REDUCTION OF THE RANDOM ACCESS MEMORY SIZE IN ADJOINT ALGORITHMIC DIFFERENTIATION BY OVERLOADING

UWE NAUMANN*

 \mathbf{Key} $\mathbf{words.}$ algorithmic differentiation, adjoint, overloading, random access memory, bandwidth

Abstract. Adjoint algorithmic differentiation by operator and function overloading is based on the interpretation of directed acyclic graphs resulting from evaluations of numerical simulation programs. The size of the computer system memory required to store the graph grows proportional to the number of floating-point operations executed by the underlying program. It quickly exceeds the available memory resources. Naive adjoint algorithmic differentiation often becomes infeasible except for relatively simple numerical simulations.

Access to the data associated with the graph can be classified as sequential and random. The latter refers to memory access patterns defined by the adjacency relationship between vertices within the graph. Sequentially accessed data can be decomposed into blocks. The blocks can be streamed across the system memory hierarchy thus extending the amount of available memory, for example, to hard discs. Asynchronous i/o can help to mitigate the increased cost due to accesses to slower memory. Much larger problem instances can thus be solved without resorting to technically challenging user intervention such as checkpointing. Randomly accessed data should not have to be decomposed. Its block-wise streaming is likely to yield a substantial overhead in computational cost due to data accesses across blocks. Consequently, the size of the randomly accessed memory required by an adjoint should be kept minimal in order to eliminate the need for decomposition. We propose a combination of dedicated memory for adjoint L-values with the exploitation of remainder bandwidth as a possible solution. Test results indicate significant savings in random access memory size while preserving overall computational efficiency.

1. Introduction. Building on prior work in [18] we consider a given implementation of a differentiable multivariate vector function $F: \mathbb{R}^n \to \mathbb{R}^m : \mathbf{x} \mapsto \mathbf{y} = F(\mathbf{x})$ over the real numbers \mathbb{R} as a differentiable computer program over floating-point numbers [1] referred to as the primal program. Note that differentiability of a function does not imply (algorithmic) differentiability of the given implementation. A prime example is a table lookup of a dedicated value of a differentiable function yielding a vanishing algorithmic derivative as for example in **if** (x==0) y=0; **else** $y=\sin(x)$; (Algorithmic) Differentiability of the given implementation implies differentiability of the function due to the chain rule of differentiation. Algorithmic differentiation (AD) [12, 17] yields a variety of implementations of the transformation of the given differentiable program into a program for evaluating its derivatives. We focus on AD by operator and function overloading as supported, for example, by C++. The following discussion is presented in the context of a serial evaluation of F as well as of its (adjoint) derivatives. Generalization to parallel scenarios based on shared [2, 7] or distributed [19, 22] memory architectures or on massively parallel accelerators [11, 16] is the subject of ongoing research.

The adjoint [6] of F becomes

$$(1.1) \bar{F}: \mathbb{R}^n \times \mathbb{R}^{1 \times m} \to \mathbb{R}^{1 \times n}: (\mathbf{x}, \bar{\mathbf{y}}) \mapsto \bar{\mathbf{x}} = \bar{F}(\mathbf{x}, \bar{\mathbf{y}}) \equiv \bar{\mathbf{y}} \cdot F' ,$$

where

$$F' = F'(\mathbf{x}) \equiv \frac{dF}{d\mathbf{x}}(\mathbf{x}) \in \mathbb{R}^{m \times n}$$

denotes the Jacobian of F at the given point \mathbf{x} . Equation (1.1) is evaluated by the

^{*}Informatik 12: Software and Tools for Computational Engineering, RWTH Aachen University, Germany. naumann@stce.rwth-aachen.de

adjoint program resulting from the application of adjoint AD to the given implementation of F.

Without loss of generality, the following discussion is based on the assumption of a single scope primal. Effects due to allocation and deallocation of program variables are not accounted for. Additional technical issues due to specifics of the programming language as well as due to the custom design of the AD tool at hand need to be taken into account when integrating this paper's ideas into state-of-the-art AD software tools. Examples throughout this paper as well the reference implementation (see Section 6) are written in C++.

