A typical Octave program consists of one or several functions. Each Octave function must reside in a separate .m file and a base name of this file must be the same as the name of the function because when the LogAD linker searches for a function it scans the list of the specified directories for files ending in .m that have the same base name as the sought function.

The output from the LogAD compiler is two files:

- *function_name.logad*: contains the corresponding fragment of C code;

- *function_name.logad.d*: contains dependency information — names of the external functions that are used by the compiled function.

In the linking stage all functions on which the compiled function is dependent are included in the final C++ function.

3.1.1 Expressions and Index Expressions

Expressions are the base building blocks of any Octave program. Each expression has a value that can be assigned to a variable, passed to a function, etc. An expression itself can serve as an Octave statement and includes variables, constants, function calls, matrix elements and combinations of these with various operators. An "index expression" is a powerful feature in Octave and was chosen to be implemented in LogAD. It allows referencing or extracting selected elements of a matrix, vector, or
gindex (_vt_0, _v_c, 1, _v_i);
f_sin (_vt_1, _v_c);
bi_minus (_vt_0, _vt_0, _vt_1);
bi_mult (_vt_0, _v_b, _vt_0);
iassign(_v_t, _vt_0, 1);

Figure 3.3 Compiled Expression Octave Expression

scalar. Indices may be matrices, vectors, scalars or the special symbol ':', which selects entire rows or columns [BB95].

For the purpose of automatic differentiation, each complex Octave expression is split into a sequence of elementary functions. This splitting may create a number of temporary variables to hold intermediate results.

Consider the example of compiling the expression:

\[ t(i) = b \times (c(i) - \sin(c)) \]

For this expression the compiler produces the code given in Figure 3.3 (the checkpointing manager code is excluded for readability). The variable names are mangled in the following way: the names of user-defined variables are prepended with _v_ prefix, and the names of temporary variables start with _vt_ followed by their numbers. The expression is divided into five elementary operations and two temporary variables are created. To minimize the number of variables, the compiler tries to reuse a temporary variable as soon as it sees that its value is not used later in the code.

3.1.2 Function Calls

In the compiled Octave program, all Octave function calls are simulated by goto statements and labels. All compiled Octave functions that are needed by the program reside in one C++ function the_diff_func.cc and each of the compiled Octave functions
starts with a label constructed from the function name with prepended \texttt{.f}. Each Octave function call is compiled to a \texttt{goto} statement (see example in Figure 3.4). Prior to executing the \texttt{goto} statement and transferring the execution flow to another “function”, the current “function” must place the arguments in the special argument buffer and push the return address to the return address stack (again, the GNU extension is used). The calling function can place any number of arguments in the buffer, limited only by the size of this buffer. After the called function returns, the calling function can extract the return values from the same argument buffer.

The Octave syntax for a function call is the same as for an index expression: \texttt{ID(...)}. LogAD uses the following rules to resolve this ambiguity at compile time (Octave uses a similar approach):

- if an expression of type \texttt{ID(...)} is followed by the equal sign “=" then it is an index expression;

- if the identifier \texttt{ID} exists in the symbol table (LogAD compiler symbol table contains only variable names) at the moment when the ambiguity has to be resolved then it is an index expression;

- otherwise, it is a function call.

For example, consider the Octave function call

\[
\text{[result, d] = myfunc(a, b, c).}
\]

After the compiling stage, it is translated into the fragment of C++ code shown in Figure 3.4 (the checkpoint manager code excluded for readability).

3.1.3 \texttt{if-elseif-else} statements

Expressions in the \texttt{if} and \texttt{elseif} clauses can be any Octave expression, including function calls. Since a function call is compiled into a \texttt{goto} C statement, the whole
push_args (3, _v_a, _v_b, _v_c); // push function arguments
push_retaddr (&&_r_sample_0);     // push return address
goto _f_myfunc;                  // 'function call'

_r_sample_0:
pop_rets(2, _v_result, _v_d);    // pop return values

Figure 3.4 Compiled Function Call

if (a + b < width)
...
elseif (a + c == height)
...
else
...
endif

Figure 3.5 Octave if-elseif-else Statement

compiled expression cannot be placed in the if clause of the C if statement. Therefore, the if and elseif clause expressions are calculated in a separate block and then their results are tested in the if clause.

Example of the Octave code in Figure 3.5 produces the fragment of C code shown in Figure 3.6.

