ACHIEVING LOGARITHMIC GROWTH OF TEMPORAL AND SPATIAL COMPLEXITY IN REVERSE AUTOMATIC DIFFERENTIATION *

ANDREAS GRIEWANK

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439

In its basic form the reverse mode of automatic differentiation yields gradient vectors at a small multiple of the computational work needed to evaluate the underlying scalar function. The practical applicability of this temporal complexity result, due originally to Linnainmaa, seemed to be severely limited by the fact that the memory requirement of the basic implementation is proportional to the run time, T, of the original evaluation program. It is shown here that, by a recursive scheme related to the multilevel differentiation approach of Volin and Ostrovskii, the growth in both temporal and spatial complexity can be limited to a fixed multiple of $\log(T)$. Other compromises between the run time and memory requirement are possible, so that the reverse mode becomes applicable to computational problems of virtually any size.

KEY WORDS: Gradient, Adjoint, Complexity, Checkpointing, Recursion

1 INTRODUCTION

Many computational problems involve nonlinear vector functions

$$F(x) \quad : \quad \mathbf{R}^{n_d} \mapsto \mathbf{R}^{n_r},$$

which are evaluated by codes written in a high-level computer language such as Fortran or C. Mathematically, this evaluation process can be interpreted as sequence of n elementary transformations

$$s_{i+1} \leftarrow f_i(s_i) \qquad f_i : S \mapsto S \quad , \tag{1}$$

where S denotes a larger vector space of real variables that includes, in particular, the dependent and independent variables of F. Provided that the transformations f_i are all differentiable with Jacobians $f'(s_i)$, the chain rule implies that

$$F'(Qs_0) = P\, f'_{n-1}(s_{n-1}) \cdot f'_{n-2}(s_{n-2}), \ldots, f'_i(s_i), \ldots, f'_0(s_0)\, Q^T,$$

where Q and P are projections onto the domain and range of F, respectively. Even though the individual Jacobians f'_i are likely to be extremely sparse, Speelpenning

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[8] and others have observed that multiplying them together from right to left (i.e., for $i = 0, 1, 2, \ldots, n-1$) may be much less efficient than multiplying them from left to right, especially when $n_r << n_d$. Of particular importance are cases where F' has only one row (i.e., $n_r = 1$) or is multiplied from the left by a row vector $\bar{y}$. Then the product $\bar{x} \equiv \bar{y}F'(x)$ can be obtained as $\bar{x} = \bar{s}_0 Q$, with $\bar{s}_0$ the result of the adjoint recurrence

$$\bar{s}_i \leftarrow \bar{s}_{i+1} f_i'(s_i) \quad for \quad i = n-1, n-2, \ldots, 0$$

starting from $\bar{s}_n = \bar{y}P$. Hence there is only one vector-matrix product associated with each elementary function, so that the effort for evaluating F and $\bar{y}F'$ should be comparable [4]. Provided that the f_i are sufficiently simple, this is indeed the case in terms of the usual operations counts. However, the potential difficulty for this *adjoint*, *top-down*, or *reverse* mode of automatic differentiation is that it seems as though either the Jacobians f_i' or the arguments s_i have to be saved in some form during the original forward sweep (i.e., the execution of the recurrence (1)). This spatial complexity is usually proportional to the temporal complexity of the original evaluation program and may therefore be quite large. Current implementations of the reverse mode [5] typically store 15–20 bytes per arithmetic operation, which may require 30 megabytes of storage for each minute evaluation time on a Sun 3. Even though care can be taken that these large data sets are accessed sequentially rather than randomly, this kind of memory requirement is clearly debilitating on larger problems.

While more economical implementations of the basic scheme can certainly extend the range of applicability of the reverse mode, we describe a modification that destroys the proportionality between the temporal complexity of the original evaluation program and the spatial complexity of the reverse mode. Since we will be able to establish the possibility of merely logarithmic growth, suitable implementations of the reverse mode should allow the automatic evaluation of gradients and other adjoint vectors for problems of virtually any size. In accounting for computational costs, we will try to be as realistic and system independent as possible. The problem we are addressing is closely related to that of logical program reversability, which has attracted some interest in theoretical computer science. Bennett[2] conjectured already in 1973 that a logarithmic growth in the spatial complexity might be achievable. The technique advocated here could also be useful for debugging purposes, where previous states need to be reconstructed by some form of *running the program backward.*

The paper is organized as follows. In Section 2 we develop a convenient model of computations and their complexities on a single-processor machine with a fixed-size core and a sequentially accessed disk file of arbitrary size. In Section 3 we consider the basic reverse method, which minimizes temporal complexity, and its opposite, an even less attractive scheme that needs very little extra storage, but essentially squares the run time. In Section 4 we describe a recursive program for adjoint evaluations and derive bounds for their complexities. In Section 5 we assume that the function evaluation can be broken into a sequence of computational steps that are roughly of equal size in terms of the tape length needed to record them. Under this quite reasonable assumption we develop in Section 6 a binomial partitioning scheme for recursive reversion that minimizes the growth in spatial and temporal complexity. Logarithmic growth is already achievable by a simpler bisection scheme that is used in our current implementation. In Section 7 we map out other feasible combinations of spatial and temporal complexities on a given problem. Section 8 contains some numerical results with this experimental program and discusses the practical ramifications and implementation issues.

2 COMPUTATIONAL FRAMEWORK AND COMPLEXITY MEASURES

For the purposes of our complexity analysis, we assume that the evaluation of the vector function in question is carried out by a sequence of calls to *elementary procedures*

$$f : S \equiv \{0,1\}^R \mapsto S \quad ,$$

where $R \equiv |S| < \infty$ is the *size* of the *state space* S. In other words, the states $s \in S$ are represented by bit vectors of length R on which the procedures f effect certain transformations. In practice, most of these bits will be part of binary representations of integers or floating-point numbers, but that is of no concern here. We refer to the application of f to a particular state $s \in S$ as a *call* to f at s.

Without loss of generality we may assume that all f belong to a given finite set L of elementary procedures. In practice, this library usually includes all binary elementary operations but may also contain more involved procedures, such as basic linear algebra routines or quadratures. Since dimensions and logical flags can be calling parameters, the complexity of these calls f may depend strongly on the state vectors s to which they apply. We will use two complexity measures: a *size* and a *work-vector* denoted by

$$|f|_s \in \mathcal{Z} \quad \text{and} \quad w(f)_s \in \mathbf{R}^o \quad ,$$

respectively. The o components of $w(f)_s$ may account separately for various computational costs (e.g., logical or floating-point operations, and memory accesses to various parts of S). Since each component of w represents a certain amount of execution time on a particular machine, we consider the whole vector as a temporal complexity measure.