Ordered sets $V = \{0, \ldots, n+q-1\}$ (indexes of variables), $X = \{0, \ldots, n-1\}$ (indexes of independent input variables) and $Y \subseteq V$, |Y| = m (indexes of dependent output variables) are induced by the primal program for given values of the inputs. The primal program is regarded as a composition of elemental functions (also: elementals) φ_j evaluated as a single assignment code (each variable v_j is written once as the result of φ_j)

C++-style incrementation k++ denotes k:=k+1. Ordered sets are traversed in the given order unless stated otherwise. For example, $i \in X$ corresponds to "for $i=0,\ldots,n-1$." Following [12] we use $i \prec j$ to denote v_i as an argument of φ_j . We write = for mathematical equality, \equiv in the sense of "is defined as" and := to represent assignment as in imperative programming languages. While all experiments are performed with C++ the conceptual results are applicable to any imperative programming language offering support for operator and function overloading.

Equation (1.2) induces a directed acyclic graph (DAG) G = (V, E) with vertices $V = \{0, \ldots, n+q-1\}$ and $E \subseteq V \times V$. Vertices are partitioned into inputs $X = \{0, \ldots, n-1\}$, outputs $Y \in \{0, \ldots, n+q-1\}$, |Y| = m. Local partial derivatives $d_{j,i} \equiv \frac{\partial v_j}{\partial v_i} ((v_l)_{l \prec j})$ of the elementals are associated with all edges.

In the following we propose special treatment of L-values associated with a physical structure of L-values associated with a physical structure.

In the following we propose special treatment of L-values associated with a physical address in the system memory occupied by the primal program. Such variables can appear on the left-hand side of assignments. The assignment operator is invoked on L-values only.

The construction of $G = G(\mathbf{x})$ relies on \mathbf{x} fixing the flow of control in the primal program. Overloading tools for AD use different representations of the DAG which are commonly referred to as tapes. We separate sequentially and randomly accessed parts of G by storing them in sequentially accessed memory (SAM) and randomly accessed memory (RAM), respectively. G is represented by three arrays $\mathbf{s} \in \mathbb{Z}^{2 \cdot |V| - |X| + |E|}$, $\mathbf{d} \in \mathbb{R}^{|E|}$, and $\bar{\mathbf{v}} \in \mathbb{R}^{|RAM|}$. The vector $\mathbf{s} \subset SAM$ describes the structure of G starting with the inputs. Entries are generated for each elemental function evaluation in form of its arguments followed by the number of arguments and the result. The derivatives associated with all edges are stored in $\mathbf{d} \subset SAM$. Adjoints are stored in $\bar{\mathbf{v}} = RAM$.

This internal representation of G is often referred to as a gradient tape. Note that the elementals φ_j , $j=0,\ldots,n+q-1$, can represent arbitrary differentiable multivariate vector functions. The sole requirement is the availability of the corresponding adjoint elementals. Without loss of generality the previously proposed internal representation makes the additional assumption that all elementals are scalar functions.

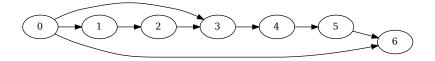


Fig. 1.1. *DAG*

As an alternative to the gradient tape a value tape would store the values v_j alongside corresponding operation codes for all φ_j , $j=0,\ldots,n+q-1$. Reevaluation of the primal at different points requires repeated recording of the gradient tape while value tapes can be reevaluated by interpretation. An in-depth discussion of the pros and cons of gradient vs. value tapes is beyond the scope of this paper. Numerous technical details and specifics of the given use cases influence the decision about the preferred approach.

State of the art implementations of AD make use of advanced template metaprogramming features of modern C++ [15, 20, 21]. As a result right-hand sides of assignments or even entire basic blocks can become elementals. The entire RAM can be occupied by *L*-values exclusively. The validity of the novel approach to handling of adjoint memory to be proposed in the following remains unaffected.

Adjoint code generated by source code transformation tools such as TAF [9] or Tapenade [14] does not benefit substantially from the following discussion. It typically uses an exact image (also: *shadow memory*) of the memory occupied by the primal program for the storage of the adjoints. The discussion of potential shortcomings of this approach is beyond the scope of this paper.

