CHECKPOINTING WITHOUT OPERATING SYSTEM INTERVENTION:
IMPLEMENTING GRIEWANK'S ALGORITHM

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

Richard Heller
August, 1998
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Automatic Differentiation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Operating System Issues</td>
<td>3</td>
</tr>
<tr>
<td>1.3 LogAD Compiler</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis Organization</td>
<td>6</td>
</tr>
<tr>
<td>2. MATHEMATICAL BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Example</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Automatic Differentiation</td>
<td>10</td>
</tr>
<tr>
<td>2.2.1 Forward Mode</td>
<td>10</td>
</tr>
<tr>
<td>2.2.2 Reverse Mode</td>
<td>11</td>
</tr>
<tr>
<td>3. GRIEWANK'S ALGORITHM</td>
<td>14</td>
</tr>
<tr>
<td>3.1 Using Griewank's Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Results of Griewank's Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>4. LOGAD COMPILER</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Architecture</td>
<td>18</td>
</tr>
<tr>
<td>4.1.1 Stacks</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2 C Language Constructs</td>
<td>20</td>
</tr>
<tr>
<td>4.1.3 LogAD Language Constructs</td>
<td>21</td>
</tr>
<tr>
<td>4.2 First Preprocessor</td>
<td>22</td>
</tr>
<tr>
<td>4.3 Second Preprocessor</td>
<td>24</td>
</tr>
</tbody>
</table>
### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1 Reversing the Code</td>
<td>24</td>
</tr>
<tr>
<td>4.3.2 The Monitor</td>
<td>28</td>
</tr>
<tr>
<td>4.4 Sample Code Trace</td>
<td>31</td>
</tr>
<tr>
<td>5. LOGAD COMPILER RESULTS</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Determinant Example</td>
<td>35</td>
</tr>
<tr>
<td>5.2 Helmholtz Energy Function</td>
<td>40</td>
</tr>
<tr>
<td>6. CONCLUSIONS AND FUTURE RESEARCH</td>
<td>49</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>51</td>
</tr>
<tr>
<td>APPENDIX</td>
<td></td>
</tr>
<tr>
<td>A. MONITOR CODE</td>
<td>54</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Sample LogAD Code</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>LogAD Overview</td>
<td>6</td>
</tr>
<tr>
<td>1.3</td>
<td>Monitor-Code Interaction</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Griewank's Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Griewank's Algorithm Applied To A Time Line</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Stack Pointers</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Goto Statement Usage</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>Setting A Variable's Adjoint Value</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Sample LogAD Function</td>
<td>22</td>
</tr>
<tr>
<td>4.5</td>
<td>Sample Code Fragment</td>
<td>23</td>
</tr>
<tr>
<td>4.6</td>
<td>Code Fragment After First Preprocessor</td>
<td>23</td>
</tr>
<tr>
<td>4.7</td>
<td>Code To Record Conditional Statements</td>
<td>25</td>
</tr>
<tr>
<td>4.8</td>
<td>Code To Reverse Conditional Statements</td>
<td>25</td>
</tr>
<tr>
<td>4.9</td>
<td>Code To Record A Loop</td>
<td>26</td>
</tr>
<tr>
<td>4.10</td>
<td>Code To Reverse A Loop</td>
<td>26</td>
</tr>
<tr>
<td>4.11</td>
<td>Recursive Function</td>
<td>27</td>
</tr>
<tr>
<td>4.12</td>
<td>Recursive Function Reversed</td>
<td>27</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Determinant Matrix and Computation Sizes</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Determinant LogAD $\sqrt{N}$ v. Original Function Time</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Determinant LogAD $\sqrt{N}$ v. Original Function Space</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Helmholtz Number of Variables and Computation Sizes</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Helmholtz LogAD $\sqrt{N}$ v. Original Function Space (In Megabytes)</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>Helmholtz LogAD $\sqrt{N}$ v. Original Function Time</td>
<td>48</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>4.14</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>4.15</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>5.7</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>5.8</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>5.9</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>5.11</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>
1. INTRODUCTION

In scientific applications ranging from weather forecasting [5, 8, 9, etc.], fluid dynamics [10, 6, 19], and beam physics [3] to neural networks [24, 4, 17] and chemical process simulation [4], the calculation of fast and accurate derivatives through automatic (or computational) differentiation is being recognized as an important step in the optimization process. *Automatic differentiation* is a class of computational techniques where the partial derivatives of functions described by computer algorithms are calculated by first computing local derivatives for the elementary functions and then by propagating derivatives through applications of the chain rule of calculus. In large-scale optimization problems, automatic differentiation is particularly useful since it calculates derivatives exactly without first computing their full symbolic representation.

As Griewank [11, 12] mentions, the alternatives to automatic differentiation (hand coding derivatives, symbolic differentiation, and numerical differentiation) each have their own shortcomings. Hand-coding derivative computations is time-consuming and error-prone. Symbolic differentiation is often computationally expensive because of expression swell. Finally, numerical differentiation is computationally expensive and also error prone since it is an approximation technique.

1.1 Automatic Differentiation

Automatic differentiation techniques fall into two rough categories: the forward mode and the reverse mode. In the forward mode, derivatives are calculated by propagating partial derivatives from the independent variables to dependent variables in
the same way as the original function was computed. While the forward approach may seem to be a natural way to compute derivatives, it is not always the most efficient means. Werbos [26], Linnainmaa [18], Speelpenning [25], and Baur and Strassen [1] independently discovered the fact that all the partial derivatives of a single dependent variable with respect to the independent variables can be calculated with a single reverse pass through a computation. This reverse mode of automatic differentiation can be an order of magnitude faster than the forward mode. Interestingly, this approach forms the basis for the backpropagation algorithm in neural networks [23, 24].

As mentioned by Griewank [11], the partial derivatives of all independent variables with respect to one dependent variable can be calculated using the reverse mode in no more than five times the number of operations as the original function. The problem with the reverse mode of automatic differentiation is that most general-purpose implementations require information about the entire computation to be recorded. For linear-time computations, such as evaluating a neural network at some input, this approach does not require significantly more memory than that needed by the original computation. However, for more computationally intensive calculations, the amount of memory needed to record the entire computation can be prohibitively expensive.

A partial solution to the memory problem of the reverse mode of automatic differentiation is given by Griewank in [13], which observes that the reverse mode of automatic differentiation does not need to record the entire computation. As Griewank observes, the reverse mode can be implemented by checkpointing the computation at appropriate points, recording only the last portion of the computation, computing derivatives on that portion of the computation, and then by restarting the computation from the most recent checkpoint. By using a divide and conquer approach to checkpointing, Griewank's checkpointed reverse mode algorithm incurs only logarithmic slowdown in execution time. More importantly, Griewank's algorithm requires
significantly less memory than the standard reverse mode, and hence can be used in situations where the standard reverse mode would be prohibitively expensive.

As Griewank mentions, the trade-off to reducing the amount of memory required is increasing the amount of time required. Assume that a calculation uses $O(T)$ computation steps and $O(S)$ memory cells. Then, Griewank shows that a checkpointing scheme can compute the reverse mode in $O(T \log T)$ steps, while using only $O(S \log T)$ space. Since $S$ is usually much less than $T$, $O(S \log T)$ space is usually less than the $O(T + S)$ space required by the standard reverse mode.

Griewank's algorithm was first implemented using ADOL-C [14] through UNIX forks and pipes. Later, Benary [2] gives a parallel algorithm that runs Griewank's algorithm on multiple processors using PVM [7]. The parallel algorithm distributes parts of the computation that need to be run forward several different times to different processors and assigns one processor the responsibility of running the code in reverse. With this method, the code running in reverse can execute without stopping because the information it needs is passed to it in a just-in-time manner.

Grimm, Pottier, and Rostaing-Schmidt [15] provide a restricted version of checkpointing for the automatic differentiation tool Odyssee [22]. They give a method of optimizing the time required to reverse loops with a fixed number of registers. The method works for straight line code with nested loops and requires the user to determine which loops to optimize. The number of registers available for use is also specified by the user.