3.1.4 while loop

For the same reasons as for the if statement, the whole compiled expression cannot be placed in the while clause of the C while statement. At the same time this
bi_plus (_vt_0, _v_a, _v_b);
cmp_less (_vt_0, _vt_0, _v_width);
if (is_true(_vt_0)) {
    ...
}
else {
    bi_plus (_vt_0, _v_a, _v_c);
cmp_eq (_vt_0, _vt_0, _v_height);
if (is_true(_vt_0)) {
    ...
}
else {
    ...
}
}

Figure 3.6 Compiled if-elseif-else Statement

eexpression cannot be placed in front of the while loop, because it must be recalculated at each iteration. For these reasons, testing the expression is done by the C if statement inside the infinite loop.

The Octave while expression and the compiled expression are shown in Figures 3.7 and 3.8, respectively.

3.1.5 for loop

The assignment expression in the Octave for statement works differently than the normal Octave assignment expression. Instead of assigning the whole expression to a
while (a + b < width)
    ...
endwhile

Figure 3.7 Octave while Statement

while (1) {
    bi_plus (_vt_0, _v_a, _v_b);
    cmp_less (_vt_0, _vt_0, _v_width);
    if (!is_true (_vt_0)) break;
    ...
}

Figure 3.8 Compiled while Statement

loop variable, each column of the result is assigned in turn. The number of iterations
through the loop body is equal to the number of columns in the result of the for
clause expression at the beginning of the loop.

Since the execution of the for statement can be interrupted by a checkpoint, the
iteration number must be stored as a part of the computation state. When the loop
starts execution, the auxiliary loop variable is initialized to 0 (we start with the first
column of the for clause expression) and pushed to the system stack _for_loop_cnt.
When exiting the for loop the loop variable is popped from the stack.

The Octave for expression and the compiled expression are shown in Figures 3.9
and 3.10, respectively.
for i = 1:a
...
endfor

Figure 3.9 Octave for Statement

set_scalar (_vt_0, 1.00000000000000000000);
set_range (_vt_0, get_double(_vt_0), get_double(_v_a));
_for_loop_cnt->push_back (1);
for (; for_element (_v_i, _vt_0, _for_loop_cnt->back());
 _for_loop_cnt->back()++) {
 ...}
 _for_loop_cnt->pop_back ();

Figure 3.10 Compiled for Statement

3.2 LogAD Linker

The LogAD linker logadl is a utility that generates a special shared library function.lso, which contains a function that will be differentiated. This library is suitable for dynamic linking by LogAD. The linker performs the following actions:

1. Finds and compiles (if needed) the Octave function that is being differentiated and all functions in its dependency tree. The user can specify a list of directories in which to search for functions on command line or in the environment variable LOGAD_LOADPATH. A function is recompiled only if its time stamp is greater than a time stamp of the corresponding .logad file. When compiling source files, the LogAD compiler generates dependency information that is written to
a function_name.logad.d file. The linker uses this dependency information to build a tree of dependencies.

2. Puts all the compiled pieces of C++ code, as well as necessary library code, into one C++ function the_diff_func in the file the_diff_func.cc;

3. compiles the_diff_func.cc with g++ to produce a shared library function.lso.

To avoid hardcoding options and command line format for the LogAD compiler and g++, the linker uses two shell scripts logad_logadc and logad_cc. The first script performs compilation of an Octave program and accepts an output directory name and a source file name as its arguments. The second script compiles the_diff_func.cc with g++. It has no arguments. These scripts can be freely modified by the user.

3.3 LogAD Shared Library

The LogAD shared library logad.so provides the execution environment for the compiled Octave program.

3.3.1 Execution Environment

The core of the LogAD execution environment is two execution stacks; one for the forward pass and one for the reverse pass. The structure of these stacks is shown in Figure 3.11. The stacks contain pointers to the variable objects but not the actual objects; hence all entries are of the same size. The pointers are of type VBase*, where VBase is the base class for all “variable” classes and corresponds to the uninitialized variable in Octave. The stacks have the frame pointer (one for two stacks), which points to the top of the current function frame. All operations are performed using
offsets from the frame pointer, as opposed to actual pointers to the objects. Each successive function places its stack frame above the current stack frame in the execution stack.