Example. Let $f: \mathbb{R} \to \mathbb{R}: x \mapsto y = f(x)$ be implemented in C++ as ¹

```
template<typename T>
void f(std::vector<T>& v) {
   T u;
for (size_t i=1;i<v.size();i++) {
   u=sin(v[i-1]);
   v[i]=u*u+v[0];
}</pre>
```

for $\mathbf{v} \in \mathbb{R}^l$, $v_0 = x = 1$ and $y = v_{l-1}$. We set l = 3. The corresponding DAG is stored as

$$\mathbf{s}^T = (0\ 0\ 1\ 1\ 1\ 1\ 2\ 2\ 0\ 2\ 3\ 3\ 1\ 4\ 4\ 1\ 5\ 5\ 0\ 2\ 6) \in \mathbb{Z}^{21}$$
$$\mathbf{d}^T = (0.54\ 1.68\ 1\ 1\ -0.14\ 1.98\ 1\ 1) \in \mathbb{R}^8\ .$$

The vector \mathbf{s} is best interpreted in reverse as it is the case in adjoint AD. Vertex 6 has the 2 predecessors 0 and 5. Vertex 5 has the single predecessor 4. ... Vertex 1 has the single predecessor 0. Vertex 0 is the only input. The vector \mathbf{d} contains the local partial derivatives associated with all edges in the order induced by \mathbf{s} . Edge (0,1) is

¹This example is supposed to illustrate certain aspects of different approaches to the propagation of adjoints. We do not claim for it to be a reasonable implementation of a practically relevant mathematical model as a computer program; see Section 5 for real-world applications.

labelled with

$$\frac{\partial v_1}{\partial v_0} = \frac{\partial \sin(v_0)}{\partial v_0} = \cos(v_0) = \cos(1) \approx 0.54$$

and so forth. Floating-point values are rounded to nearest [1]. A visualization of the DAG can be found in Figure 1.1.

In the following we present three methods for the propagation of adjoints using different approaches to the allocation of the vector of adjoints $\bar{\mathbf{v}} \in \mathbb{R}^{|\mathrm{RAM}|}$. We aim for $|\mathrm{RAM}| \to \mathrm{min}$. Naive adjoint AD records the entire DAG followed by its interpretation as recalled in Section 2. Exploitation of bandwidth is likely to result in a reduction of the adjoint memory requirement as described in Section 3. More substantial improvements can typically be expected from the provision of dedicated memory for adjoint L-values; see Section 4.

2. Flat adjoints by interpretation of the DAG. Basic adjoint AD builds G = (V, E) (recording) followed by allocation of $\bar{\mathbf{v}} \in \mathbb{R}^{|V|}$, initialization of \bar{y}_k , $k \in Y$ (seeding), interpretation of $(G, \bar{\mathbf{v}})$ (also: back-propagation) and extraction of adjoints of the inputs \bar{v}_i , $i \in X$ (harvesting).

For given \bar{y}_k , $k \in Y$ and setting $\bar{v}_j := 0$ for $j \in V \setminus Y$, back-propagation amounts to the evaluation of the adjoint program as

(2.1)
$$\bar{v}_{j_k} = \bar{y}_k \quad \text{for } k = 0, \dots, m-1 \text{ and } j_k \in Y;$$

$$\bar{v}_i += \bar{v}_j \cdot d_{j,i} \quad \text{for } i \prec j \text{ and } j = n+q-1, \dots, n;$$

$$\bar{x}_i = \bar{v}_i \quad \forall i \in X.$$

All conditions formulated for Equation (1.2) apply. See [12] for a proof of correctness of Equation (2.1). We use the C++-style notation $\bar{v}_i += \bar{v}_j \cdot d_{j,i}$ to abbreviate $\bar{v}_i := \bar{v}_i + \bar{v}_j \cdot d_{j,i}$. RAM of size $|V| \cdot \sigma$ is required, where σ denotes the number of bytes occupied by a scalar adjoint, e.g. $\sigma = 8$ for double precision floating-point variables according to the IEEE 754 standard [1].