1.2 Operating System Issues

Any checkpointing scheme requires that the computation be occasionally interrupted so that the checkpoints can be created. These interventions can come either from the underlying operating system, in the form of fork calls or threads, or from

---

1 Throughout this thesis, all logs are base 2
code generated by the compiler. Using the operating system is simpler from a programming perspective, but can be unpredictable. Because operating systems give each process a unique number, the number of processes allowed could be limited. In multi-user environments each user is only allowed a small fraction of the total number of processes the machine can handle. This means that the number of checkpoints that can be created is limited, regardless of the size of the checkpoints.

Because each fork call creates an exact copy of the running process, forking large computations can be expensive in terms of virtual memory and page swapping. Since the processes are all performing part of the same task, they will need to communicate with each other, which could require a large number of context switches to occur. This would hinder the speed performance of the overall algorithm.

1.3 LogAD Compiler

Although partial implementations of Griewank's checkpointing algorithm exist, no general purpose checkpointing scheme is available in any currently available automatic differentiation tool. This thesis presents a compiler, called the LogAD compiler, that generates code that implements Griewank's algorithm without using operating system intervention. Checkpointing is implemented via a compiler generated monitor which controls the execution flow and determines at run-time when to checkpoint and how much of the algorithm to record.

The LogAD language is a subset of C, including for loops, assignment statements, functions, basic data types double and integer, and one dimensional arrays, and is meant to be an experimental language to test Griewank's algorithm. Although the language is small, it is powerful enough to implement some non-trivial mathematical algorithms. Figure 1.1 shows a recursive function that calculates the product of an array of values using LogAD code.
double prod(double a[], int index){
    double r;
    if(index == 0){ r = a[index]; }
    else{
        r = a[index] * prod(a, index - 1);
    }
    return(r);
}

void LogAD(){
    double array[5], result;
    int i;
    for(i = 0; i < 5; i = i + 1){
        array[i] = i + 1;
    }
    result <<=;
    result = prod(array, 4);
    printf("result: %g
", result);
    printf("array:
");
    for(i = 0; i < 5; i = i + 1){
        printf(" %g:%g\n", array[i], array[i]');
    }
}

Figure 1.1 Sample LogAD Code

As shown in Figure 1.2, the compiler generates the monitor as well as a forward and reverse version of the code. The resulting code is then compiled with the GNU gcc compiler.

Figure 1.3 shows how the monitor interacts with the code. The number of steps to be executed, S, is determined by the monitor. After S steps have been executed, either forward or in reverse, control is returned to the monitor. The decision of whether or not to create or restore a checkpoint is also determined by the monitor.
1.4 Thesis Organization

This thesis is organized as follows. The reverse mode of automatic differentiation is discussed in more detail in Chapter 2 and Griewank's algorithm is explained in Chapter 3. Chapter 4 gives a detailed description of the LogAD compiler. The results for code generated by the LogAD compiler on some select algorithms, compared with results from code using ADOL-C, is given in Chapter 5. Finally, some conclusions and areas of future work are given in Chapter 6.
Figure 1.3 Monitor-Code Interaction
2. MATHEMATICAL BACKGROUND

At first, the problem of numerically calculating derivatives may appear rather simple, since the definition of a derivative is

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}. \quad (2.1)$$

By setting $h$ to be a "small number," we could get a close approximation to the derivative; however, the choice of $h$ depends on the function as well as the value of $x$. Determining an appropriate $h$ value for a given function is a non-trivial problem. For very complex functions, $h$ needs to be very small. Due to round off errors in finite precision arithmetic, if $h$ is made too small the computer may round $x + h$ to be $x$, thus causing the approximated derivative value to always be zero. If the function has more than one variable and we want to calculate the gradient, then equation 2.1 needs to be evaluated once for each variable, which makes this method very inefficient.

The alternative to approximating derivatives is to calculate them exactly. A technique, known as computational (or automatic) differentiation, uses the chain rule from calculus to compute derivatives without approximation. For further reading on automatic differentiation, see the work of Rall [20], Rall and Corliss [21], or Griewank [11, 12].

One way to think of how reverse automatic differentiation works is in terms of Jacobian matrices. Given a function, $f$, with $n$ dependent variables, $< y_1, \ldots, y_n >$, and $m$ independent variables, $< x_1, \ldots, x_m >$, the Jacobian matrix, $J_f$, is defined as

$$J_f = \left[ \frac{\partial y_i}{\partial x_j} \right]_{m \times n}. \quad (2.2)$$
Given a function, $f$, with $m$ independent variables, $< x_1, \ldots, x_m >$, and a single dependent variable, $y$, the gradient of $f$ is the $m$-element vector

$$\nabla f = \left[ \frac{\partial y}{\partial x_i} \right]_m. \quad (2.3)$$

The chain rule states that given $y = f(u)$ and $u = g(x)$, where $\frac{\partial y}{\partial u}$ and $\frac{\partial u}{\partial x}$ are known, then $\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \times \frac{\partial u}{\partial x}$. We can also apply the chain rule to Jacobians. Given a composite function, $f = f_1 \circ f_2$, where $J_{f_1}$ and $J_{f_2}$ are known, then $J_f = J_{f_1} \times J_{f_2}$. The following example illustrates how we can apply the chain rule to Jacobian matrices.

2.1 Example

Given a function, $f_1 : a = (b + c) \times c$, we wish to compute $\frac{\partial a}{\partial b}$ and $\frac{\partial a}{\partial c}$. This can be done as follows.

1. Break the equation into elementary operations.

   $$f_2 : t_1 = b + c. \quad (2.4)$$

   $$f_3 : a = t_1 \times c. \quad (2.5)$$

2. Compute the Jacobian matrix for each of the elementary operations.

   $$J_{f_2} = \begin{bmatrix} a & b & c & t_1 \\ a & 1 & 0 & 0 & 0 \end{bmatrix} \quad (2.6)$$

   $$J_{f_3} = \begin{bmatrix} a & b & c & t_1 \\ a & 1 & 0 & 0 & 0 \end{bmatrix} \quad (2.7)$$
3. Apply the chain rule, \( J_{f_1} = J_{f_2} \times J_{f_3} \)

\[
J_{f_1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ t_1 & 0 & 1 & 0 \\ c & 0 & 0 & 1 \end{bmatrix}
\]

(2.8)

\[
J_{f_1} = \begin{bmatrix} a \\ b \\ c \\ t_1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ c & 1 & 0 & 1 \\ t_1 + c & 0 & 1 & 1 \\ c & 0 & 0 & 1 \end{bmatrix}
\]

(2.9)

4. By substituting \( t_1 = b + c \), we get \( \frac{\partial a}{\partial b} = c \) and \( \frac{\partial a}{\partial c} = b + 2c \).

2.2 Automatic Differentiation

Automatic differentiation is a class of computational techniques used to calculate gradients and Jacobian matrices of composite functions. In the above example, we can extract \( \frac{\partial a}{\partial b} \) and \( \frac{\partial a}{\partial c} \) by using either a vector times \( J_{f_1} \) or \( J_{f_1} \) times a vector. These two methods are the \textit{forward} and \textit{reverse} modes of automatic differentiation [16].

2.2.1 Forward Mode

We can multiply a vector, \( \bar{x} \), times the Jacobian matrix, \( J_{f_1} \), to extract certain partial derivative values. Since \( J_{f_1} \) is the product of \( J_{f_2} \) and \( J_{f_3} \), then computing \( \bar{x} \) times \( J_{f_1} \) is equivalent to

\[
(\bar{x})^T \times J_{f_1} = \left( ((\bar{x})^T \times J_{f_2}) \times J_{f_3} \right).
\]

(2.10)

Because each Jacobian is very sparse, we can do each vector-matrix multiplication in \( O(1) \) operations.
Because each forward pass extracts a row from the Jacobian matrix, the forward mode needs to be used once for each independent variable in order to compute the gradient. Specifically, if a function has \( n \) independent variables, then \( n \) forward passes are required. For more information about the forward mode, see the work of Rall and Corliss \[21\].