Arguments and return values of compiled Octave functions are passed through the argument buffer because a calling function does not know how many arguments are expected and how many values are returned by the function that is being called. There are four functions in the class Checkpoint that manage function calls and stack frame operations:

- **add_frame** - creates a new stack frame with a specified number of dependent, independent, and temporary variables; initializes independent variables to the values in the argument buffer;
• `del_frame` - places dependent variables in the argument stack; deletes the current stack frame;

• `push_args` - places specified arguments in the argument buffer;

• `pop_args` - initializes specified variables to the values in the argument buffer.

When a function starts its execution, all of its dependent and temporary variables are set to a `VBase` object, which corresponds to a uninitialized variable in Octave. If such an uninitialized variable is used on the right hand size of the assignment or indexed assignment operator, an error message is displayed.

All operations on variables are resolved at run time. Unary operations are resolved via the C++ virtual function mechanism. To resolve a binary operation several approaches can be taken:

• double virtual redirection;

• lookup tables (used in Octave);

• `switch` structures.

Although with lookup tables new operations can be added without touching the existing code, for performance considerations our implementation uses `switch` structures to resolve binary operations.

3.3.2 Checkpointing

As was mentioned above, LogAD uses the compiler-generated code to checkpoint the computation. The computational state includes the following information:

• a snapshot of the execution stack;

• a snapshot of the return address stack;

• the frame pointer;
• the for-loop counter stack,

and checkpoint parameters are:

• a computational fragment size;

• a start address.

Since checkpoints are accessed in LIFO manner, LogAD uses a stack structure to store them.

Checkpoints are scheduled by the checkpoint manager code, which is inserted in front of every operation. It should be noted that the computation can be checkpointed more than one time at the same point. This situation usually happens at the beginning of a large computation segment because the segment cannot be fully recorded. When such a checkpoint is restored, LogAD must immediately create the same checkpoint so that the computation can be restarted to record the remaining portion. This may occur several times.

3.3.3 Tape

In the recording session of the forward sweep the computation is written out to a "tape." As in ADOL-C, every computational step may record the following information:

• an old value of the resulting variable;

• an operation code;

• argument offsets;

• other information, which varies between operations.

This information is used later in the reverse sweep.

One of the problems that arose during the implemention of the tape class was data alignment because some machines have strict alignment requirements. For example,
Sun SPARC workstations demand that double-precision floating point numbers be aligned at 8-byte boundaries. Therefore, writing data of different types on one tape is problematic. Several methods can be used to overcome the problem; e.g., padding or using several buffers, one for each data type. However, these methods have obvious shortcomings. The method that was implemented in LogAD is the double-ended buffer, because the two most commonly used data types are byte and double. Byte values are added from the left side and double values are added from the right side of the buffer (see Figure 3.12). When there is a need to store a value of another type, it is added byte by byte from the left side of the buffer.

![Figure 3.12 LogAD Tape Structure](image)

3.4 Memory Pooling

Objects of some LogAD classes are created and destroyed very frequently. For example, an elementary operation usually creates a new object for storing the result and deletes the old variable. System’s performance may degrade if global C++ operators `new` and `delete` are called for every object that is created or destroyed. For these reasons the system implements memory pooling technique for some classes (namely, all “variable” classes and the Checkpoint class).
4. LOGAD RESULTS

The chapter presents results of running derivative codes generated by LogAD and compares them to derivative codes generated by ADOL-C for similar programs. The parameters of interest are the running time, the space required to perform calculations, and a number of checkpoints.

All tests were run on a 333 MHz Intel Celeron processor with 96M of RAM. The operating system was Red Hat Linux 6.1. Two algorithms were chosen to test the system: the Helmholtz energy function and chain matrix multiplications. These algorithms were differentiated using LogAD and ADOL-C and the results were plotted and compared.

4.1 Chain Matrix Multiplications

Chain matrix multiplications are considered in this example:

\[ f(X) = \text{sum}(\prod_{i=1}^{m} X), \]  

(4.1)

where \( X \) is the square \( n \times n \) matrix and \( \text{sum}(X) \) calculates the sum of all elements of the matrix \( X \). Each matrix-matrix multiplication requires \( n^2(2n - 1) \) operations, and the \( \text{sum}() \) function requires \( n^2 - 1 \) operations. The \( 8 \times 8 \) identity matrix was used for the test. The numbers of multiplications \( m \) and the corresponding computation sizes are given in Table 4.1.