The spatial complexity measure $|f|_s$ counts the number of bits one has to *record* on some internal or external medium in order to remember which procedure f was applied and to undo its call at s (i.e., restore s given $f(s)$). For optimizing the recursive differentiation process described in the next section, we will require that a suitable upper bound $|f|_s$ is computable at negligible cost for any particular call f_s. On the other hand, the work vectors $w(f)_s$ need not be known or estimated for that optimization. It is also interesting to note that, at least on a shared-memory machine, the temporal complexity $w(f)_s$ is likely to depend strongly on the number of available processors, but the the spatial complexity $|f|_s$ should be essentially constant.

To indicate that the action of a call to f at s must be recorded, we will use the statement $s \leftarrow \hat{f}(s)$ rather than just $s \leftarrow f(s)$. The cost for this action will be denoted by

$$\hat{w}(f)_s \equiv w(\hat{f})_s \in \mathbf{R}^o$$

We will make sure that all recorded data can be retrieved in a last-in-first-out fashion and, therefore, will refer to the corresponding storage device as the *tape*. Apart from recording some elementary procedures, we will also use the *system* utilities **snapshot(s)** and **retrieve(s)** to copy the current state vector $s \in S$ onto the tape and then to reinitialize S to the snapshot later on. All data are written forward and read backward so that none can be recovered twice. Apart from restricting the maximal length of the tape, one may also be concerned about the total amount of data transfer to and from the tape. Fortunately, we will see that this I/O effort grows at worst proportional to the the amount of arithmetic operations needed fo the total adjoint calculation.

The ratio $w(f)_s/|f|_s$ can be interpreted as a measure of computational intensity, which is quite small for single arithmetic operations but should be rather large for well-designed subroutines. For example, consider an elementary procedure f that multiplies a variable vector by an $m \times n$ matrix that resides as a constant in S. Then only the values of the input vectors and output vectors (plus the dimensions m, n and the addresses of the first matrix and vector elements) need to be recorded on tape. Consequently, the size $|f|_s$ of a call to f is of order $m + n$, whereas its temporal complexity grows like $m \cdot n$, in any sensible measure. In this example the recording on the tape would also be quite cheap, so that $\hat{w}(f)_s \approx w(f)_s$ for typical s. The decomposition of a calculation into elementary procedures in our sense is, of course, not unique, and one may ask how a given problem should be decomposed for the purposes of reverse automatic differentiation. As a general rule, we suggest that the elementary procedures should be computationally intensive, in that $w(f)_s/|f|_s$ is not too small for most s, but at the same time the adjoint $\bar{f}_s$ (discussed below) must be easy to code and evaluate. In the case of the linear transformation discussed above, the corresponding adjoint procedure amounts merely to multiplying an adjoint m vector by the transposed of the constant matrix.

The key ingredient of reverse automatic differentiation is that each call of f at s has a unique *adjoint* procedure call

$$\bar{f}_s : \bar{S} \mapsto \bar{S} \quad ,$$

where the adjoint state space $\bar{S}$ is simply a replica of S. We consider the restoration of s from $f(s)$ as an implicit part of the adjoint call $\bar{f}_s$, so that even logical and integer procedures have nontrivial adjoints. It is assumed that there is a library of adjoint procedures $\bar{f} \in \bar{L}$ that can be invoked at any pair $(s, \bar{s})$. The size $|f|_s$ defined above determines how much information must be retrieved from the tape if one wishes to apply $\bar{f}_s$, assuming S is in the state $f(s)$. Since the adjoints $\bar{f}_s$ are, in fact, linear mappings on $\bar{S}$, we may assume that their work vector

$$\bar{w}(f_s) \equiv w(\bar{f})_s \in \mathbf{R}^o$$

is essentially independent of the particular adjoint state to which they are applied. Moreover, we will assume that the complexity of an adjoint call can be bounded by a multiple of the underlying direct call, so that for some diagonal matrix D of order o,

$$\bar{w}(f)_s \;\leq\; D\, w(f)_s \;\leq\; D\, \hat{w}(f)_s$$

for all $f \in L$ and $s \in S$. If the library L consists solely of binary arithmetic operations and univariate system functions, one can show that, under reasonable conditions on the computer system in use, the scaled identity matrix $D = 5\,I$ is large enough [4].

For the sake of completeness we may formalize the concept of a computer program P as follows. Let P be a numbered set of m instructions each of which consists of two components: a procedure $f \in L$ and a mapping from S to the counter range $[1, \ldots, m]$. The second component lets every instruction nominate its successor, possibly as a function of flags and counters in the state space. Thus we allow for loops and conditional jumps rather than restricting ourselves to straight-line code. There must be terminal instructions that nominate themselves as successors for certain acceptable states s, and one or more entry instructions to begin the execution. All these details have no bearing on our analysis, except that we assume the effort of stepping through the program P to be negligible compared to the cost of manipulating the state spaces $\bar{S}$ and S and the tape. Finally, we assume that the overhead in calling the adjoint procedures $\bar{f} \in \bar{L}$ is also small.

Concluding this section, we summarize the framework developed. There are two randomly accessed *state* spaces S and $\bar{S}$ of fixed size $R = |\bar{S}|$ bits and a strictly sequentially accessed *tape file* of arbitrary length. Procedures $f \in L$ are called to transform a given state $s \in S$ into $f(s) \in S$, an action which may be recorded on the tape with $|f|_s$ bits. Using this record, one can subsequently call on an adjoint procedure $\bar{f}_s$ that recovers s from $f(s)$ on S and effects some (linear) transformation on the adjoint space $\bar{S}$. Associated with unrecorded, recorded and adjoint procedure calls are computational cost vectors $w(f)_s, \hat{w}(f)_s \geq w(f)_s$, and $\bar{w}(f)_s \leq Dw(f_s)$, respectively. The system utilities **snapshot**(s) and **retrieve**(s) transfer a copy of the vector s between the state space and the tape.

3 MINIMIZING EITHER TEMPORAL OR SPATIAL COMPLEXITY

Throughout the remainder of the paper, we consider a sequence of $(n+1)$ states s_i and n procedure calls f_i at s_i that are generated according to

$$s_{i+1} \leftarrow f_i(s_i) \quad for \quad i = 0, 1, \ldots n-2, n-1 \tag{2}$$

from some fixed initial state $s_0 \in S$. Correspondingly, we may define for a fixed terminal $\bar{s}_n \in \bar{S}$ the adjoint states $\bar{s}_i$ by the reverse recurrence

$$\bar{s}_{i-1} \leftarrow \bar{f}_i(\bar{s}_i) \quad for \quad i = n-1, n-2, \ldots, 1, 0 \quad , \tag{3}$$

where we abbreviate $\bar{f}_i \equiv \bar{f}_{s_i}$. The adjoint vector $\bar{s}_0$ is the actual target of the calculation and could, for example, represent the gradient of a scalar function evaluated by the successive calls f_i.