### 2.2.2 Reverse Mode

Because each column in the full Jacobian is a gradient, we can calculate each gradient by multiplying the Jacobian times a unit vector. Multiplying \( J_{f_1} \) times a unit vector, \( \bar{x} \), is equivalent to

\[
J_{f_1} \times \bar{x} = (J_{f_2} \times (J_{f_3} \times \bar{x})).
\]  

(2.11)

Again, we can compute each matrix-vector product in \( O(1) \) steps because the Jacobians for elementary operations are sparse. Each pass of the reverse mode extracts one column from the Jacobian, which is equivalent to extracting one gradient. For a function with \( n \) variables, the reverse mode can calculate the gradient \( n \) times faster than the forward mode \[21\].

Because the reverse mode can calculate one gradient per pass, it is the preferred method when differentiating functions with many independent variables and few dependent variables. However, notice in equation 2.11 that the first step is to multiply \( J_{f_3} \), which corresponds to the final step in the computation, by \( \bar{x} \). Since the final step is used first, information about the computation needs to be recorded so that the steps of the computation can be used in reverse.

Each elementary operation has a small, sparse Jacobian matrix associated with it. If we perform a matrix-vector multiplication with one of these Jacobians, the elements of the resulting vector are equations formed from elements of the matrix and elements of the original vector. For the purposes of reverse automatic differentiation, the elements of the original vector are the partially computed derivative values of the
variables used in the equation. These values are called *adjoint values* and the vector is referred to as an *adjoint vector*.

As an example, consider the elementary operation of addition. The general form of the operation is

\[ x = y + z. \]  

(2.12)

From this equation, we get the following Jacobian matrix

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

(2.13)

We also get the following adjoint vector

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

(2.14)

When we multiply the Jacobian matrix times the adjoint vector, we get

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

(2.15)

Notice that only the elements of the result vector corresponding to \( y \) and \( z \) change.

These changes give the following equations

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

(2.16)

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

(2.17)
These two equations, 2.16 and 2.17, are the *adjoint equations* for addition.

As another example, consider the equation for multiplication, $x = y \times z$. This equation gives the following Jacobian

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

Multiplying this by the adjoint vector yields

\[
\begin{bmatrix}
  x \\
y \\
z
\end{bmatrix}
\begin{bmatrix}
  \hat{x} \\
\hat{y} \\
\hat{z}
\end{bmatrix}
= \begin{bmatrix}
  \hat{x} \\
(\hat{x} \times z) + \hat{y} \\
(\hat{x} \times y) + \hat{z}
\end{bmatrix}
\]

This gives the following adjoint equations

\[
\hat{y} = \hat{y} + (\hat{x} \times z), \quad \text{(2.20)}
\]

\[
\hat{z} = \hat{z} + (\hat{x} \times y). \quad \text{(2.21)}
\]

Notice in equations 2.20 and 2.21 that the adjoint equations use the actual values of $y$ and $z$ and not just their adjoint values. Therefore, these values need to be stored so that they can be accessed during the reverse pass.
3. GRIEWANK'S ALGORITHM

As mentioned earlier, the reverse mode of automatic differentiation can calculate one gradient per pass. However, the memory requirement can make it infeasible to use this method. By using a checkpointing scheme, Griewank's algorithm can significantly reduce the amount of memory required to record the computation and thus makes using the reverse mode a viable choice.

Determining when to checkpoint a computation requires that the size of the computation be known. The size can be determined by doing a preliminary pass without recording by simply counting how many operations are required by the computation. After the size of the computation is determined, checkpoints are placed in the computation according to some partitioning algorithm. Griewank suggests two such algorithms, *binomial* and *bisection*. Partitioning using bisection will be discussed in this thesis. Although Griewank proves that binomial partitioning is optimal, the difference between the binomial and bisection methods is a constant two.

Given a time line with starting point $t_0$ and ending point $t_n$, the time line is partitioned using bisection as follows. Let the initial partitioning point, $p_0$, be equal to $t_0$. The successive partitioning points are placed according to the following equation

$$p_t = p_{t-1} + \frac{t_n - p_{t-1}}{2}. \quad (3.1)$$

In general, each partitioning point is placed halfway between the previous partitioning point and the end of the time line. The time line is bisected until the distance between the last partitioning point and the end of the time line is less than some bound, $b$.

In Griewank's algorithm, the bound used for the bisection is the number of operations that can be recorded and a checkpoint is placed at each partitioning point.
After the bisection partitioning has completed, the algorithm shown in Figure 3 is used.

Let \textit{End} be the end of the computation.
Let \textit{Bound} be the bound on the number of operations to record.
While \textit{End} is not equal to the beginning of the computation, loop

1. Let \textit{CP} be the last checkpoint
2. If the distance from \textit{CP} to \textit{End} is greater than \textit{Bound}, then
   (a) Bisect from \textit{CP} to \textit{End}
   (b) Let \textit{CP} be the last checkpoint
3. Record from \textit{CP} to \textit{End}
4. Reverse from \textit{End} to \textit{CP}
5. Let \textit{End} equal \textit{CP}

End Loop

Figure 3.1 Griewank's Algorithm

3.1 Using Griewank's Algorithm

Consider a computation that has 10 computation steps with the bound set to be 2. We can look at the computation on a time line that starts at 0, the beginning of the computation, and ends at 10, the final step in the computation. Figure 3.2 shows the steps involved in using Griewank's algorithm on a the time line.

The time line starts out with points 0 and 10, which represent the beginning and end of the computation, respectively. By the algorithm, the time line is then bisected from beginning to end.
At points 0, 5, and 8 we checkpoint the computation, keeping the complete state of the computation. Because the computation steps must be numbered with integers, we cannot checkpoint at a fraction of a computation step. After the checkpoints are in place, we restore the final checkpoint, which is point 8, and then record and reverse from there to the end. We then set the end of the computation to be the final checkpoint.

We now have the end of the computation set to point 8. From point 8 to point 10 has been recorded and reversed, so it is no longer needed. The algorithm now restores the last checkpoint, which is now point 5. Since the distance from point 5 to point 8 is three and our bound is set to two, we need to bisect from point 5 to point 8.
We now restore point 7, record until point 8, and then reverse from point 8 to point 7. The end of the computation is then set to point 7. The algorithm continues until the entire computation has been reversed.

3.2 Results of Griewank’s Algorithm

Let $S$ be the amount of space and $T$ be the number of computation steps in the original computation, and let $B$ be the bound set on the number of computation steps to record. Griewank [13] proves that the amount of space required to perform the algorithm is $S\log T$ and the amount of time required is $T\log T$ if bisection is used for the partitioning scheme. The results can be improved by a constant factor of two if binomial partitioning is used instead.

If $B$ is set to one, then only one computation step is recorded at a time. This is the worst case for the speed of the algorithm. The memory requirement is very small, but the algorithm will need to create more checkpoints and will therefore perform slower. If we let $B$ equal $T$, then the entire computation can be recorded. This is equivalent to performing the normal reverse mode of automatic differentiation. The results shown in Chapter 5 show that setting $B$ to $\sqrt{T}$ achieves the above results of $S\log T$ and $T\log T$ with bisection used for partitioning.
4. LOGAD COMPILER

Using Griewank's algorithm will require less memory to record a computation because only a small portion of the computation is recorded at a time. However, checkpoints need to be created and stored, which requires more memory. If the size of the checkpoints is too large, using the checkpointing scheme will not have the desired effect of reducing memory usage. Griewank showed that the checkpoints will not be too large and the increase in time and space over computing the original function will be logarithmic. This chapter gives the architecture of a compiler, called the LogAD (for Logarithmic Automatic Differentiation) compiler, that implements Griewank's algorithm.

The LogAD compiler uses two preprocessors to generate C code that performs reverse automatic differentiation using Griewank's algorithm. The output from the second preprocessor is legal C code that can be compiled using the GNU gcc compiler. Running both preprocessors and then compiling the output with the gcc compiler is done using a shell script, called lcc. Lcc runs the preprocessors, which generate temporary files, compiles the final output with the gcc compiler, and then deletes the temporary files. This effectively allows lcc to behave as a usual compiler. It takes code designed for use with the LogAD compiler and outputs an executable file that performs reverse automatic differentiation using Griewank's algorithm.