Example. Setting $\bar{y} = 1$ yields $\bar{\mathbf{v}}^T = (0\ 0\ 0\ 0\ 0\ 1) \in \mathbb{R}^7$ as |V| = 7. Interpretation makes $\bar{\mathbf{v}}^T$ evolve as

$$\begin{array}{lll} (1\ 0\ 0\ 0\ 0\ 1\ 0) & [\bar{v}_0\ +=\ d_7\cdot\bar{v}_6;\ \bar{v}_5\ +=\ d_6\cdot\bar{v}_6;\ \bar{v}_6:=0] \\ (1\ 0\ 0\ 0\ 1.98\ 0\ 0) & [\bar{v}_4\ +=\ d_5\cdot\bar{v}_5;\ \bar{v}_5:=0] \\ (1\ 0\ 0\ -0.27\ 0\ 0\ 0) & [\bar{v}_3\ +=\ d_4\cdot\bar{v}_4;\ \bar{v}_4:=0] \\ (0.73\ 0\ -0.27\ 0\ 0\ 0\ 0) & [\bar{v}_0\ +=\ d_3\cdot\bar{v}_3;\ \bar{v}_2\ +=\ d_2\cdot\bar{v}_3;\ \bar{v}_3:=0] \\ (0.73\ -0.46\ 0\ 0\ 0\ 0\ 0) & [\bar{v}_1\ +=\ d_1\cdot\bar{v}_2;\ \bar{v}_2:=0] \\ (0.48\ 0\ 0\ 0\ 0\ 0) & [\bar{v}_0\ +=\ d_0\cdot\bar{v}_1;\ \bar{v}_1:=0] \\ \end{array}$$

resulting in $\bar{x} = \bar{v}_0 = 0.48$.

Setting $\bar{v}_j := 0$ after use in the second line of Equation (2.1) is actually obsolete for flat adjoints. It becomes essential for ensuring correctness of the alternative methods to be proposed in Sections 3 and 4.

3. Exploitation of bandwidth. The bandwidth β of G = (V, E) is defined for a given topological order of the vertices as the length j - i of the longest edge $(i, j) \in E$, that is,

$$\beta \equiv \max_{(i,j)\in E} (j-i) .$$

The topological order is induced by the sequence of elemental functions evaluated by the primal program; see Equation (1.2). Finding a topological order which minimizes the bandwidth is known to be NP-complete [8]. RAM of size $\max(\beta, n, m)$ is sufficient to evaluate the primal program and its adjoint, respectively.

Equation (1.2) can hence be evaluated as

$$(3.1) \quad \begin{aligned} v_i &= x_i & \text{for } i \in X \;; \\ v_{j\%\beta} &:= \varphi_j \left((v_{i\%\beta})_{i \prec j} \right) \\ y_{k++} &:= v_{j\%\beta} \quad \text{if } j \in Y \end{aligned} \quad \text{for } j \in V \setminus X \text{ and } k := 0 \text{ initially.}$$

Dedicated outputs are required as the exploitation of bandwidth may yield overwrites; see the explicit assignment in the third line of Equation (3.1). Correctness follows immediately from Equation (1.2) realizing that liveness of the v_j can be restricted to the evaluation of all v_k with $k \leq j + \beta$.

Setting $\bar{v}_j := 0$ for $j \in V$ the interpretation of G can be performed for given $\bar{\mathbf{y}}$ in RAM of size $\beta \cdot \sigma$ as

$$\bar{v}_{(j_k\%\beta)} := \bar{y}_k \qquad \text{for } k = 0, \dots, m-1 \text{ and } j_k \in Y;$$

$$w := \bar{v}_{(j\%\beta)}; \ \bar{v}_{(j\%\beta)} := 0$$

$$\bar{v}_{(i\%\beta)} := \bar{v}_{(i\%\beta)} + w \cdot d_{j,i} \text{ for } i \prec j$$

$$\bar{x}_i = \bar{v}_{(i\%\beta)} \qquad \text{for } i = 0, \dots, n-1.$$

Reuse of RAM locations for distinct variables requires resetting them to zero after use in the second equation. Correctness follows immediately from Equation (3.1).

Example. Setting $\bar{y}=1$ yields $\bar{\mathbf{v}}^T=(1\ 0\ 0\ 0\ 0\ 0)\in\mathbb{R}^6$ as $\beta=6-0=6$. Interpretation makes $\bar{\mathbf{v}}^T$ evolve as

resulting in $\bar{x} = \bar{v}_0 = 0.48$.

The reduction in the size of required RAM is not impressive as the longest edge spans nearly the entire computation. Similar observations can be made for many practically relevant numerical simulations which is why the exploitation of bandwidth alone is typically not enough. *Perpetuation* has been proposed in [18] to address this issue. A potentially more powerful method will be proposed in Section 4.

The effect of exploiting the bandwidth becomes much more significant for simple evolutions of length l defined as

$$\mathbf{v} = \underbrace{F(F(\dots F(\mathbf{v})\dots))}_{l \text{ times}}$$

for $F: \mathbb{R}^n \to \mathbb{R}^n$. The bandwidth remains bounded by $2 \cdot n$ independent of l. The adjoint interpretation of G can be performed in RAM of size $2 \cdot n \cdot \sigma$.

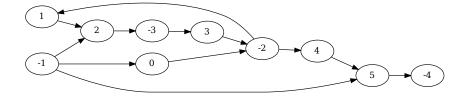


Fig. 4.1. DCG

4. Dedicated adjoint *L***-values.** Unique locations in system memory are assigned to all *L*-values. A directed cyclic graph (DCG) is induced due to potential overwrites. The total size of RAM can often be reduced significantly, possibly at the expense of an increase in the size of SAM.

Let $V = (L, R), R \equiv V \setminus L$ (remainder) with $X \cup Y \subseteq L$. Define

$$\&: V \to \{-p_L, \dots, |R|\}: i \mapsto \begin{cases} j \in \{-1, \dots, -p_L\} & i \in L \\ j \in \{0, \dots, |R|\} & i \in R \end{cases}$$

to be injective over L and bijective over R with p_L denoting the number of distinct L-values (with distinct physical addresses) in the primal program.

Let $E|_R \equiv \{(i,j) \in E : i,j \in R\}$. Define the remainder bandwidth of G as

$$\beta_R = \beta(E|_R) \equiv \max_{(i,j)\in E|_R} \&(j) - \&(i) .$$

Let the mapping of DAG vertices $i=0,\ldots,n+q-1$ to RAM with dedicated L-values be defined as

$$\#(i) \equiv \begin{cases} \&(i) & \text{if } i \in L \\ \&(i)\%\beta_R & \text{if } i \in R \end{cases}.$$

The primal program can be evaluated in RAM of size $p_L + \beta_R$ as

$$\begin{split} v_{\#(i)} &= x_i & \text{for } i \in X \;; \\ v_{\#(j)} &:= \varphi_j \left((v_{\#(i)})_{i \prec j} \right) & \text{for } j = n, \dots, n+q-1 \;; \\ k &:= 0 & \\ y_{k++} &= v_{\#(j_k)} & \text{for } j_k \in Y \text{ and } k := 0 \text{ initially }. \end{split}$$

The corresponding adjoint becomes equal to

$$\bar{v}_{\#(j_k)} = \bar{y}_{k++}$$
 for $j_k \in Y$ and $k := 0$ initially;
 $w := \bar{v}_{\#(j)}; \ \bar{v}_{\#(j)} := 0$ for $j = n + q - 1, \dots, n$;
 $\bar{x}_i = \bar{v}_{\#(i)}$ for $i < j$ for $i = 0, \dots, n - 1$.

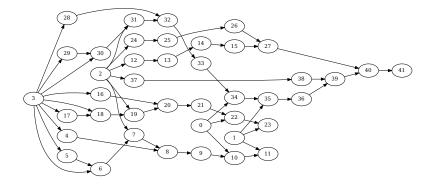


Fig. 5.1. Black-Scholes equation: DAG for three Monte Carlo paths.

Example. The DCG is stored as

$$\mathbf{s}^{T} = (-1 \ -1 \ 1 \ 0 \ 0 \ 1 \ -2 \ -2 \ -2 \ 2 \ 1 \ 1 \ -1 \ 2 \ 2 \ 2 \ 1 \ -3 \ -3 \ 1 \ 3 \ 3 \ 1 \ -2 \dots$$

$$\dots -2 \ -2 \ 2 \ 4 \ 4 \ -1 \ 2 \ 5 \ 5 \ 1 \ -4) \in \mathbb{Z}^{35}$$

$$\mathbf{d}^{T} = (0.54 \ 1 \ 1.68 \ 1 \ 1 \ 1 \ -0.14 \ 1 \ 1.98 \ 1 \ 1 \ 1) \in \mathbb{R}^{12}$$

A visualization can be found in Figure 4.1. Note that vertex -2 corresponds to the program variable u and is hence visited twice.

Setting $\bar{y} = 1$ yields $\bar{\mathbf{v}}^T = (0\ 0\ 0\ 1\ 0) \in \mathbb{R}^5$ as $p_L + \beta_R = 4 + 1 = 5$. The vertices of the DCG are visited as

$$-4$$
, 5, -1 , ,4, -2 , 3, -3 , 2, -1 , 1, -2 , 0, -1 ,

where all non-negative vertices are mapped onto \bar{v}_4 while the L-values corresponding to the program variables v[0], u, v[1], and v[2] are represented by vertices -1, -2, -3, and -4, which are mapped onto \bar{v}_0 , \bar{v}_1 , \bar{v}_2 , and \bar{v}_3 , respectively. Interpretation makes $\bar{\mathbf{v}}^T$ evolve as

resulting in $\bar{x} = \bar{v}_0 = 0.48$.

5. Case Studies. One of the case studies considers the solution of the Black-Scholes stochastic differential equation [3] by Monte Carlo simulation [10]. First, we

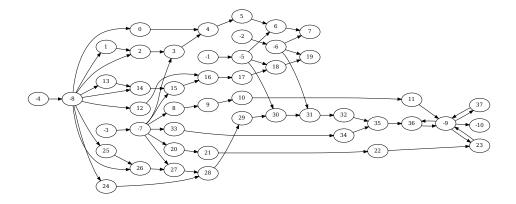


Fig. 5.2. Black-Scholes equation: DCG for three Monte Carlo paths.

take a closer look at a simplified scenario running only three paths resulting in the DAG in Figure 5.1. Total flattening yields a RAM requirement of $42 \cdot 8b = 336b$. The associated SAM occupies 992b. The longest edge is (2,37) yielding a total RAM requirement of $(35+1) \cdot 8b = 288b$ under exploitation of bandwidth and with a dedicated adjoint for the single output. The SAM size remains unchanged. Dedicated adjoint L-values lead to the DCG in Figure 5.2. The longest edge is (0,4). With ten dedicated adjoint L-value the total RAM requirement becomes equal to $14 \cdot 8b = 112b$, which amounts to one third of the RAM required by a total flattening approach due to complete mutual independence of the individual paths; see also [13]. The SAM requirement is increased slightly to 1172b.

Test results for a selection of practically relevant problems are listed in Tables 5.1 and 5.2. They illustrate the potential of dedicating memory to adjoint L-values. RAM requirement can be reduced significantly. We compare RAM sizes resulting from the approaches discussed in Sections 2-4 in Table 5.1. The following problems are considered:

- (a) Black-Scholes Equation; finite difference scheme on $3 \cdot 10^2 \times 9 \cdot 10^4$ grid
- (b) Burgers Equation [5]; finite difference scheme with upwind on $10^2 \times 10^4$ grid
- (c) LIBOR Market Model [4]; Monte Carlo simulation with 10⁴ paths
- (d) Black-Scholes Equation; Monte Carlo simulation with 10⁷ paths.

All numerical results were cross-validated and compared with finite difference approximations. Table 5.2 shows the corresponding SAM sizes. An increase between seven and up to thirty per cent can be observed. This drawback can be mitigated by asynchronous streaming of the data across the memory hierarchy. Acceptable slow-down due to the implementation of the additional logic or even speed-up due to the reduction in RAM size was observed as supported by the wall clock time measurements in Table 5.1. All problem sizes allowed for accommodation of both RAM and SAM within the system's main memory (approximately 15gb available on our Intel Xeon workstation running Linux and gcc version 9.4.0).

6. Conclusion. The allocation of dedicated memory for adjoint L-values combined with the exploitation of remainder bandwidth yields a substantial reduction of random access memory requirement in derivative programs obtained by adjoint AD

problem	Sec. 2	Sec. 3	Sec. 4
(a)	3.383.079.584 (17,4)	3.383.077.128 (18,1)	9.736 (17,2)
(b)	901.360.800 (4,3)	901.356.056 (4,7)	133.920.840 (6,0)
(c)	1.745.200.424 (8,1)	1.745.199.448 (8,9)	16.240.664 (9,0)
(d)	999.989.800 (4,6)	999.989.744 (5,2)	112 (5,0)

Table 5.1

RAM requirement in bytes (wall clock times in seconds).

	problem	Sec. 2 & Sec. 3	Sec. 4
Î	(a)	11.306.171.264	12.361.174.724
	(b)	2.863.480.400	3.906.400.400
	(c)	5.232.699.600	6.243.799.340
ı	(d)	3.279.959.064	3.479.959.184

Table 5.2

SAM requirement in bytes.

by overloading. Our reference implementation can be found on

https://github.com/un110076/dedicated_l_values.git

allowing for qualitative reproduction of all results. Its level of sophistication lacks behind state-of-the-art implementations of adjoint AD by overloading in C++ such as the solutions offered, for example, by NAG.² It pursues a different purpose by aiming to serve as an easy-to-follow illustration of the main algorithmic ideas presented in this paper.

REFERENCES

- [1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
- [2] C. Bischof, N. Guertler, A. Kowarz, and A. Walther. Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, Advances in Automatic Differentiation, pages 163–173. Springer, 2008.
- [3] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.
- [4] A. Brace, D. Gatarek, and M. Musiela. The market model of interest rate dynamics. Mathematical Finance, 7:127–147, 1997.
- [5] J. Burgers. Mathematical examples illustrating relations occurring in the theory of turbulent fluid motion. Verhandelingen der Koninklijke Nederlandse Akademie van Wetenschappen, Afdeeling Natuurkunde, 2(17):1–53, 1939.
- [6] N. Dunford and J.T. Schwartz. Linear Operators, Part 1: General Theory. Wiley Classics Library. Wiley, 1988.
- [7] M. Förster. Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP. Springer, Wiesbaden, 2014.
- [8] M. Garey, R. Graham, D. Johnson, and D. Knuth. Complexity results for bandwidth minimization. SIAM J. Appl. Math., 34:477-495, 1978.
- [9] R. Giering and T. Kaminski. Recipes for adjoint code construction. ACM Transactions on Mathematical Software, 24(4):437–474, 1998.
- [10] P. Glasserman. Monte Carlo Methods in Financial Engineering. Springer, New York, NY, USA, 2004.
- [11] F. Gremse, A. Hoefter, L. Razik, F. Kiessling, and U. Naumann. GPU-accelerated adjoint algorithmic differentiation. Computer Physics Communications, 200:300–311, 2016.
- [12] A. Griewank and A. Walther. Evaluating Derivatives: Principles and Techniques of Algo-

²Numerical Algorithms Group Ltd., Oxford, UK; www.nag.com

- rithmic Differentiation. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008.
- [13] L. Hascoët, S. Fidanova, and C. Held. Adjoining independent computations. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, Automatic Differentiation of Algorithms: From Simulation to Optimization, Computer and Information Science, chapter 35, pages 299–304. Springer, New York, NY, 2002.
- [14] L. Hascoët and V. Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. ACM Transactions on Mathematical Software, 39(3):20:1–20:43, 2013.
- [15] R. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. ACM Transactions on Mathematical Software, 40(4):26:1–26:24, jun 2014.
- [16] W. Moses, V. Churavy, L. Paehler, J. Hückelheim, K. Narayanan, M. Schanen, and J. Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.
- [17] U. Naumann. The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation. Number SE24 in Software, Environments, and Tools. SIAM, 2012.
- [18] U. Naumann and K. Leppkes. Low-memory algorithmic adjoint propagation. In 2018 Proceedings of the Eighth SIAM Workshop on Combinatorial Scientific Computing, pages 1–10.
- [19] U. Naumann, J. Utke, J. Riehme, P. Hovland, and C. Hill. A framework for proving correctness of adjoint message passing programs. In *Proceedings of EUROPVM/MPI 2008*, pages 316–321, 2008.
- [20] E. Phipps and R. Pawlowski. Efficient expression templates for operator overloading-based automatic differentiation. In S. Forth, P. Hovland, E. Phipps, J. Utke, and A. Walther, editors, Recent Advances in Algorithmic Differentiation, volume 87 of Lecture Notes in Computational Science and Engineering, pages 309–319. Springer, Berlin, 2012.
- [21] M. Sagebaum, T. Albring, and N. Gauger. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. Optimization Methods & Software, 33(4– 6):1207–1231, 2018.
- [22] J. Utke, L. Hascoët, C. Hill, P. Hovland, and U. Naumann. Toward adjoinable MPI. In Proceedings of IPDPS 2009, 2009.