The basic implementation of the reverse mode (i.e., the calculation of $\bar{s}_0$ based on (3)) consists of recording all procedure calls f_i on the tape during the *forward sweep* (2) and then executing the *reverse sweep* reading the tape backwards. The spatial and temporal complexity of the basic scheme is described by

$$|S| + |\bar{S}| + T \;=\; 2\,R + T \tag{4}$$

and

$$\hat{W} + \bar{W} \;\leq\; \hat{W} + DW \;\leq (I + D)\,\hat{W} \quad , \tag{5}$$

where

$$T \equiv \sum_{i=0}^{n-1} |f_i|_{s_i} \quad , \quad W \equiv \sum_{i=0}^{n-1} w(f_i)_{s_i} \tag{6}$$

and

$$\hat{W} \equiv \sum_{i=0}^{n-1} \hat{w}(f_i)_{s_i} \quad , \quad \bar{W} \equiv \sum_{i=0}^{n-1} \bar{w}(f_i)_{s_i} \quad . \tag{7}$$

Thus we have reestablished the by now well-known — but still surprising — result that the basic reverse scheme yields the adjoint vector $\bar{s}_0$ as a function of the pair $(s_0, \bar{s}_n)$ for a fixed multiple of the temporal complexity needed to calculate s_n as a function of s_0. Unfortunately, compared to the original evaluation process, the spatial complexity grows essentially by the factor

$$h \quad \equiv \quad T/R \quad , \tag{8}$$

which will be central to our analysis.

To get an idea of what the ratio h means in practice, let us briefly consider a time-dependent partial differential equation on a square. Using a grid with N nodes in each spatial direction, we may describe the state at any time by several vectors of size N^2 (plus some counters and flags that we may neglect). To progress to the next step, we need one or more copies of each original state variable and a few intermediates, so that R could be something like $50\,N^2$ bytes, assuming reals are stored in four bytes each. Over $M > N$ discrete time steps, the total number floating-point operations would be about $50\,M\,N^2$, which corresponds to a tape size of $T = 1000\,M\,N^2$ bytes. (Here we have assumed that each arithmetic operation is recorded using 20 bytes). The resulting ratio $h = 20\,M$ can obviously be arbitrarily large. In general, h can be thought of as the *height* of the computational graph [1], with R representing its width and T the area (i.e., the total number of nodes). This visualization of the situation is utilized in Fig. 1.

It appears that with regards to the size of h most computations are made up of components that belong to one of the following three classifications:

$h = \mathcal{O}(one)$ For Stationary Structures in Euclidean Plane and Space.
(Circuit Boards, Buildings, Vehicles, Satellites, Molecules, etc.)

$h = \mathcal{O}(T/t)$ For Evolution Calculations over a Period T with Time-steps t.
(Multibody dynamics, Fluid Flow, Weather/Climate, etc.)

$h = \mathcal{O}(its)$ For fixed-point Iterations over a variable number of its steps.
(Adaptive Quadratures, Newton Variants, Gradient Methods, etc.)

On problems in the first class the reverse mode suffers no serious memory growth, because the function evaluation process involves only nearest neighbor interaction between structural components. Therefore the evaluation process can be done essentially in-place, i.e., on the data structures representing the individual components and their connectivity. That applies even for molecules and other multi-bosy systems with long-range interactions, provided these enter additively into the overall energy. Then the individual energy contributions can be evaluated and differentiated separately, and their gradients may be accumulated immediately without any need to keep a global tape.

All problems of the first class turn into an element of the second if one wishes to study the evolution of the structure in question as a function of time. In robot or satellite design, weather data assimilation and other optimal control problems one needs the gradient of some performance measures or fitting functions with respect to model parameters, boundary conditions, and controls. In control theory it is well known that these gradients can be obtained with low temporal complexity by integrating the linear co-state equation backward in time. The close relation between this well-established technique and reverse automatic differentiation was analyzed in Evtushenko's contribution to the proceedings [3] of the first SIAM Workshop on the Automatic Differentiation of Algorithms. The same volume contains several papers by leading researchers from Meteorology and Oceanography, where adjoint models are in regular use. Despite great efforts to hand-code the adjoints as economically as possible, the storage requirement prevents their application to operational models with fine resolution.

Problems in the third class are characterized by numerically induced iterations, which may be interpreted as pseudo-time evolutions. In the case of implicit functions, quadratures, and other problems with a well defined mathematical structure their characteristics can often be exploited to obtain derivatives quite economically. However, this requires a lot of insight and intervention by the user and the relevant program fragments may be hard to isolate in a larger code. Therefore it is desirable that the reverse mode can be implemented automatically such that numerical iterations in some part of the evaluation process do not let the tape grow unacceptably long.

The maximal tape size T is likely to be proportional to some norm of the work vector W, unless the key elementary procedures f are composites such as matrix-vector products with large computational intensities $w(f)_s/|f|_s$. We have implicitly