4.1 Architecture

At each checkpoint, the complete state of the computation needs to be stored. Because the LogAD compiler does not use the operating system to generate checkpoints,
the state of the computation needs to be stored manually. Manually manipulating system resources, such as the system stack and stack pointers, is not easy to do. Therefore, the LogAD compiler keeps its own stacks and stack pointers to mimic the behavior of the system stack. Therefore, the code generated by the LogAD compiler needs to behave in a manner similar to assembler code. However, generating actual assembler code is not desirable and isn’t portable.

4.1.1 Stacks

Because the compiler needs to keep track of everything about the computation, it must keep its own stacks so that it knows the value of every variable and parameter at any point in time. To avoid problems caused by the need to use different types of variables throughout the computation, several different stacks are kept.

For each stack, three stack pointers are used: base of stack pointer, frame pointer, stack pointer. The base of stack pointer points at the bottom of the stack. This pointer is necessary because the base of the stack is not always zero because of the checkpoints. The frame pointer points at the beginning of the current function’s stack frame and the stack pointer points at the top of the stack. Figure 4.1 shows the relationship between the stack pointers.

Variable names are not kept in the generated code. Each variable is given a stack offset that is used instead of the name. For global variables, the offset is relative to the base of stack pointer and for function parameters and local variables the offset is relative to the frame pointer.

In the current implementation, the normal execution stack is an array of structures. Each structure holds the address of the variable as well as the value of the variable stored at that stack entry. By doing it this way, every variable gets passed by reference and value at the same time. Future implementations should implement this stack in a more efficient manner.
A one-dimensional array of double precision floating point numbers that holds the adjoint values of the variables mirrors the normal stack. Another one-dimensional array of doubles is used as the recording stack that holds the recorded values from the computation. There are two address stacks that are arrays of pointers to type void. One of these stacks holds the addresses used in the forward passes and one holds the addresses used in the reverse pass. Two integer arrays are used. One is used to record the boolean values of conditional statements and another is used to record the counters needed for loop statements.

4.1.2 C Language Constructs

In order to simulate the behavior of assembler code, during a function call the values of the parameters and the return address need to be pushed onto the stack, execution jumps to the beginning of the function, and at the end of the function the return address is popped off the stack and control jumps back. It was undesirable to generate assembler code, so the same behavior needs to be done using C language constructs. The constructs needed for this are goto statements and labels.
A label in C is a name followed by a colon. The GNU C compiler, gcc, allows the address of a label to be stored in a variable that is declared as a pointer to type void. Using a goto statement, the program can then jump to what is pointed at by the variable. The usage of the goto statement is shown in Figure 4.2.

```
double foo(){
    void *addr;

    // code block
    addr = &&label;
    goto *addr;
    // code block

    label:
}
```

Figure 4.2 Goto Statement Usage

4.1.3 LogAD Language Constructs

Some constructs had to be added to the LogAD language to initialize the adjoint values of the dependent variables. Before a computation begins, the adjoint value of the dependent variable needs to be set to one. The LogAD compiler adds the $\ll$ operator for this purpose. Code that creates a variable, x, and sets its adjoint value to one is shown in Figure 4.3.

After a computation is completed, it is necessary to be able to access the adjoint value of the variables involved in the computation. The adjoint value of a variable can be accessed by using a back single quote after the variable's name. The code segment shown in Figure 4.4 is legal LogAD code that calls a function with a parameter and then outputs the value of that variable and its adjoint.
void LogAD()
{
    double x;
    x <<;
}

Figure 4.3 Setting A Variable's Adjoint Value

void LogAD()
{
    double x,y;
    y = square(x);
    printf("x: %g\n",x);
    printf("x adjoint: %g\n",x');
}

Figure 4.4 Sample LogAD Function

The main function in a LogAD program is called LogAD, not main, and the return type is void. The printf statement does work with the LogAD compiler and can be used to output results.

4.2 First Preprocessor

The first preprocessor takes as input code that is a subset of C. The following C features have been implemented: double type, int type, one-dimensional arrays, functions, conditional statements, for statements, return statements, basic math operators, and assignment statements. Other features of C were not necessary to test Griewank's algorithm. Future versions of the compiler should implement all features of the C language.
The main purpose of the first preprocessor is to separate complex mathematical operations into elementary operations. This requires temporary variables to be created. Consider the code fragment shown in Figure 4.5. After the first preprocessor is done, the code in Figure 4.5 will get transformed into the code shown in Figure 4.6.

```c
double foo()
{
    double a,b,c,d;
    b = 2;
    c = 3;
    d = 4;
    a = b * c * d;
}
```

Figure 4.5 Sample Code Fragment

```c
double foo()
{
    double a,b,c,d;
    double t0,t1;
    b = 2;
    c = 3;
    d = 4;
    t0 = b * c;
    t1 = t0 * d;
    a = t1;
}
```

Figure 4.6 Code Fragment After First Preprocessor
4.3 Second Preprocessor

The second preprocessor is responsible for generating the code to implement Griewank's algorithm, including code to reverse the computation and a run-time monitor. It assumes that the input has come from the first preprocessor, and therefore all mathematical functions are elementary operations.

4.3.1 Reversing the Code

Reversing a sequential block of code is straightforward to do. Given a block of statements, \( S_1, \ldots, S_n \), reversing the execution simply reverses the order of the statements, \( S_n, \ldots, S_1 \). However, there are some special cases that need to be considered when reversing the execution flow of a program, including conditional statements, loops, and function calls.

4.3.1.1 Conditional Statements

To correctly reverse a conditional statement, a value of either true or false is stored when recording the computation and used when the computation is reversed. To guarantee that the recorded boolean values are used in the correct order, the value is stored at the end of the block of statements executed. The LogAD compiler keeps a stack of integers that is used exclusively for boolean values from conditional statements. The code that is generated to record a conditional statement has the form shown in Figure 4.7 and the reversed equivalent has the form shown in Figure 4.8.

4.3.1.2 Loops

The method of reversing a loop used by the LogAD compiler uses a counter for each loop to be kept to guarantee that the loop is executed the correct number of times. In order to guarantee that nested loops or function calls within loops do not
if(condition){
    // statements
    push(TRUE);
}
else{
    // statements
    push(FALSE);
}

Figure 4.7 Code To Record Conditional Statements

if(pop()){  
    // statements
}
else{ 
    // statements
}

Figure 4.8 Code To Reverse Conditional Statements

cause problems, the LogAD compiler uses a stack of counters. When a loop is entered, a new counter is pushed onto the stack and used during the execution of the current loop. When the loop is completed, the value of the counter is popped off the stack and recorded so that it can be used when the code is reversed. The form of the code to record a for loop is shown in Figure 4.9 and the code to reverse a for loop is shown in Figure 4.10.

4.3.1.3 Function Calls

When a function is called in reverse, the steps taken are similar to normal function calls, but the function starts at the end instead of the beginning. Therefore, a return statement in reverse does nothing. Returning from a function is done at the end of
push(0);
for(count=0; count<n; count=count+1) {
    // statements
    stack[stack_ptr - 1] ++;
}
record(pop());

Figure 4.9 Code To Record A Loop

for(index = pop(); index>0; index--){
    // statements reversed
}

Figure 4.10 Code To Reverse A Loop

the reverse code, which is equivalent to the beginning of the forward version of the code. A function call in reverse is equivalent to undoing a return call going forward. To see this, consider the recursive function shown in Figure 4.11. The reversed version of the function is given in Figure 4.12.