Figure 4.1 shows the time results of ADOL-C and LogAD codes. LogAD was tested with two tape sizes 64K and 10M. Since ADOL-C performs a lot of disk I/O operations, total running time as well as running time in the user mode were measured. The plot shows that LogAD was faster than ADOL-C for all input sizes, even though
Table 4.1 Number of Multiplications and Computation Sizes for Matrix Multiplications

<table>
<thead>
<tr>
<th>$m$</th>
<th>Computation Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>960,063</td>
</tr>
<tr>
<td>5000</td>
<td>4,000,063</td>
</tr>
<tr>
<td>10000</td>
<td>9,600,063</td>
</tr>
<tr>
<td>50000</td>
<td>48,000,063</td>
</tr>
<tr>
<td>100000</td>
<td>96,000,063</td>
</tr>
</tbody>
</table>

Figure 4.1 LogAD and ADOL-C Time Results for Matrix Multiplications
LogAD uses run time operation resolution. One possible reason for this is that LogAD performs and records a matrix-matrix multiplication as one atomic operation, while ADOL-C records each operation between matrix elements.

The space results for ADOL-C are shown in Figure 4.2. It can be seen that the ADOL-C tape grew in proportion to the computation size. 100,000 multiplications could not be finished by ADOL-C code because it ran out of space on a 500M disk partition.

Figure 4.3 shows the number of checkpoints that LogAD used to perform the matrix multiplications. For the 10M tape, LogAD started checkpointing at 50,000 multiplications and used 8 checkpoints for 100,000 multiplications. When the tape size is limited to 64K, LogAD uses checkpointing scheme heavily. It already starts
checkpointing at 1,000 multiplications and uses 2,048 checkpoints for 100,000 multiplications. Notice that the number of checkpoints actually created by LogAD can be greater than the number of logical checkpoints, since the computation sometimes is checkpointed more than one time in the same point (see Section 3.3.2).

This test shows that LogAD matrix-matrix operations are quite efficient. Even though LogAD has the interpretation overhead, it outperforms ADOL-C for applications where matrix-matrix operations dominate.
4.2 Helmholtz Energy Function

The Helmholtz energy function of a mixed fluid at the absolute temperature $T$ in a unit volume [Gri89] was used as another test for the LogAD system:

$$f(x) = RT \sum_{i=1}^{n} x_i \log \frac{x_i}{1 - b^T x} - \frac{x^T Ax}{\sqrt{b^T x}} \log \frac{1 + (1 + \sqrt{2})b^T x}{1 + (1 - \sqrt{2})b^T x},$$

(4.2)

where $R$ is the universal gas constant and

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

(4.3)

This function is used in the oil industry for extracting fractions of certain viscosity from oil.

This function was chosen because it can use an arbitrary sized input vector. The calculations were performed for 1,000 - 6,000 independent variables. The computation sizes are given in table 4.2.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Computation Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>500,501</td>
</tr>
<tr>
<td>2000</td>
<td>2,001,001</td>
</tr>
<tr>
<td>3000</td>
<td>4,501,501</td>
</tr>
<tr>
<td>4000</td>
<td>8,002,001</td>
</tr>
<tr>
<td>5000</td>
<td>12,502,501</td>
</tr>
<tr>
<td>6000</td>
<td>18,003,001</td>
</tr>
</tbody>
</table>

The ADOL-C program was implemented using the ADOL-C active vector class 
`adoubleu` (included in the ADOL-C package). The LogAD program was implemented
using mostly scalar operations and tested with tape sizes 100K and 10M. The timing results in Figure 4.4 show that ADOL-C outperforms LogAD for all input sizes because LogAD run time operation resolution introduces considerable overhead for scalar operations. The amount of space needed by ADOL-C to complete the calculations is shown in Figure 4.5. For 6000 independent variables ADOL-C could not finish the calculations because it ran out of space on 500M disk partition.

The numbers of checkpoints for LogAD are shown in Figure 4.6. It can be seen that LogAD used checkpointing heavily for 100K tape – it produced 32768 checkpoints for 6000 variables! The curves are not exactly logarithmic because in some cases the computation cannot be bisected where it is required by the theory, especially, when a large matrix needs to be saved.
For Helmholtz energy we also experimented with different tape sizes for different number of independent variables. The results are in Figures 4.7 and 4.8. It can be seen that these dependencies are logarithmic (Tape Size axis is logarithmic). The computation time nearly doubles when the tape size is reduced by a factor of 2. The number of produced checkpoints greatly depends on tape size. For example, for 2000 variables with 100K tape the calculations can be completed in 130 seconds using 2048 checkpoints. When using bigger tape of 50M the calculations run two times faster using only 4 checkpoint.