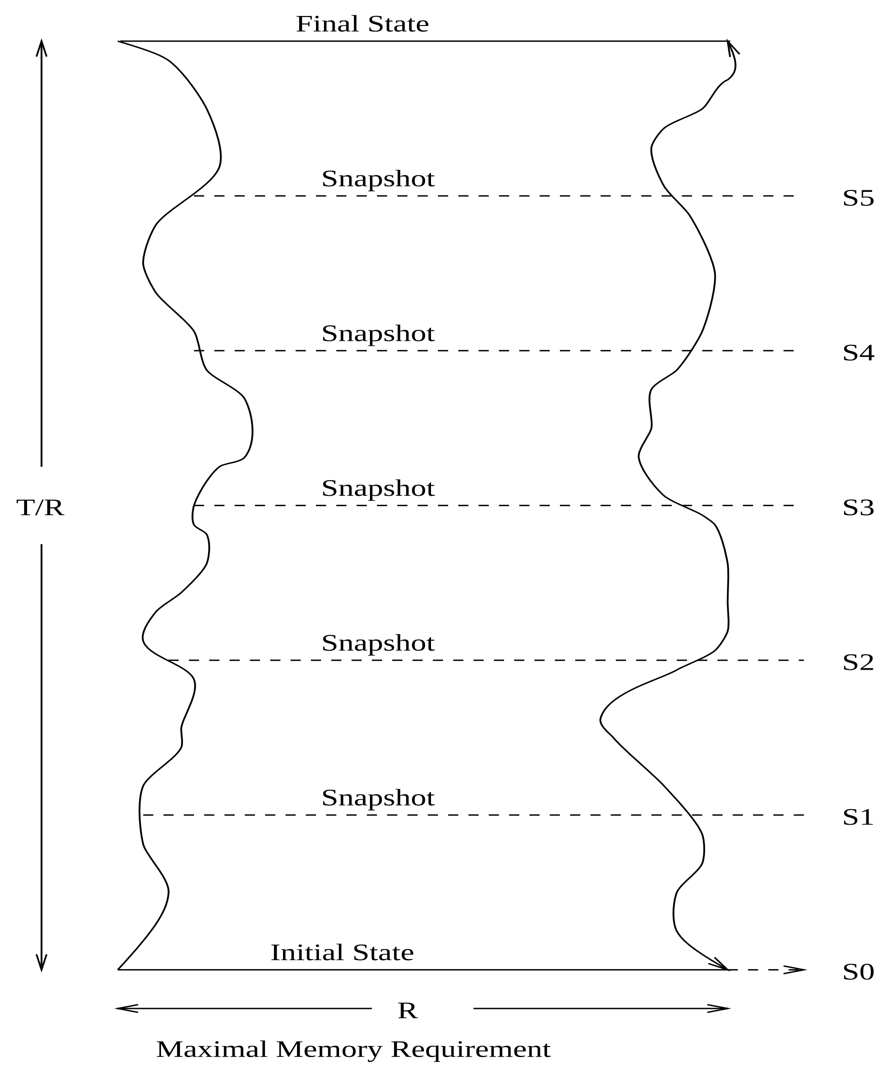

FIGURE 1: Computational Graph with vertical Time and horizontal Space axis

restricted the complexity of the elementary procedures $f \in L$ by requiring that they have adjoint procedures $\bar{f}$ that can be coded easily for inclusion in the adjoint library $\bar{L}$. Volin and Ostrovskii [9] suggested recursively treating procedures as programs with their own work space and evaluating their adjoints by performing forward and reverse sweeps within these subprograms. We prefer to consider the elementary procedures as lowest computational units and refer to the maximal size

$$G \quad \equiv \quad \max\{\, |f_i| \;:\; 0 \leq i < n \,\}$$

as the granularity of the calculation. Like the maximal tape size T the granularity is dependent on the initial state s_0, but by our assumptions both integers can be computed during a preliminary forward sweep.

The basic method described above has optimal temporal complexity but requires a potentially excessive amount of storage. At the opposite extreme one can minimize spatial complexity at the expense of temporal complexity by the following simple scheme. Suppose we have a tape file of size at least $S + G$. Then we can take a snapshot of s_0 and execute the forward sweep without recording until reaching s_{n-1}. The final call $\hat{f}_{n-1}$ can now be executed and recorded so that the application of $\bar{f}_{n-1}$ yields $\bar{s}_{n-1}$ from $\bar{s}_n$. We may then repeat the process by reinitializing S to s_0 and then executing another, mostly unrecorded, forward sweep. Since one adjoint call can be applied each time, $\bar{s}_0$ must be reached after at most n (partial) forward sweeps. Even if we record several smaller calls together, the number of forward sweeps will be at least $\tilde{n} \equiv \lceil T/G \rceil$, so that the resulting temporal complexity is of order $\tilde{n}W$, provided the work is reasonably evenly distributed between early and late procedure calls.

4 RECURSIVE ADJOINT EVALUATION METHOD

An $\tilde{n}$-fold increase in the spatial complexity is clearly unacceptable even for comparatively small problems. However, the basic approach of repeating forward sweeps from a previously saved state is quite viable. Rather than returning to the initial state every time, we will instead take snapshots at checkpoints that are carefully chosen to minimize the number of times a particular procedure has to be evaluated. The basic idea is depicted in Fig. 1, where the checkpoints are spaced evenly. By optimizing the number of snapshots, one can reduce the spatial complexity to $\mathcal{O}(\sqrt{h}R)$. This very significant saving in memory requires only one extra sweep through each segment of the calculation sequence, so that the work estimate (5) is increased by W. The work W for an unrecorded forward is always smaller than the effort $\hat{W}$ for a recorded sweep, which in turn is typically dominated by the cost $\bar{W}$ of the full adjoint sweep. Therefore, we can expect that a few additional unrecorded sweeps will have little impact on the overall temporal complexity. Rather than using just one level of snapshots as depicted in Fig. 2, one can apply the same technique recursively to the horizontal slices of the graph.

Provided $s \in S$ and $\bar{s} \in \bar{S}$ have been initialized to the given s_0 and $\bar{s}_0$, the reverse sweep can be performed in pieces by a call **treeverse**$(0, 0, n)$, where the recursive procedure **treeverse** is defined as the informal program listed in Fig. 1.

treeverse($base$,$start$,$finis$)
  if $start > base$ **snapshot**(s)
  $s \leftarrow f_i(s)$ **for** $i = base, \ldots, start-1$
  while $base << finis$
    pick $kidstart \in (start, finis)$
    treeverse($start, kidstart, finis$)
    $finis = kidstart$
  $s \leftarrow \hat{f}_i(s)$ for $i = start, \ldots, finis-1$
  $\bar{s} \leftarrow \bar{f}_i(\bar{s})$ for $i = finis-1, \ldots, start$
  if $start > base$ **retrieve**(s)
return

FIGURE 2: Recursive Adjoint Calculation Routine

The name **treeverse** was chosen to reflect the fact that we are traversing a tree of recursive calls to perform a reverse sweep on the given computational sequence. The index $base$ indicates what state of the forward calculation has been reached and the index bracket $[start, finis]$ tells the invocation of **treeverse** which subrange of the calculation it has to "cover." Here, to "cover" means to move the adjoint state back from $\bar{s} = \bar{s}_{finis}$ to $\bar{s}_{start}$. To this end **treeverse** first advances from s_{base} to s_{start}, and then calls up several children to reduce the current target $finis$ until it is deemed close enough to $start$ that the remaining stretch from s_{start} to s_{finis} can be recorded and reversed directly. Here the notion *close enough* is the negation of the relation $<<$ that will be defined more specifically later on. Except when $base = start$ the initial state s_{base} is saved and later restored just before control returns to the calling program. The recursion must terminate since the child's start $kidstart$ must always lie inside the current bracket $(start, finis)$, and we have tacitly assumed that the test $start << finis$ can hold only if the width $finis - start$ is greater than one. Under these two simple conditions on the *while* condition and the choice of $kidstart$, the recursive procedure must terminate with the correct result $\bar{s} = \bar{s}_0$.

Sometimes the calculations performed by the functions f_i for $base \leq i < finis$ may be known in advance to take place in a conveniently addressed subset of the state space S. In this case we need only save and restore that subspace at the beginning and end of the call to **treeverse**, repectively. In this way our recursive formalism can be modified to include Volin and Ostrovskii's multilevel differentiation approach [9]. However, in this paper we assume that the simplicity and convenience of taking snapshots of the whole state space outweigh the savings in storage that might be achieved by a more localized approach. As we will briefly discuss in the final section, the operating system may automatically provide this localization through its virtual memory manager.

If one imposes an a priori bound on the size of the recordings, the maximal tape length becomes essentially equal to R times the maximal depth of the recursion, which we will denote by d. Hence we obtain the spatial complexity bound

$$T_d \quad \leq \quad G + (d + r)\, R \quad , \tag{9}$$

where the *constant* term

$$r \;\equiv\; \frac{1}{R} \max \left\{ \sum_{i=start}^{finis} |f_i| - G \right\}$$

represents the maximal size of any recording measured in units of R.

As with the spatial complexity, we wish to bound the temporal complexity relative to the effort of executing the original forward calculation. For this purpose let us consider how often any particular elementary procedure f_i is evaluated during one of the partial forward advances. By inspection of the informal program for **treeverse**, we see that this procedure occurs exactly as often as the particular index i is contained in the interval $[base, start)$ for a call to **treeverse**. Clearly i is contained in the intervals $[start, kidstart)$ for all children of the particular call that eventually records f_i. For that unique parent i must be contained in the interval $[start, finis)$ at the time of its call. This interval is traversed by all older siblings of the parent, as their $kidstart$ values exceed the value of $finis$ at the time when the parent itself was born. Since the grandparents' interval $[start, finis)$ contains that of the parent, we conclude that the number of all older siblings of all direct ancestors must be added as well. While this argument seems rather complicated, we will see below that the number of forward sweeps through any part of the calculation sequence can be computed quite easily.

5 SEGMENTATION INTO COMPUTATIONAL STEPS

The spatial bound (9) is not optimal, because one may use shorter recordings at larger depth. However, this does not help much, and a uniform recording size has other advantages. Given r as a parameter, we will therefore partition the original sequence of elementary procedures f_i into segments called *computational steps*

$$F_j \quad \equiv \quad [f_{i_j}, f_{i_j+1}, \ldots, f_{i_{j+1}-1}] \quad \text{for} \quad j = 0, \ldots, \eta - 1$$

Here the indices $i_j \leq n$ are defined as large as possible, subject to the constraint that

$$\sum_{k<i_j} |f_k| \quad \leq \quad j\, R\, r \quad . \tag{10}$$

This definition implies that the total number of computational steps is given by

$$\eta_r \quad \equiv \quad \lceil T/(r\, R) \rceil$$

and that for all $j \leq \eta_r$

$$|F_j| \equiv \sum_{i_j \leq k < i_{j+1}} |f_k| \quad \leq \quad r\,R + G \quad .$$

Our analysis will reveal that good choices of r are at least sizable fractions and often greater than one, so that $r\ R$ should be quite large compared to the granularity. Consequently, we can expect that the sizes $|F_j| \geq r\,R - G$ are very close to their average $r\,R$. To indicate that the elements of F_i are applied with or without recording, and similarly to represent the application of their adjoints in reverse order, we use the statements

$$s \leftarrow F_j(s) \quad , \quad s \leftarrow \hat{F}_j(s) \quad , \quad \text{and} \quad \bar{s} \leftarrow \bar{F}_j(\bar{s}) \quad ,$$

respectively. Hence, we have coarsened the original sequence of elementary procedures f_i into a sequence of computational steps F_j of nearly uniform size $|F_j| \approx r\ R$. Here *size* represents again the length of the recording needed to prepare the ground for the corresponding adjoint step at a later time.

In explicitly time-dependent problems the F_j can be defined naturally as a subsequence of several time steps. While imposing spatial uniformity on the computational steps we will not make any assumption regarding their temporal complexity. Even if the computational effort per step varies widely, we can bound the overall temporal complexity of the adjoint calculation by limiting the number of times any one of the steps is (re)evaluated. Therefore, we can even allow for the possibility that the individual computational steps are evaluated with varying degrees of concurrency on a parallel machine. In that case the steps must be separated by synchronization boundaries, and a shared memory should be established at least in the virtual sense.

At the level of the computational steps, we will replace the original counters *base*, *start*, *kidstart*, and *finis* by the Greek indices β, σ, κ, and ϕ, respectively. These single-character names will also be more convenient in the subsequent mathematical analysis. To keep track of the depth, we will use a counter δ that is decremented from its initial value d at the beginning of each call. Similarly we will use a counter τ that is decremented from its initial value t every time the computational steps F_j with j in the current range $[\sigma, \phi)$ are executed without recording. The role of δ and τ may be better understood if one rearranges the calling tree of **treeverse** as a binary tree. To this end each call to **treeverse** generates a single left child, and the parent duplicates itself as the right child. Then the values $(d - \delta)$ and $(t - \tau)$ in a particular instance of **treeverse** count how many left and right children occur in its line of ancestry from the root, i.e., the top-level call.

In the absence of any other information regarding the nature of the computational steps, we must choose the new kidstart version κ by a partition function of the form

$$\kappa = \mathbf{mid}(\delta, \tau, \sigma, \phi) \quad .$$

To signal that no more children are required or that the limits d and t are about to be violated because either δ or τ have been reduced to zero, we use the special

```
treeverse(δ, τ, β, σ, φ)
    if σ > β
        δ = δ − 1
        snapshot(s)
        s ← F_j(s) for j = β, ..., σ − 1
    while κ = mid(δ, τ, σ, φ) < φ
        treeverse(δ, τ, σ, κ, φ)
        τ = τ − 1
        φ = κ
    if φ − σ > 1 exit("treeverse fails")
    s ← F̂_σ(s)
    s̄ ← F̄_σ(s̄)
    if σ > β retrieve(s)
return
```

FIGURE 3: Recursive Adjoints by Range Partitioning

values

$$\kappa = \phi \quad \text{if} \quad \phi = \sigma + 1 \text{ or } \delta\,\tau = 0 \quad .$$

Now we can finally use the top-level call $\mathbf{treeverse}(d, t, 0, 0, \eta_r)$ with the routine **treeverse** reprogrammed as listed in Fig. 3.

Unless either δ or τ is reduced to zero prematurely, the reverse sweep is completed and satisfies the temporal complexity bound

$$W_t \equiv \hat{W} + \bar{W} + t\,W \quad . \tag{11}$$

Here W, $\hat{W}$, and $\bar{W}$ are as defined in (6) and (7). By comparison with (9) we see that the two parameters $d + r$ and t directly and independently determine upper bounds on the spatial and temporal complexity, respectively. Naturally the limits t and d as well as the recording size r cannot be chosen arbitrarily, but they must be large enough to allow the recursion to terminate successfully for a suitable definition of the partition function **mid**. In the following section we will derive the partition function that is optimal under our complexity assumptions. Since that derivation is complex, let us first consider a simple bisection scheme, which is sufficient to yield the logarithmic growth alluded to in the title. Suppose we define κ simply as the midpoint

$$\kappa = \mathbf{mid}(\delta, \tau, \sigma, \phi) \;\equiv\; \lceil(\sigma + \phi)/2\rceil \quad . \tag{12}$$

This choice means that the width $\phi - \sigma$ of the range to be covered is halved at least once at each level. More precisely, at any particular instance in the calling tree it will have been halved $(d - \delta) + (t - \tau)$ times, where d and t are the actual parameters of the top-level call. Consequently the choice

$$t \;=\; d \;=\; \lceil \log_2 \eta_r \rceil$$

ensures that the recursive procedure must terminate regularly. For simplicity we may define the computational steps according to (10) with $r = 1$ so that their recording requires no more than R bits. Then we have $\eta \approx h = T/R$ so that by (9) and (11)

$$\frac{(T_d - G - r\,R)}{R} \approx \frac{(W_t - \hat{W} - \bar{W})}{W} \leq \log_2\left(\frac{T}{R}\right) \quad .$$

In other words, for this particular partitioning strategy the increase in both temporal and spatial complexity is roughly equal to the logarithm base two of the number of computational steps. As we will show in the next section, this penalty factor can be reduced by a factor of two, and one can decrease the storage requirement further by a more careful choice of the stepsize r. Until the end of the next section the size r plays no role at all, and only the total number η_r of computational steps is important.

6 OPTIMAL COMPLEXITY BY BINOMIAL PARTITIONING.

Given the current limits δ and τ on the number of additional generations and the number of extra forward sweeps, the value κ =**mid**$(\ldots)$ determines a split of the current range $[\sigma, \phi]$ into the two subranges $[\sigma, \kappa]$ and $[\kappa, \phi]$. Since the cost penalty for covering the second subrange by the nested call to **reverse** is bounded in terms of the parameters δ and τ, one might as well choose the width $\kappa - \phi$ as large as possible, subject to the condition that the recursion can still be completed. In this way the task of constructing an optimal function **mid** can be interpreted as a dynamic programming problem.

For given $\tau \geq 0$ and $\delta \geq 0$ let us denote by $\eta(\tau, \delta)$ the maximal number of computational steps that can be covered by **treeverse** using any possible choice of **mid**. At this stage we have to allow for the theoretical possibility that some $\eta(\tau, \delta)$ are infinite, which would mean that arbitrarily long computations could be inverted at a fixed increase in computational complexity. Unfortunately, that is not the case, as we can see from the following inductive argument. By inspection of the second program above, we have the inequality

$$\eta(\delta, \tau) \quad \leq \quad 1 + \sum_{\alpha=1}^{\tau} \eta(\delta - 1, \alpha) \quad . \tag{13}$$

The summation should be interpreted backwards in that the last term $\eta(\delta - 1, \tau)$ is the width of the range covered by the first child of the current node. Subsequently τ is reduced by one and the next child is called with the parameters $\delta - 1, \tau - 1$, and so on until τ has been reduced to zero. At that stage no further children can be generated and the current node records and reverts a single step, which is reflected by the leading 1 on the right-hand side of (13). In calls where the first parameter δ has already been reduced to zero, not a single child can be generated, as that would require another snapshot being taken. Hence we find that

$$1 \;=\; \eta(\delta, 0) \;=\; \eta(0, \tau) \quad \text{for all} \quad \delta, \tau \geq 0 \quad . \tag{14}$$

Because of (13) each $\eta(\delta,\tau)$ is bounded by sums of the limiting unit values above so that all of them must be finite. In fact since the $\eta(\delta,\tau)$ are supposed to be as large as possible, we might as well define them recursively by (13) with $\leq$ replaced by an equal sign. By comparison with the summation formula for binomial coefficients we obtain the explicit formula

$$\eta(\delta,\tau) \equiv \begin{pmatrix} \delta+\tau \\ \tau \end{pmatrix} \equiv \frac{(\delta+\tau)(\delta+\tau-1)\cdots(\delta+1)}{\tau!} \quad , \tag{15}$$

which satisfies both the marginal conditions (14) and the relation (13) as an equality. The corresponding partition function is given by the convex combination

$$\kappa = \mathbf{mid}(\delta,\tau,\sigma,\phi) \quad \equiv \quad \left\lceil \frac{\delta\cdot\sigma+\tau\cdot\phi}{(\tau+\delta)} \right\rceil \quad . \tag{16}$$

To verify this assertion, one may simply check that if the current range is maximal in that

$$(\phi-\sigma) = \eta(\delta,\tau) \quad ,$$

then the value of κ defined above ensures for the right subrange

$$(\phi-\kappa) = \eta(\delta-1,\tau)$$

and for the left subrange

$$(\kappa-\sigma) = \eta(\delta,\tau-1) \quad .$$

The decrement in τ for the left subrange makes sense since the right subrange is covered first, and in doing so **treeverse** once more marches through the left subrange. The decrement of δ for the right subrange makes sense since each call entails another snapshot. Note that the last three equations are consistent with the addition formula for binomial coefficients. To conclude this section, we can now summarize the central result of this paper as follows.

THEOREM 6.1. *A sequence of η computational steps F_j can be reverted using up to d snapshots and (re)evaluating each F_j without recording at most t times if and only if*

$$\eta \quad \leq \quad \eta(d,t) \;=\; (d+t)!/(d!\,t!) \quad .$$

In addition, this procedure involves one recorded evaluation $\hat{F}_j$ and one adjoint evaluation $\bar{F}_j$ of each computational step. The spatial and temporal complexity of the complete reverse calculation is bounded by

$$T_d \equiv G + (d+r)\,R \quad \text{and} \quad W_t \equiv \hat{W} + \bar{W} + t\,W \quad ,$$

respectively. Here $R = |S|$ and $W, \hat{W}, \bar{W}$ are defined in (6) and (7).

The fact that the number of snapshots d and the number of extra passes t enter in a completely symmetric fashion into the maximal number of computational steps η is aesthetically quite pleasing. According to Stirling's formula we have almost exactly

$$\eta(t,d) \;\approx\; \frac{1}{\sqrt{(2\pi)}}\cdot\left[1+\frac{d}{t}\right]^t \left[1+\frac{t}{d}\right]^d \sqrt{\left(\frac{1}{d}+\frac{1}{t}\right)} \quad , \tag{17}$$

so that for given η one may choose for example equal parameters

$$t \;=\; d \;\approx\; \log_4(\eta_r) \approx log_4[T/(Rr)] \quad .$$

Thus we see that this particular optimized scheme is about twice as efficient as the simpler bisection scheme, whose complexity is already logarithmic in $h = T/R$.

If one fixes either parameter d or t at a certain level, the other parameter grows like a fractional power of the number of steps to be covered. More specifically, it follows again from (17) that for fixed positive d or t

$$t = \mathcal{O}\left(\sqrt[d]{h/r}\right) \quad \text{or} \quad d = \mathcal{O}\left(\sqrt[t]{h/r}\right) \quad ,$$

respectively. Which one of the feasible combinations (d, t) and r should be selected for given h depends, of course, very much on the computing environment and the user's priorities. However, as we have noticed before, a small value of t, say 4, is unlikely to increase the run time by much in comparison to the basic $t = 0$ scheme. At the same time this choice would reduce the memory requirement for the adjoint calculation to the fourth root of the height h times the size of the state space.

Figure 2 depicts the optimal schedule for a sequence of $\eta(3, 5) = 56$ computational steps. Each horizontal line at level $\sigma + 1$ from the bottom represents an instantiation of **treeverse**. The vertical lines emanating from the horizontals represent calls to their children, whose level is given by the current value of $\kappa + 1$. The slanted lines connect groups of siblings who are called by their parents to revert one computational step at a time without generating any more children of their own. The recursive nature of the whole process is clearly discernible. An alternative interpretation of Fig. 1 is that it represents a schedule for a system of elevators in a high-rise building with 56 floors. In that situation $t = 5$ represents the number of shafts, and $d = 3$ bounds the number of times any rider traveling from the ground level to any one of the 58 floors has to switch elevators or merely halts at a scheduled stop.

7 SELECTING THE SIZE OF COMPUTATIONAL STEPS

In this section we have so far assumed that the recording size r had been selected a priori. On the other hand, we found that the complexity bounds d and t are optimally exploited if the number of computational steps is exactly equal to the corresponding binomial $\eta(d, t)$. Hence it makes sense to define r as

$$r \quad \equiv \quad h/\eta(d, t),$$

which is always possible even if $d = 0 = t$ and thus $\eta(0, 0) = 1$. This extreme choice records the whole calculation as a single computational step and corresponds therefore to the basic version of reverse automatic differentiation. In general, we see from (9) that the total length of the tape is proportional to the sum $(d + r)$. Hence

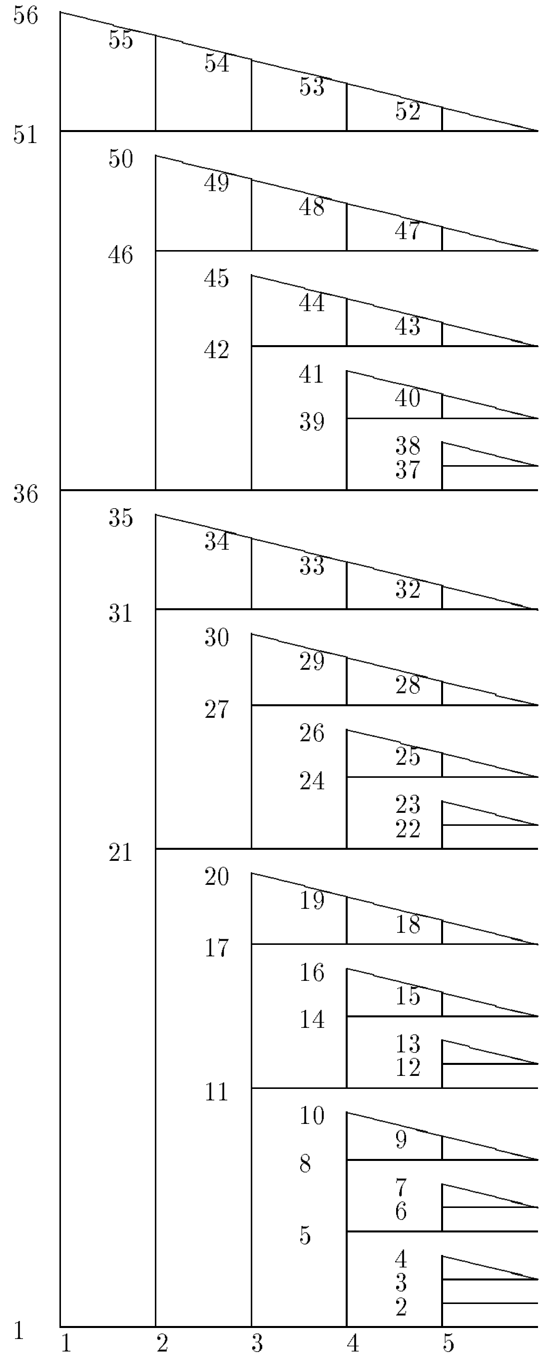

FIGURE 4: Optimal Schedule for 5 Sweeps and 3 Snapshots

it makes sense to select d such that it minimizes the spatial complexity multiplier

$$d + r \quad = \quad d + h/\eta(d,t) \tag{18}$$

for fixed t. An elementary examination shows that this objective is attained at the unique $d \geq 0$ for which

$$d \cdot (d+1) \cdots (d+t) \;\leq\; t \cdot h \cdot t! \;\leq\; (d+1) \cdots (d+t+1) \quad . \tag{19}$$

The corresponding recording size satisfies

$$0 \;\leq\; r - d/t \;\leq\; 1 + 1/t \quad , \tag{20}$$

so that we have approximately $r \approx d/t$, and the maximal length of the tape is about

$$T_d \;\approx\; d(1+1/t)\, R \;\approx\; r(1+t)\, R \quad .$$

This means that ideally the fraction $1/(1+t)$ of the available tape length should be allocated to the recordings, with the remaining larger fraction $t/(1+t)$ being used for snapshots.

Since each call to **treeverse** reverts one computational step their total number is exactly $eta_r = \lceil /R/r \rceil$. Except for the top-level instance each of them calls the system routines **snapshot(s)** and **retrieve(s)** once so that the total bit tranfer between the state space and the tape is bounded by

$$2\,[T + (\eta_r - 1)\, R] \;<\; 2\,T\,(1 + 1/r) \;<\; 2\,T\,(1 + t/d) \quad ,$$

where the last inequality follows from (20). Thus we see that even when there is only room for one state space copy on the tape, i.e., $d = 1$, the total bit transfer is at most $(1+t)$ times the minimal amound $2\,T$, which is always needed to record and revert each computational step once. Consequently our main complexity result remains true even if the total bit transfer is included into the temporal complexity measure W_t.

Each computer program for the evaluation of the vector function in question has a well defined height $h = T/R$. If it is not known from evaluations at previous arguments the height can be determined in a preliminary forward sweep. Alternatively one might devise an adaptive scheme that dynamically allocates snapshots and revises the schedule when the evaluation runs longer or shorter than expected. This is a field for future research and development. If h is known one can compute a profile of pareto optimal space-time trade-offs by minimizing $d + r$ for $t = 0, 1, \ldots$ The resulting contours for $h = 10, 100, \ldots, 10,000,000$ are plotted in Fig. 5. Along the diagonal $t = d$ the growth in both complexity factors is clearly logarithmic, whereas either of them explodes in a hyperbolic fashion near the axes. Key observations are that adjoints of calculations involving 100,000 or 1,000,000 computational steps can be obtained at a cost increase by a factor of 10 or 12, respectively. Here cost accounts for spatial and temporal complexity, including the bit transfer between the state space and tape. While these cost increases are nonnegligible one obtains full sensitivity information for one key objective or response with respect to all

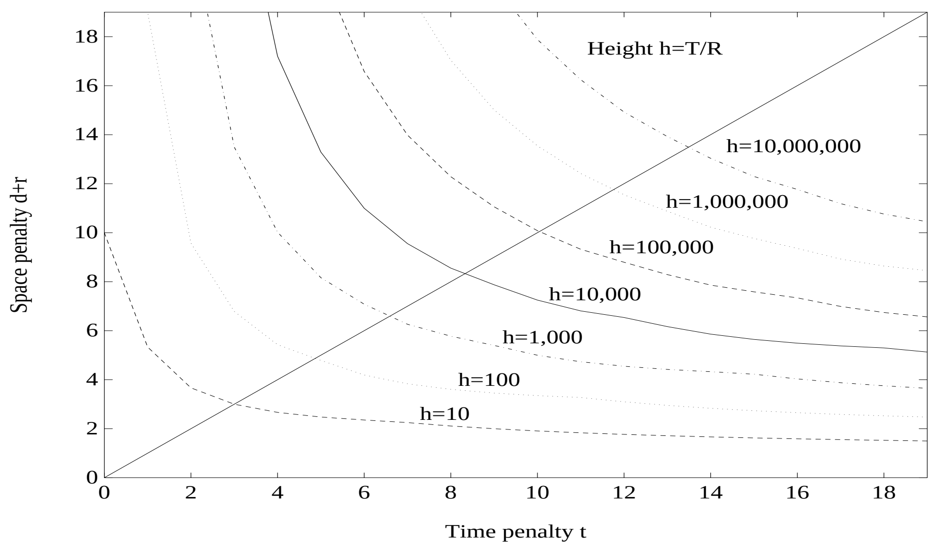

FIGURE 5: Feasible space-time Combinations for given Height h

input variables, parameters, and controls. Currently, this information cannot be obtained any other way.

8 IMPLEMENTATION QUESTIONS AND DISCUSSION

The logarithmic complexity growth of the recursive method described and analyzed in this paper has been verified by an experimental implementation. For this purpose our C++ package ADOL-C [5] was modified using the forking and piping facilities of UNIX System V. The tree of *calls* to **treeverse** was implemented as a tree of processes with each child being spawned by a fork, which generates a full duplicate of the parent's environment. Therefore the child inherits all information accessible to the parent, who halts its own calculations until the child signals the completion of its task. This information is passed back by sending the updated adjoint values through a pipe that has been set up by the parent for that particular purpose. The parent then either sends out another child or records and reverts its own computational step before returning control to the grandparent. The computational steps are simply defined as a sequence of elementary operations, whose recording in the buffer for the tape takes up a certain number of bytes. There is no attempt to estimate or utilize the size of the state space in use. Instead the user picks desired bounds d, t, but neither δ, τ are never reduced below 1. Consequently the binomial partitioning reduces to the bisection scheme when the combination d, t turns out to be infeasible.

For example, the calculation of a 10×10 determinant using Legendre's rule involves $10! = 3,628,800$ multiplications and additions or subtractions. Since the determinant was computed using a recursive function call, many assignments and so-called death notices had to be recorded. These overhead operations brought the total length of the tape to almost $T = 814$ megabytes, corresponding to nearly a million computational steps of one kilobyte size. As predicted by the theory this problem could be solved for the combination $d = t = 12$. The computing time was extensive, because all forward sweeps were performed with recording for programming simplicity. An efficient version is currently being developed.

The UNIX implementation sketched above is surprisingly simple and even elegant. To limit the size of the state space, one should relegate the evaluation of the function and its adjoint to a separate process that contains only the data that are actually needed for this purpose. Often this is only a comparatively small part in a larger computational environment. In case of 10×10 determinant, there were only 123 live variables, so that the data set that must be duplicated by fork could be very small indeed. Another crucial question is how the operating system handles pages that are paged out at the time of a fork. It appears that current implementations immediately duplicate all these pages so that they may generate many identical copies of pages that may never be touched by the function evaluation. The UNIX documentation on fork promises for later releases a *copy on write* system, where pages are duplicated only when either child or parent processes actually start changing them. This localization to the active areas of the state space could make checkpointing by forking quite efficient.

For the sake of user convenience and computational efficiency, it would be ideal if reverse automatic differentiation were implemented at the compiler level. Some compiler directives or new language constructs to identify independent and dependent variables as well as critical program sections are needed. Almost everything else could be handled by the compiler, if not the operating system. After profiling an evaluation routine in terms of computational cost and memory usage at various stages of the execution, the operating system could suggest a checkpoint schedule that is more or less optimal under the particular circumstances.

ACKNOWLEDGEMENTS

The author is indebted to Ted Gaunt and Chuck Tyner for their invaluable help in implementing the proposed method.

REFERENCES

1. F. L. Bauer, *Computational graphs and rounding errors.* SINUM, Vol. 11, No. 1 (1974), pp. 87–96.
2. C. H. Bennett, *Logical Reversability of Computation*, IBM Journal of Research and Development, Vol. 17 (1973), pp. 525–532.
3. Yu. G. Evtushenko, *Automatic differentiation viewed*, in: Automatic Differentiation of Algorithm: Theory, Implementation, and Application, A. Griewwank and G. F. Corliss, eds., SIAM, Philadelphia, 1991.

4. A. Griewank, *On automatic differentiation*, in: Mathematical Programming: Recent Developments and Applications, ed. M. Iri and K. Tanabe, Kluwer Academic Publishers, Tokyo, pp. 83–108, 1989.
5. A. Griewank, D. Juedes, and J. Srinivasan, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, Preprint MCS-180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1990. To appear in TOMS.
6. M. Iri, T. Tsuchiya, and M. Hoshi, *Automatic computation of partial derivatives and rounding error estimates with applications to large-scale systems of nonlinear equations*, Journal of Computational and Applied Mathematics, 24 (1988), pp. 365–392.
7. S. Linnainmaa, *Taylor expansion of the accumulated rounding error*, BIT, 16 (1976), pp. 146–160.
8. B. Speelpenning, "Compiling Fast Partial Derivatives of Functions Given by Algorithms," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980.
9. Yu. M. Volin and G. M. Ostrovskii, *Automatic computation of derivatives with the use of the multilevel differentiation technique*, Computers and Mathematics with Applications, Vol. 11, No. 11 (1985), pp. 1099–1114.