For the sake of argument, assume the condition for the if statement is true the first time it is hit and false the second time. When the code is run forward block 1 is executed and the function is called recursively. The second time the condition is hit it is false, so block 2 is executed. A value of false is then pushed onto the stack, and the function returns. After returning from the recursive call, a value of true is pushed onto the stack and the function returns from the original call. When reversed, the first time the condition is hit a value of true is popped off the stack. Next, the function is called recursively, which reverses a return statement. After the recursive call, the if statement is once again hit. A value of false is popped off the stack and block 2 is run. Then the function returns. After returning, block 1 is run and then the program exits.
double foo(){
    if(condition){
        // block 1
        foo();
    }
    else{
        // block2
    }
    return();
}

Figure 4.11 Recursive Function

double foo_reverse(){
    return();
    if(condition){
        foo_reverse();
        // block 1 reversed
        push(TRUE);
    }
    else{
        // block 2 reversed
        push(FALSE);
    }
}

Figure 4.12 Recursive Function Reversed
It can be seen that this correctly reverses the execution flow of the code. In the forward mode, block 1 is executed, the function is called, block 2 is executed, and then the program exits. In the reverse, a return is reversed, block 2 is executed, the end of the reverse code is reached and the function returns, and then block 1 is executed.

4.3.2 The Monitor

The monitor is responsible for determining the size of the computation, bisecting the computation, and handling the checkpoints. The monitor is called when a function is called from the main program block which means that any code in the main function block does not get reversed and is therefore not differentiated. Any code that needs to be differentiated must start in a function that gets called from the main program block.

The size of the computation is calculated by running the computation forward to completion without recording. For purposes of efficiency, each block is counted instead of each statement. A counter is incremented at the beginning of each block and when the computation is finished control is returned to the monitor.

After the size of the computation is determined, it needs to be bisected and checkpoints put in place. The counter is reset to zero and the computation is again run forward to completion without recording. This time through, however, control is returned to the monitor at the end of each block. If the monitor determines a checkpoint is needed, the checkpoint is created and control is returned to the program at the same point it stopped at. Because of the way the bisection is done, the computation is not exactly bisected. However, it is safe to assume that any large mathematical calculation will not be written in one large block, but will contain several blocks, either conditionals or loops. Therefore counting each block does a good job of approximately bisecting the computation.
When a section of the computation needs to be recorded, the number of blocks to record is equal to the distance between the current checkpoint and the end of the computation. A counter is kept to make sure that the correct number of steps are recorded. This counter is originally set to the number of blocks to record. As the computation is run, the counter is decremented at the beginning of each block. At the end of each block, if the counter is zero then control is returned to the monitor.

When a section of the computation is run in reverse, a similar counter is used. It is again set to the number of blocks to execute. As the computation is run in reverse, at the beginning of each block, which is equivalent to the end of the block in the forward version, the counter is checked to see if it has reached zero. If it has, control is returned to the monitor. At the end of each block, which is equivalent to the beginning of the block in the forward version, the counter is decremented.

The complete code for the monitor is given in Appendix A. Because of the use of goto labels, all the code generated by the LogAD compiler is in the same main function. The variables used by the monitor are global variables and some of them are also used outside the monitor code. For example, the ref_addr variable is a pointer to type void and is used in the forward mode to determine if control should return to the monitor. If ref_addr is NULL when the code is run forward, control is not returned to the monitor until the computation has finished. If ref_addr is not NULL, it holds the address to return to when the forward code needs to return to the monitor.

4.3.2.1 Checkpoints

When a checkpoint is needed, the complete state of the computation is saved. This is done by pushing important values onto the stack and making a copy of the current stack. This is done for the normal stack and the forward address stack. Values that are considered important are the stack pointers, the number of the checkpoint (the point in the computation where the checkpoint is placed), the address where
execution stopped, the current return value of the function, and the line number where execution stopped. After all the needed values are pushed onto the stack, a copy of the original stack is made.

When it becomes necessary to restore a checkpoint, the values that were pushed onto the stack are restored by popping their old values off the stack. This is all that is necessary to restore the state of the computation. By popping the old values, the stack pointers are restored to their original values.

If the computation needs to be bisected after restoring a checkpoint, then the checkpoint that was just restored needs to be replaced. Consider Figure 4.13. The computation is recorded and reversed from checkpoint P2 to the end, and then from checkpoint P1 to checkpoint P2. When this is finished, the computation is restored at P0. The distance from P0 to P1 is too large, so bisection is required. This means that eventually the computation will return to P0. However, that checkpoint was just removed so P0 must be restored before bisection is done from P0 to P1.

![Figure 4.13 Time Line With Checkpoints](image-url)
It is important to note that all checkpointing is done within the computation itself. The code to create and restore checkpoints is compiled into the computation as part of the monitor, so the operating system is not used at all.

4.4 Sample Code Trace

Figure 4.14 contains the skeleton of a recursive computation. The actual statements in the statement blocks are irrelevant, only the statement blocks themselves are necessary. Figure 4.15 shows an example execution trace for the computation in Figure 4.14.

The first decision that had to be made with how the LogAD compiler handles execution of the code is how the blocks are counted. It was decided that the counter would be incremented at the beginning of code blocks, such as conditional statements and loops, and checking whether or not to return to the monitor would be done at the end of the code blocks. Notice in Figure 4.14 that in this case the only statement blocks that do not increment the counter are statement blocks 7 and 8 and the return statement. Notice in Figure 4.15 that if this method is used, the last step in the execution trace to increment the counter is step 6.

After the preliminary run of the code to determine the size of the computation, the size will be 5, with steps 1-4 and step 6 incrementing the counter. When the program is recorded, it will execute until the counter reaches 0, which will be 4 steps too early. When the code is reversed, it will start reversing at step 6 instead of step 10 because that is where recording stopped. Because of this, it is necessary to treat return statements as code blocks. This way, the counter will be incremented and the check to return to the monitor will be done when a return statement is encountered.

If we set the recording bound to 1, then in Figure 4.15 the final checkpoint will be placed between steps 7 and 8. When the computation is recorded from the final checkpoint to the end, steps 7, 8, and the final return statement are performed. The
counter used is set to one because only one code block had been counted. This means that the monitor will reverse the return statement, decrement the counter, and then stop when the check is done before step 9 is reversed. However, it should still reverse steps 7 and 8 even though they are not counted.

This problem was solved by keeping track of the line number where recording began. When the final section is reversed, the counter is decremented after step 10 is reversed. When the check is done before reversing step 9, the counter will be the correct value to return to the monitor but the line number will not be correct. The computation will continue to reverse until the check before reversing step 7. At this point, the counter is the correct value and the line number is also correct so control will return to the monitor. The only time this method could fail would be if the same line of code is executed more than once without entering another code block. This is not possible. The only way a line of code could be executed more than once would be in the case of either a loop or recursive function call.
double foo()
{
    if(condition 1) statement block 1
    else{
        statement block 2
        if(condition 2) statement block 3
        else statement block 4
        for()
        {
            statement block 5
            if(condition 3){
                statement block 6
                foo()
                statement block 7
            }
            statement block 8
        }
    }
    return()
}

Figure 4.14 Sample Computation Outline

1. statement block 2
2. statement block 3
3. statement block 5
4. statement block 6
5. foo()
6. statement block 1
7. return()
8. statement block 7
9. statement block 8
10. return()

Figure 4.15 Sample Code Execution Trace
5. LOGAD COMPILER RESULTS

This chapter describes the results achieved from running code generated by the LogAD compiler. The LogAD code was run using different recording sizes and the results are compared with ADOL-C code.

In order to test the performance of the code generated by the LogAD compiler, two algorithms were chosen because of the amount of memory required to calculate their derivatives using the reverse mode of automatic differentiation. The first algorithm calculates the determinant of a matrix and the second algorithm computes the Helmholtz energy function. These algorithms were run using LogAD code as well as ADOL-C code and the results of the LogAD compiler will be compared with the ADOL-C results. The computer used to run these tests is a 100 Mhz Pentium with 32M of RAM running Linux. The ADOL-C code was compiled with the GNU g++ compiler and the LogAD code was compiled with the GNU gcc compiler.