This example shows that LogAD scalar operations are relatively inefficient because of considerable interpretation overhead. Also the way in which the Octave for loop is implemented makes it inefficient compared to the C for loop.
Figure 4.6 Number of Checkpoints for Helmholtz Energy
Figure 4.7  Tape Size - Time Dependency for Helmholtz Energy
Figure 4.8 *Tape Size - Number of Checkpoint* Dependency for Helmholtz Energy
5. CONCLUSIONS AND FUTURE RESEARCH

In this thesis, we have presented an automatic differentiation tool that can find a function's gradient using much smaller space than required by the general purpose reverse mode. In fact, the tool can differentiate a function containing 1000 matrix-matrix multiplications using a tape of size 1300 bytes, whereas ADOL-C requires a tape of size 10 megabytes to complete the same task.

The Octave language, which is used as an input language in LogAD, is convenient for mathematical computations and has a number of useful high-level facilities. At the same time it introduces interpretation overhead, which can be significant in some cases. For example, run time operation resolution can slow down the computations if the algorithm uses a lot of scalar operations. Another example are function calls. In Octave a calling function does not know the number and types of arguments expected by the called function, that is why function arguments need to be passed through a separate buffer. This way of a function call resolution can slow down the computations significantly, especially for recursive algorithms. The Octave documentation recommends replacing recursive algorithms with corresponding iterative algorithms for efficiency reasons.

At the present moment, only a subset of Octave language is implemented: expressions, index expressions, function calls, if statements, for and while loops. The features that are not implemented are: switch statements, I/O functions, many built in mathematical functions, etc. The future versions should enhance the LogAD compiler and the execution environment to accommodate all features of Octave language.

As of now, the LogAD compiler that converts Octave code into a fragment of C++ code does not perform any optimization. The future versions should do at
least basic optimization, for example, common expression elimination and moving constant expressions out of loops. These two techniques can significantly improve the performance for iterative computations.

Reference counts and copy-on-write strategies can also improve the performance of some LogAD objects. As of now, when executing a function call, LogAD makes copies of function arguments because they may change their values during the function execution, while with reference counts an argument may be cloned only when its value is about to change. This technique may improve performance and save the tape space when large matrices are passed as function arguments.

The current version of LogAD uses the bisection algorithm to checkpoint the computation. While the use of binomial checkpointing may introduce some overhead for more careful placement of checkpoints, the overall performance of LogAD generated codes should improve.
BIBLIOGRAPHY


[MO96a] Andrew Mauer-Oats. Obstacles to implementing checkpointing in ADOL-C. A part of Checkpoint 1 package, December 1996.


A. LOGAD CONFIGURATION AND COMMAND LINE ARGUMENTS

A.1 Configuration

LogAD run time parameters are set through Octave variables:

- `LOGAD_TAPE_SIZE` (in bytes): used to set LogAD tape size;

- `LOGAD_STACK_SIZE` (in slots): used to set the size of LogAD execution stack (notice that LogAD creates two stacks of this size – for forward and reverse computations);

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

A.2 LogAD Compiler

Usage: `logadc [options] <source>`

<table>
<thead>
<tr>
<th>short</th>
<th>long</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>----</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>-h</td>
<td>--help</td>
<td>display help screen</td>
</tr>
<tr>
<td>-o &lt;file&gt;</td>
<td>--output &lt;file&gt;</td>
<td>place output in file &lt;file&gt;</td>
</tr>
<tr>
<td>-v</td>
<td>--version</td>
<td>display version</td>
</tr>
</tbody>
</table>
### A.3 LogAD Linker

**Usage:** `logadl [options] <function name>`

**options:**

<table>
<thead>
<tr>
<th>short</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----</td>
<td>--------------------------</td>
</tr>
<tr>
<td>-h</td>
<td>--help</td>
</tr>
<tr>
<td>-o &lt;file&gt;</td>
<td>--output &lt;file&gt;</td>
</tr>
<tr>
<td>-k</td>
<td>--continue</td>
</tr>
<tr>
<td>-s &lt;dir&gt;</td>
<td>--search-dir &lt;dir&gt;</td>
</tr>
<tr>
<td>-v</td>
<td>--version</td>
</tr>
</tbody>
</table>

- **-h** or **--help**: display help
- **-o <file>** or **--output <file>**: place output in file `<file>`
- **-k** or **--continue**: continue as much as possible after the error
- **-s <dir>** or **--search-dir <dir>**: specify search directory
- **-v** or **--version**: display version
B. LOGAD INPUT LANGUAGE GRAMMAR

rule 1  function -> func_header statement_list end_function
rule 2  func_header -> FUNCTION return_list ID arg_list
rule 3  end_function -> END
rule 4  end_function -> ENDFUNCTION
rule 5  return_list -> ID '='
rule 6  return_list -> '][' opt_id_list ']'] '='
rule 7  arg_list -> '(' opt_id_list ')' 
rule 8  statement_list -> /* empty */
rule 9  statement_list -> statement_list statement
rule 10 statement -> assignment ';'
rule 11 statement -> for_loop opt_semi
rule 12 statement -> while_loop opt_semi
rule 13 statement -> full_if_stat opt_semi
rule 14 statement -> command ';'
rule 15 assignment -> ID ']=' expr
rule 16 assignment -> ID '(' opt_expr_list ')') '=' expr
rule 17 assignment -> '][' opt_id_list ']'] '=' ID '(' opt_expr_list '
rule 18 expr -> NUM
rule 19 expr -> ID
rule 20 expr -> ID '(' opt_expr_list ')
rule 21 expr -> ID '=' expr
rule 22 expr -> ID '(' opt_expr_list ')') '=' expr
rule 23 expr -> colon_expr
rule 24 expr -> ' ( expr )'
rule 25 expr -> '-' expr
rule 26 expr -> '+' expr
rule 27 expr -> NOT expr
rule 28 expr -> expr TRANSPOSE
rule 29 expr -> expr CONJ_TRANSPOSE
rule 30 expr -> expr '+' expr
rule 31 expr -> expr '-' expr
rule 32 expr -> expr EPLUS expr
rule 33 expr -> expr EMINUS expr
rule 34 expr -> expr '*' expr
rule 35 expr -> expr '/' expr
rule 36 expr -> expr EMULT expr
rule 37 expr -> expr EDIV expr
rule 38 expr -> expr '<' expr
rule 39 expr -> expr LESSE expr
rule 40 expr -> expr '>' expr
rule 41 expr -> expr GRE expr
rule 42 expr -> expr EQ expr
rule 43 expr -> expr NEQ expr
rule 44 expr -> expr '&' expr
rule 45 expr -> expr '|' expr
rule 46 expr -> expr EAND expr
rule 47 expr -> expr EOR expr
rule 48 expr -> expr EPOW expr
rule 49 colon_expr -> expr ':.' expr
rule 50 colon_expr -> colon_expr ':.' expr
rule 51  command -> BREAK
rule 52  command -> CONTINUE
rule 53  command -> RETURN
rule 54  for_loop -> FOR for_cond statement_list endfor
rule 55  for_cond -> ID '=' expr
rule 56  endfor -> END
rule 57  endfor -> ENDFOR
rule 58  while_loop -> begin_while '(' while_expr ')' statement_list endwhile
rule 59  begin_while -> WHILE
rule 60  while_expr -> expr
rule 61  endwhile -> END
rule 62  endwhile -> ENDFOR
rule 63  full_if_stat -> if_stat opt_elseifs opt_else endif
rule 64  if_stat -> IF if_expr statement_list
rule 65  if_expr -> '(': expr ')' 
rule 66  elseif_clause -> elseif_begin if_expr statement_list
rule 67  elseif_begin -> ELSEIF
rule 68  elseifs -> elseif_clause
rule 69  elseifs -> elseifs elseif_clause
rule 70  opt_elseifs -> /* empty */
rule 71  opt_elseifs -> elseifs
rule 72  else_clause -> else_begin statement_list
rule 73  else_begin -> ELSE
rule 74  opt_else -> /* empty */
rule 75  opt_else -> opt_else else_clause
rule 76  endif -> END
rule 77  endif -> ENDIF
rule 78  colon_or_expr -> ':'
rule 79  colon_or_expr -> expr
rule 80  expr_list -> colon_or_expr
rule 81  expr_list -> expr_list ',' colon_or_expr
rule 82  opt_expr_list -> /* empty */
rule 83  opt_expr_list -> expr_list
rule 84  id_list -> ID
rule 85  id_list -> id_list ',' ID
rule 86  opt_id_list -> /* empty */
rule 87  opt_id_list -> id_list
rule 88  opt_semi -> /* empty */
rule 89  opt_semi -> ';';