To compare how well the LogAD compiler compared with ADOL-C, it is necessary to compare results from similar situations. It is not easy to reduce the amount of system resources that the ADOL-C code can use, so for straight comparisons the recording size of the LogAD code was set to be as large as possible so that it, too, could use full system resources. To show what happens to the time and space requirements under severely restricted conditions, the recording size of the LogAD code was set to the log base 2 of the computation size, \( N \). Tests were also run setting the recording size to the square root of the computation size. Comparisons were also done between LogAD code and computing the original function.
5.1 Determinant Example

Computing the derivative of the determinant function for a matrix, \( M \), results in the cofactor matrix, \( C \), of \( M \). Dividing \( C \) by the determinant of \( M \) gives \( M^{-1} \). This makes this a desirable algorithm to differentiate because the determinant and inverse of the matrix are calculated at the same time. The straightforward algorithm to compute the determinant of a matrix is an \( O(n!) \) recursive algorithm. The amount of space required to perform this algorithm is small, but the amount of space required to record the computation is extremely large. For this reason, it was picked as a good test of the LogAD compiler's ability to reduce the space required to perform the reverse mode of automatic differentiation. The size of the matrices used and the corresponding computation sizes are given in Table 5.1.

<table>
<thead>
<tr>
<th>Matrix Size</th>
<th>Computation Size</th>
<th>( \log_2(N) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 ( \times ) 3</td>
<td>87</td>
<td>6</td>
</tr>
<tr>
<td>4 ( \times ) 4</td>
<td>399</td>
<td>8</td>
</tr>
<tr>
<td>5 ( \times ) 5</td>
<td>2,213</td>
<td>11</td>
</tr>
<tr>
<td>6 ( \times ) 6</td>
<td>14,529</td>
<td>13</td>
</tr>
<tr>
<td>7 ( \times ) 7</td>
<td>110,379</td>
<td>16</td>
</tr>
<tr>
<td>8 ( \times ) 8</td>
<td>952,331</td>
<td>19</td>
</tr>
<tr>
<td>9 ( \times ) 9</td>
<td>9,194,529</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure 5.1 shows the timing results of LogAD and ADOL-C. Notice that the LogAD code was faster than the ADOL-C code until a matrix of order 7 was used. The reason for this is that when ADOL-C records the computation and the size becomes large, it starts recording to disk. The LogAD compiler, however, keeps
everything in memory. When the recording becomes too large, the number of page faults increases. For the matrix of order 7, the LogAD code had over 2,000 page faults while the ADOL-C code had less than 600.

![Figure 5.1 ADOL-C v. LogAD Determinant Time Results](image)

The memory used for the determinant computations are shown in Figure 5.2. For the smaller examples, the amount of memory used by the LogAD code was much less than the amount of memory used by the ADOL-C code. For medium sized examples, the amount of space used by the LogAD code was greater than that used by the ADOL-C code until the matrix of order 9 was used. The memory usage was basically the same for the order 8 and order 9 matrices using LogAD code because the order 9 matrix computation was too large for the LogAD compiler to record. Therefore, it started checkpointing. This shows an attractive quality of the LogAD code. When there are not enough resources available to record the entire computation, it can still perform reverse automatic differentiation by falling back on the checkpointing scheme.
Figure 5.2 ADOL-C v. LogAD Determinant Space Results

Figure 5.3 shows the timing results of the LogAD code when the recording size is set to the log of the computation size compared with the results when the entire computation could be recorded. Notice that for the smaller matrices, the time increase was not noticeable. However, as the matrices become larger the time increase becomes more pronounced.

Figure 5.4 shows the memory requirement of the LogAD code for the case when the entire computation is recorded compared with when the log of the computation size is used as a bound. In Figure 5.4, the memory usage for when the recording size is set to log(N) grows very slowly while the memory usage for the case when the entire computation is recorded grows rather quickly. This shows that the LogAD compiler can significantly reduce the amount of memory needed to calculate the derivatives.

For the next test, the bound of the recording size was set to the square root of the computation size. The space comparisons between this test and setting the bound to the log of the computation size is shown in Figure 5.5 and the time comparison
Figure 5.3 LogAD Determinant Time Results

Figure 5.4 LogAD Determinant Space Results
is shown in Figure 5.6. Notice that for the determinant algorithm, using $\sqrt{N}$ as a bound was faster and used less memory than using $\log(N)$ as a bound. Since the recording size was very small for the $\log(N)$ case, the number of checkpoints required became significantly large. This required the computation to be interrupted more often, which slowed it down, and required more checkpoints to be created, which required more memory.

![Figure 5.5 Determinant LogAD $\sqrt{N}$ v. $\log(N)$ Space Results](image_url)

For the final test, the LogAD compiler results were compared with computing the original function. A desirable goal is to have the LogAD code perform the reverse mode of automatic differentiation with logarithmic growth in time and space over computing the original function. For the determinant example, this was achieved with setting the record size to $\sqrt{N}$.

Table 5.2 shows the timing comparisons between the original function and using Griewank’s algorithm with the recording size set to the square root of the computation
Figure 5.6 Determinant LogAD $\sqrt{N}$ v. log(N) Time Results

As can be seen from the table, for smaller examples there is very little increase in time. For middle and large examples, the time increase is very close to log(N).

The space comparisons between the LogAD code with a recording size of $\sqrt{N}$ and the original function are shown in Table 5.3. As can be seen from the table, the increase in the space requirement is less than log(N).

5.2 Helmholtz Energy Function

The Helmholtz energy function is an interesting test for automatic differentiation tools because it can use a large number of independent variables. The equation for the Helmholtz energy function is given in [11] as

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

(5.1)

where $0 \leq x, b \in R^n$, and $A = A^T \in R^{n \times n}$. The reverse mode works particularly well on it because the partial derivatives of all the independent variables, no matter how
Table 5.2 Determinant LogAD $\sqrt{N}$ v. Original Function Time

<table>
<thead>
<tr>
<th>Matrix Order</th>
<th>log($N$)</th>
<th>Original Function</th>
<th>LogAD $\sqrt{N}$ Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>0.22</td>
<td>0.23</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>0.23</td>
<td>0.25</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>0.23</td>
<td>0.36</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>0.3</td>
<td>1.18</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>0.8</td>
<td>8.96</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>5.14</td>
<td>93.62</td>
</tr>
<tr>
<td>9</td>
<td>23</td>
<td>47.3</td>
<td>1,136.05</td>
</tr>
</tbody>
</table>

Table 5.3 Determinant LogAD $\sqrt{N}$ v. Original Function Space

<table>
<thead>
<tr>
<th>Matrix Order</th>
<th>log($N$)</th>
<th>Original Function</th>
<th>LogAD $\sqrt{N}$ Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>2,080</td>
<td>7,464</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>2,656</td>
<td>11,100</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>3,328</td>
<td>17,420</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td>4,096</td>
<td>24,212</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>4,960</td>
<td>39,112</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>5,920</td>
<td>60,660</td>
</tr>
<tr>
<td>9</td>
<td>23</td>
<td>6,976</td>
<td>113,300</td>
</tr>
</tbody>
</table>
many there are, can be calculated in one pass. For a large number of variables, the computation becomes very large and requires a lot of memory to record. The results from the LogAD compiler are compared with results from ADOL-C as well as results from using divided differencing. The number of variables used and the corresponding computation sizes are shown in Table 5.4.

Table 5.4 Helmholtz Number of Variables and Computation Sizes

<table>
<thead>
<tr>
<th>Variables</th>
<th>Computation Size</th>
<th>log(N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>5,051</td>
<td>12</td>
</tr>
<tr>
<td>200</td>
<td>20,101</td>
<td>14</td>
</tr>
<tr>
<td>300</td>
<td>45,151</td>
<td>15</td>
</tr>
<tr>
<td>400</td>
<td>80,201</td>
<td>16</td>
</tr>
<tr>
<td>500</td>
<td>125,251</td>
<td>17</td>
</tr>
<tr>
<td>600</td>
<td>180,301</td>
<td>17</td>
</tr>
<tr>
<td>700</td>
<td>245,351</td>
<td>18</td>
</tr>
<tr>
<td>800</td>
<td>320,401</td>
<td>18</td>
</tr>
<tr>
<td>900</td>
<td>405,451</td>
<td>18</td>
</tr>
<tr>
<td>1000</td>
<td>500,501</td>
<td>19</td>
</tr>
<tr>
<td>2000</td>
<td>2,001,001</td>
<td>21</td>
</tr>
<tr>
<td>3000</td>
<td>4,501,501</td>
<td>22</td>
</tr>
<tr>
<td>4000</td>
<td>8,002,001</td>
<td>23</td>
</tr>
<tr>
<td>5000</td>
<td>12,502,501</td>
<td>23</td>
</tr>
</tbody>
</table>

Figure 5.7 shows the comparison between LogAD, ADOL-C, and divided differencing. Notice that the time for divided differencing is increasing at a much faster rate than the others. Also, LogAD and ADOL-C have effectively identical times until
2,000 variables were used. ADOL-C could not do more than 2,000 variables because it ran out of memory, but LogAD could do considerably more. Tests were done with up to 5,000 variables, which took around 12 minutes to compute. However, given enough time LogAD could do examples with far more than 5,000 variables.

![Graph showing time vs. number of variables for LogAD, ADOL-C, and Divided Difference methods.](image)

**Figure 5.7** LogAD, ADOL-C, Divided Difference Helmholtz Time Results

The space requirements for LogAD and ADOL-C are shown in Figure 5.8. Divided differencing does not require much memory because it does not use the reverse mode of automatic differentiation. With an input of 1,000 variables, LogAD could no longer record the entire computation and started checkpointing. The results of this are dramatic. The memory requirement from that point on is almost constant. ADOL-C, however, could not complete the function with more than 2,000 variables. That example required 45 Megabytes of disk space. The next example, 3,000 variables, ran out of space on an 85 Megabyte disk partition and could not finish.
The time differences between using LogAD with no explicit bound and using LogAD with the bound set to \( \log(N) \) is shown in Figure 5.9. Notice that with the bound, the time requirement grows rather quickly and performing all the same examples as the unbounded version would require large amounts of time. The space difference is shown in Figure 5.10. Again, the LogAD compiler is able to perform the calculations with very limited memory.

The next comparison to be done is between LogAD code with a bound of \( \log(N) \) and LogAD code with a bound of \( \sqrt{N} \). The time comparison is shown in Figure 5.11 and the space comparison is shown in Figure 5.12. Using the square root again requires less memory and is faster than using the log.

The next question is whether or not using \( \sqrt{N} \) as a bound attains the desired logarithmic growth over computing the original function. Table 5.5 shows that the increase in the space requirement is much less than logarithmic. Table 5.6 shows that
Figure 5.9 LogAD v. LogAD log Helmholtz Time Results

Figure 5.10 LogAD v. LogAD log Helmholtz Space Results
Figure 5.11 LogAD $\sqrt{N}$ v. LogAD $\log(N)$ Helmholtz Time Results

Figure 5.12 LogAD $\sqrt{N}$ v. LogAD $\log(N)$ Helmholtz Space Results
the increase in time is very close to logarithmic. The times for the larger examples are slightly more than logarithmic, but not much.

Table 5.5 Helmholtz LogAD $\sqrt{N}$ v. Original Function Space (In Megabytes)

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>log(N)</th>
<th>Original Function</th>
<th>LogAD $\sqrt{N}$ Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>12</td>
<td>0.009</td>
<td>0.04</td>
</tr>
<tr>
<td>200</td>
<td>14</td>
<td>0.015</td>
<td>0.09</td>
</tr>
<tr>
<td>300</td>
<td>15</td>
<td>0.022</td>
<td>0.13</td>
</tr>
<tr>
<td>400</td>
<td>16</td>
<td>0.029</td>
<td>0.18</td>
</tr>
<tr>
<td>500</td>
<td>17</td>
<td>0.036</td>
<td>0.22</td>
</tr>
<tr>
<td>600</td>
<td>17</td>
<td>0.043</td>
<td>0.26</td>
</tr>
<tr>
<td>700</td>
<td>18</td>
<td>0.05</td>
<td>0.3</td>
</tr>
<tr>
<td>800</td>
<td>18</td>
<td>0.057</td>
<td>0.37</td>
</tr>
<tr>
<td>900</td>
<td>18</td>
<td>0.064</td>
<td>0.41</td>
</tr>
<tr>
<td>1000</td>
<td>19</td>
<td>0.07</td>
<td>0.46</td>
</tr>
<tr>
<td>2000</td>
<td>21</td>
<td>0.139</td>
<td>0.97</td>
</tr>
</tbody>
</table>
Table 5.6 Helmholtz LogAD $\sqrt{N}$ v. Original Function Time

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>log($N$)</th>
<th>Original Function</th>
<th>LogAD $\sqrt{N}$ Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>12</td>
<td>0.25</td>
<td>0.64</td>
</tr>
<tr>
<td>200</td>
<td>14</td>
<td>0.33</td>
<td>1.9</td>
</tr>
<tr>
<td>300</td>
<td>15</td>
<td>0.45</td>
<td>4.3</td>
</tr>
<tr>
<td>400</td>
<td>16</td>
<td>0.62</td>
<td>8.1</td>
</tr>
<tr>
<td>500</td>
<td>17</td>
<td>0.85</td>
<td>12.6</td>
</tr>
<tr>
<td>600</td>
<td>17</td>
<td>1.12</td>
<td>18.3</td>
</tr>
<tr>
<td>700</td>
<td>18</td>
<td>1.44</td>
<td>25.7</td>
</tr>
<tr>
<td>800</td>
<td>18</td>
<td>1.81</td>
<td>34.2</td>
</tr>
<tr>
<td>900</td>
<td>18</td>
<td>2.26</td>
<td>43.7</td>
</tr>
<tr>
<td>1000</td>
<td>19</td>
<td>2.71</td>
<td>55.1</td>
</tr>
<tr>
<td>2000</td>
<td>21</td>
<td>10.13</td>
<td>245.6</td>
</tr>
</tbody>
</table>
6. CONCLUSIONS AND FUTURE RESEARCH

In this thesis, we have shown that the lack of system resources does not exclude the use of the reverse mode of automatic differentiation as an option on large computations. By using Griewank's algorithm, computations that cannot be finished by normal implementations of the reverse mode of automatic differentiation can be finished with a small increase in time. By setting the recording size to be the square root of the computation size, the reverse mode of automatic differentiation can still be done with logarithmic growth over calculating the original function.

As of now, the code generated by the LogAD compiler can take one argument on the command line, the record size to use. If no arguments are given, a default size is used. The default size is compiled into the monitor and is usually the log or the square root of the computation size. It would be useful to have the monitor calculate the record size based on available system resources. Future implementations should use this method. However, because the size of the computation is currently calculated by how many code blocks are entered during the computation this is not simple to do because different computations will have different amounts of code inside each block.

The size of the stacks used by the LogAD compiler is currently hard-coded. In the future, it would be better to have the monitor determine how big each stack needs to be and dynamically allocate the necessary stack space. Because large computations were used to test the LogAD compiler, the stack sizes used were very large. This means that for the smaller tests even though very little of the stack space was used, large stacks were still allocated. This should be changed in the future.

The code generated by the LogAD compiler is fairly inefficient. Because of the use of goto statements and labels, using gcc’s optimization options had no effect on
the performance of the LogAD code. An area to look into in the future would be to find a way to have the same behavior that LogAD code currently has but use a more efficient and compiler-friendly method. Even with the inefficiencies, the code was still faster than ADOL-C code. Increasing the efficiency of the LogAD code should produce an extremely fast and useful automatic differentiation tool.

The hardest problem to solve with the LogAD compiler is the need to checkpoint the computation. An efficient method of checkpointing needs to be found. The LogAD compiler used simulated assembler code to handle checkpointing. This method worked very well, but cannot be optimized by the compiler. Griewank used fork calls to handle checkpoints in his implementation. This has the benefit of being able to use normal C or C++ code that can be optimized by the compiler. However, using fork calls can use a lot of extra memory and has slow downs from context switching. A possible solution is to use threads instead of forks. This would still allow the compiler to generate code that can be optimized and would have less overhead from context switching.
BIBLIOGRAPHY
BIBLIOGRAPHY


program generation framework: A case study for a weather forecasting grid-
point model. In Martin Berz, Christian Bischof, George Corliss, and Andreas
Griewank, editors, *Computational Differentiation: Techniques, Applications,

[9] Victor V. Goldman and Gerard Cats. Program generation techniques for the
development and maintenance of numerical weather forecast grid models. In
Jack Dongarra, Kai Madsen, and J. Wasniewski, editors, *Proceedings of the
Workshop on Applied Parallel Computing in Physics, Chemistry and
Engineering Science (PARA95)*, Lecture Notes in Computer Science, Vol.

Conference, AIAA Paper 93-3321*. American Institute of Aeronautics and

abe, editors, *Mathematical Programming: Recent Developments and


[13] Andreas Griewank. Achieving logarithmic growth of temporal and spatial com-
plexity in reverse automatic differentiation. *Optimization Methods and Software*,

[14] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C, a package for the

[15] José Grimm, Loïc Pottier, and Nicole Rostaing-Schmidt. Optimal time and
minimum space-time product for reversing a certain class of programs. In Martin
Berz, Christian H. Bischof, George F. Corliss, and Andreas Griewank, editors,
*Computational Differentiation: Techniques, Applications, and Tools*,

Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation


A. MONITOR CODE

monitor_start:
/* Addresses of the forward and reverse versions of the function
to call are passed in on the stack. */
ref_reverse_addr = aPop();
ref_forward_addr = aPop();

/* Store the original base and stack pointers for the normal stack
and address stack */
ref_base = dStack.base_ptr;
ref_stack = dStack.stack_ptr;
ref_aBase = aStack.base_ptr;
ref_aStack = aStack.stack_ptr;

/* Create a checkpoint at point to */
aPush(&&prelim_cp);
record = 0;
goto checkpoint_start;
prelim_cp:

/* Do preliminary run to determine the size of the computation */
aPush(&&check_done);
ref_addr = NULL;
goto *ref_forward_addr;
If an argument was given on the command line, use it for the record size. Otherwise use the square root of the computation size. 

```c
if(argc == 1){
    dblRevSize = sqrt(execCount);
    revSize = (int)dblRevSize;
}
else{
    if(sscanf(argv[1], "%d", &revSize) != 1){
        dblRevSize = sqrt(execCount);
        revSize = (int)dblRevSize;
    }
}
```

Store the return value from the function. The return value isn't right after doing Griewank's algorithm, so we need to store it. 

```c
ref_retVal = retVal;
```

/* Restore original base and stack pointers */
```c
dStack.base_ptr = ref_base;
dStack.stack_ptr = ref_stack;
aStack.stack_ptr = ref_aStack;
aStack.base_ptr = ref_aBase;
```

/* Set bisection to start at 0 and end at the size of the
computation */
refBiStart = 0;
refBiMid = 0;
refBiEnd = execCount;

/* If the entire computation can be recorded, don't bisect.
   If the entire computation cannot be recorded, do bisection. */
if((refBiEnd - refBiMid) > revSize){
aPush(&&prep);
goto refBisect;
}
prep:

/* Reset stack pointers. */
dStack.base_ptr = ref_base;
dStack.stack_ptr = ref_stack;
aStack.stack_ptr = ref_aStack;
aStack.base_ptr = ref_aBase;

/* Push final return address */
revPush(&&Griewank_reverse);

/* Loop to do Griewank's algorithm */
Griewank_while:

/* If the end equals the beginning, we're done */
if(refBiEnd == 0) goto Griewank_while_end;
/* Restore previous checkpoint */
aPush(&&Griewank_restore1);
goto undo_checkpoint;
Griewank_restore1:

/* If the distance from the checkpoint to the end is too large, then we need to bisect */
if((refBiEnd - refBiMid) > revSize){
aPush(&&Griewank_bisect);
refBiStart = refBiMid;
goto refBisect;
Griewank_bisect:
  /* Restore previous checkpoint */
  aPush(&&Griewank_restore2);
  goto undo_checkpoint;
Griewank_restore2:
}

/* Record from last checkpoint to the end */
recCount = refBiEnd - refBiMid; /* Record size */
ref_addr = &&Griewank_record; /* Return address */
record = 1; /* Turn recording on */
goto *ref_forward_addr;
Griewank_record:
record = 0; /* Turn recording off */

/* Reverse from end to last checkpoint */
revCount = refBiEnd - refBiMid; /* Reverse size */
ref_addr = &Griewank_reverse; /* Return address */
goto *ref_reverse_addr;
Griewank_reverse:

/* Set the end to be the last checkpoint */
refBiEnd = refBiMid;
goto Griewank_while;
Griewank_while_end:

/* Entire computation has been reversed */

/* Restore return value */
retVal = ref_retVal;
dStack.stack_ptr = dStack.base_ptr + 1;

/* Return to caller */
temp = aPop();
goto *temp;

/* Routine to bisect the computation */
refBisect:

/* The middle starts out equal to the beginning */
refBiMid = refBiStart;
ref_ret_addr = aPop();
while_start:

/* If the middle is beyond the end, we're done */
if(refBiMid >= refBiEnd) goto while_end;

/* If the middle is half way between the start and the end, 
 create a checkpoint */
if(refBiMid >= (refBiStart + ((refBiEnd - refBiStart)/2))){
aPush(&&BiCheckPt);
goto checkpoint_start;
BiCheckPt:
/* Set the start point to be the current mid-point */
refBiStart = refBiMid;

/* If we're close enough to the end, we're done */
if((refBiEnd - refBiMid) < revSize){
    goto while_end;
}
}

/* Go execute another block */
execCount = 0;
ref_addr = &&BiLoop;
goto *ref_forward_addr;
BiLoop:

/* Increment mid-point by how many blocks were executed */
refBiMid += execCount;
goto while_start;
while_end:
/* Computation has been bisected. Return to caller. */
ref_addr = NULL;
goto *ref_ret_addr;
refBisect_end:

/* Routine to create a checkpoint */
checkpoint_start:
/* Keep track of how many check points currently exist */
cpCount++;
temp = aPop();
{
    /* Checkpoint double stack */
    int t,i;
    t = dStack.stack_ptr;
dPush(retVal,-1);
dPush(line,-1);
dPush(refBiMid,-1);
dPush(dStack.base_ptr,-1);
dPush(t,-1);
dPush(dStack.bs_ptr,-1);
    for(i = dStack.bs_ptr;i < (dStack.bs_ptr + t);i++){
        dPush(dStack.stack[i].val,dStack.stack[i].addr);
    }
    dStack.stack_ptr = t;
dStack.bs_ptr += t + 6;
}
{
    /* Checkpoint address stack */
int t, i;
t = aStack.stack_ptr;
apush(ref_forward_addr);
apush((void*)aStack.base_ptr);
apush((void*)t);
apush((void*)aStack.bs_ptr);
for(i = aStack.bs_ptr; i < (aStack.bs_ptr + t); i++){
apush(aStack.stack[i]);
}

aStack.stack_ptr = t;
aStack.bs_ptr += t + 4;

/* Checkpoint done, return to caller */
goto *temp;
checkpoint_end:

/* Routine to restore a checkpoint */
undo_checkpoint:

cpCount--;
temp = aPop();

/* Restore the stored values */
retVal = dStack.stack[dStack.bs_ptr - 6].val;
line = dStack.stack[dStack.bs_ptr - 5].val;
refBiMid = dStack.stack[dStack.bs_ptr - 4].val;

/* Restore double stack */
dStack.base_ptr = dStack.stack[dStack.bs_ptr - 3].val;
dStack.stack_ptr = dStack.stack[dStack.bs_ptr - 2].val;
dStack.bs_ptr = dStack.stack[dStack.bs_ptr - 1].val;
ref_forward_addr = aStack.stack[aStack.bs_ptr - 4];

/* Restore address stack */
aStack.base_ptr = (int)aStack.stack[aStack.bs_ptr - 3];
aStack.stack_ptr = (int)aStack.stack[aStack.bs_ptr - 2];
aStack.bs_ptr = (int)aStack.stack[aStack.bs_ptr - 1];

/* If we just restored the last checkpoint, put it back */
if(cpCount == 0){
    temp1 = temp;
    aPush(&&fixChkPnt);
    goto checkpoint_start;
fixChkPnt:
    temp = temp1;
}
/* Checkpoint restored, return to caller */
goto *temp;
undo_checkpoint_end:
monitor_end: