stats

MDPI

Article

An Analysis of Vectorised Automatic Differentiation for Statistical Applications

Chun Fung Kwok [1], Dan Zhu [2,*] and Liana Jacobi [3]

[1] St. Vincent's Institute of Medical Research, Melbourne 3065, Australia; jkwok@svi.edu.au
[2] Department of Econometrics and Business Statistics, Monash University, Melbourne 3800, Australia
[3] Department of Economics, University of Melbourne, Melbourne 3010, Australia; ljacobi@unimelb.edu.au
* Correspondence: dan.zhu@monash.edu

Abstract: Automatic differentiation (AD) is a general method for computing exact derivatives in complex sensitivity analyses and optimisation tasks, particularly when closed-form solutions are unavailable and traditional analytical or numerical methods fall short. This paper introduces a vectorised formulation of AD grounded in matrix calculus. It aligns naturally with the matrix-oriented style prevalent in statistics, supports convenient implementations, and takes advantage of sparse matrix representation and other high-level optimisation techniques that are not available in the scalar counterpart. Our formulation is well-suited to high-dimensional statistical applications, where finite differences (FD) scale poorly due to the need to repeat computations for each input dimension, resulting in significant overhead, and is advantageous in simulation-intensive settings—such as Markov Chain Monte Carlo (MCMC)-based inference—where FD requires repeated sampling and multiple function evaluations, while AD can compute exact derivatives in a single pass, substantially reducing computational cost. Numerical studies are presented to demonstrate the efficacy and speed of the proposed AD method compared with FD schemes.

Keywords: automatic differentiation; derivative computation; matrix calculus; MCMC; MLE; optimisation; simulation-based inference

JEL Classification: C11; C53; E37

check for updates

Academic Editor: Wei Zhu

Received: 20 March 2025
Revised: 28 April 2025
Accepted: 12 May 2025
Published: 19 May 2025

Citation: Kwok, C.F.; Zhu, D.; Jacobi, L. An Analysis of Vectorised Automatic Differentiation for Statistical Applications. *Stats* **2025**, *8*, 40. https://doi.org/10.3390/stats8020040

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Automatic differentiation (AD) has become a foundational tool in modern statistical computing, enabling efficient and exact gradient computation in a wide range of applications—from parameter estimation [1,2] and sensitivity analysis [3–5] to simulation-based methods such as variational inference and Markov-Chain Monte Carlo (MCMC) inference [6–11]. Its widespread adoption is evident in major software ecosystems such as PyTorch [12], TensorFlow [13], Stan [14], and Julia [7], where AD powers both machine learning workflows and traditional statistical methods.

AD works by transforming a program that computes the value of a function into one that also computes its derivatives by systematically applying the chain rule to elementary operations. This allows AD to compute derivatives with machine-level precision and minimal overhead, avoiding truncation and round-off errors and eliminating the need for repeated function evaluations, a known bottleneck in numerical differentiation, especially in high-dimensional problems [8,15,16]. While symbolic differentiation can provide exact derivatives, it requires closed-form expressions and cannot handle procedural logic (e.g., if-else statements and for-loops) or stochastic elements such as random number generation

or Monte Carlo simulations. AD bridges this gap, offering a robust and general-purpose solution for derivative computation.

Early implementations of AD relied on operator overloading [17] and source code translation [18] techniques that, while powerful, had notable limitations. Operator overloading incurs significant runtime overhead and is inherently local, recording operations as they occur without access to global program structure or opportunities for optimisation. In contrast, compiler-based systems transform entire programs automatically but often make it difficult to selectively extract intermediate values for debugging or inspection. Modern AD frameworks improve on these predecessors by adopting either eager execution (as in PyTorch 1 and 2) or static computational graphs (as in TensorFlow 1). These approaches offer greater flexibility, traceability, and support for modular development and deep introspection. However, the eager mode requires users to structure their code in specific ways; for example, making explicit calls such as `backward()` and `zero_grad()` in PyTorch can appear rigid and error-prone. Moreover, because it operates step-by-step, the eager approach often fails to exploit the broader structure of computations, such as block matrix operations, thereby missing optimisation opportunities. Conversely, while static graph systems are more declarative and amenable to global analysis, they can struggle with dynamic control flow and runtime-dependent logic. In both paradigms, the need to conform to AD-specific programming idioms often shifts user attention away from the statistical problem itself and toward the mechanics of the AD system.

In this work, we present a vectorised formulation of AD grounded in the matrix calculus of [19], designed to align more naturally with the matrix-oriented style prevalent in the field of statistics and the statistical programming language R. Our approach mirrors the derivation style of analytical work, enabling clearer and more intuitive implementations. It also exposes opportunities for high-level optimisation, including the use of sparse matrix representations and block-wise computations, features often inaccessible in bottom-up, scalar-based AD systems. This formulation supports transparent complexity analysis and efficient implementations, particularly in settings involving Kronecker products. It also enables fully automatic workflows, akin to source code translation techniques, while preserving the ability to inspect intermediate variables as in eager execution and graph-based systems.

We introduce a complete set of matrix calculus rules for building an AD system tailored to statistical applications, including operations for random variable simulations and structural transformations, many of which are undocumented in the existing literature. We also introduce the sparse representation of transformation matrices and discuss a range of optimisation techniques applied to the AD system to achieve significant performance gains in practice, which we demonstrate through comparisons with finite differences (FD). As an illustration, we apply the proposed methods using a real data example: a factor model estimated using simulated maximum likelihood, a setting commonly encountered when modelling dependence structures in complex data. The numerical results confirm the computational advantage of the proposed vectorised AD, particularly for simulation-intensive functions where FD incurs unnecessary repeated calculations.

The remainder of the paper is organised as follows: Section 2 introduces the core mechanism of the AD system and presents the full set of matrix calculus rules. Section 3 discusses optimisation strategies and evaluates computational performance, while Section 4 details the application. All code listings are provided in the Appendix B.

2. Materials and Methods

2.1. AD via Vectorisation

2.1.1. From Vector Calculus to Matrix Calculus via Vectorisation

Our AD formulation builds on a set of vector calculus rules rather than elementary scalar calculus [19]. Before presenting the full framework, we introduce three key definitions: Definition 1 defines the derivative of a vector-valued function; Definition 2 introduces the vectorisation operator; and Definition 3 combines the first two to define the derivative of a matrix-valued function.

Definition 1. *Suppose* $\mathbf{x} \in \mathbb{R}^n$ *and* $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$*, then Jacobian matrix* J *of* $\mathbf{f}$ *is a* $m \times n$ *matrix* $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ *with* (i, j) *entry given by:*

$$\frac{\partial f_i}{\partial x_j}, \quad i = 1, 2, \ldots, m; \quad j = 1, 2, \ldots, n$$

where f_i *and* x_j *are components of* $\mathbf{f}$ *and* $\mathbf{x}$*.*

Definition 2. *Let* A *be an* $m \times n$ *matrix and* a_j *its* j*-th column. Then vec* A *is the* $mn \times 1$ *column vector (i.e., vec* A *stacks the columns of* A*):*

$$[a_{11}\ a_{21}\ \ldots\ a_{m1}\ a_{12}\ a_{22}\ \ldots\ a_{m2}\ \ldots\ a_{1n}\ a_{2n}\ \ldots\ a_{mn}]^T.$$

Note that $\operatorname{vec} ABC = (C^T \otimes A)\operatorname{vec} B$, where A, B, and C are three matrices with appropriate dimensions such that the matrix product ABC is well-defined; C^T denotes the transpose of C, and $C^T \otimes A$ denotes the Kronecker product of C^T and A.

Definition 3. *Let* $F : \mathbb{R}^{n \times q} \to \mathbb{R}^{m \times p}$ *be a real matrix function, the Jacobian matrix of* F *at* X *is defined to be the* $mp \times nq$ *matrix:*

$$DF(X) := \frac{\partial vecF(X)}{\partial (vecX)^T}$$

For notational convenience, we write $\partial \mathrm{vec} X$ as dX. In the definition above, the numerator is always treated as a column vector and the denominator as a row vector. This allows us to write $DF(X)$ as $\frac{dF(X)}{dX}$ without ambiguity, rather than the more cumbersome $\frac{dF(X)}{(dX)^T}$. Indeed, since $\frac{\partial \mathrm{vec} X}{\partial (\mathrm{vec} X)^T} = I$, the identity matrix of dimension mp, accepting dX as $(dX)^T$ in the denominator amounts to a notational simplification:

$$\frac{dF(X)}{dX} = \frac{\partial \mathrm{vec} F(X)}{\partial \mathrm{vec} X} \cdot I = \frac{\partial \mathrm{vec} F(X)}{\partial \mathrm{vec} X} \cdot \frac{\partial \mathrm{vec} X}{\partial (\mathrm{vec} X)^T} = \frac{\partial \mathrm{vec} F(X)}{\partial (\mathrm{vec} X)^T},$$

aligning with the definition above.

A key advantage of Definition 3 is that it allows higher-order matrix derivatives to remain within the familiar matrix framework, rather than escalating into high-order tensors, e.g., the Hessian of F at X is also a matrix. This simplifies notation and facilitates the use of matrix algebra to exploit structure in Jacobian and Hessian matrices, making the formulation more efficient. For further discussion and critique of alternative matrix derivative conventions—including the numerator and denominator layouts—see [19].

Once derivatives are defined for the basic operations, they can be propagated through a computation using the matrix-based chain rule. Suppose that at a certain stage, we have already computed matrices A and B, along with their derivatives $DA(X)$ and $DB(X)$, with

respect to some input X. The next step of the computation involves evaluating a new matrix $C = F(A, B, X)$, where F is differentiable in all parameters. Using the chain rule in vectorised form, the derivative of C with respect to X is given by:

$$DC(X) = \frac{\partial \text{vec} F}{\partial \text{vec} X}(A, B, X) + \frac{\partial \text{vec} F}{\partial \text{vec} A}(A, B, X)DA(X) + \frac{\partial \text{vec} F}{\partial \text{vec} B}(A, B, X)DB(X). \quad (1)$$

In our formulation, any computation that can be decomposed into a sequence of basic matrix operations—each admitting a tractable and well-defined derivative—can be differentiated efficiently using the chain rule. These operations include (i) basic matrix arithmetic such as addition, subtraction, product, inverse, and Kronecker product; (ii) element-wise arithmetic such as Hadamard product/division and element-wise univariate differentiable transformations; (iii) scalar matrix arithmetic such as scalar matrix addition, subsection, multiplication, and division; (iv) structural transformations such as extracting elements and rearranging or combining matrices; and (v) operations on matrices such as Cholesky decomposition, column/row sum, cross-products, transposition of cross-products, determinants, and traces. They will be presented in Section 2.1.3.

2.1.2. Dual Construction

We present an implementation of an AD system that can differentiate any multivariate matrix polynomial to illustrate the underlying logic of our AD formulation. Let A, B, and C be $n \times n$ matrices and I_n the $n \times n$ diagonal (identity) matrix, and consider the following two matrix calculus rules:

$$C = A + B \quad \Rightarrow \quad dC = dA + dB, \quad (2)$$

$$C = AB \quad \Rightarrow \quad dC = (B^T \otimes I_n)dA + (I_n \otimes A)dB. \quad (3)$$

To implement these rules, first, we attach to each matrix a dual component that stores the derivative with respect to some other parameters (i.e., let $A_{dual} = \left(A, \frac{dA}{d*}\right), B_{dual} = \left(B, \frac{dB}{d*}\right)$) and refer to them as the dual matrices. For example, if the parameters are the entries of A and B, then $\frac{dA}{d*} = \left[\frac{dA}{d\,[A \mid B]}\right] = [I_{n^2}, 0_{n^2}]$; similarly $\frac{dB}{d*} = \left[\frac{dB}{d\,[A \mid B]}\right] = [0_{n^2}, I_{n^2}]$. We can then define the arithmetic for dual matrices using (2) and (3):

$$A_{dual} + B_{dual} = \left(A, \frac{dA}{d*}\right) + \left(B, \frac{dB}{d*}\right) = \left(A + B, \frac{dA}{d*} + \frac{dB}{d*}\right) \quad (4)$$

$$A_{dual} \cdot B_{dual} = \left(A, \frac{dA}{d*}\right) \cdot \left(B, \frac{dB}{d*}\right) = \left(AB, (B^T \otimes I_n)\frac{dA}{d*} + (I_n \otimes A)\frac{dB}{d*}\right) \quad (5)$$

and program them as shown in Listing 1.

These 16 lines of code define an AD system that can handle the class of multivariate matrix polynomials formed by addition and multiplication. For example, the derivative of the function $f(A, B) = A(AB + B^2) + B$ is simply the one-line

```
df <- function(A, B) A %times% ((A %times% B) %plus% (B %times% B)) %plus% B
```

rather than a tedious program associated with the analytical derivative

$$df(A, B) = \left[(AB + B^2)^T \otimes I_n + (I_n \otimes A)(B^T \otimes I_n)\right]dA + \left[(I_n \otimes A)^2 + (I_n \otimes A)(B^T \otimes I_n + I_n \otimes B) + I_{n^2}\right]dB. \quad (6)$$

Readers encountering AD for the first time may be surprised that the program df above appears to compute f itself, rather than its derivative—which is precisely what makes AD so appealing. Given the addition and multiplication operators defined for dual matrices, any function constructed using these operators will automatically have its derivative computed. Specifically, derivatives are evaluated on the fly each time `%times%` or `%plus%` is called. The final output of `df` is a dual matrix, where the first component is the result $f(A, B)$, and the second component is the derivative of f at (A, B). This approach abstracts away the derivative calculation, allowing users to obtain derivatives automatically once the function f is implemented. To complete the system, subtraction and inverse operations are also required (also 16 lines). To main the flow, we list them under the Appendix B, along with a complete working example.

Listing 1. Implementation of the sum and product matrix calculus rules in R.

```
`%plus%` <- function(A_dual, B_dual) {
  A <- A_dual$X;  dA <- A_dual$dX;
  B <- B_dual$X;  dB <- B_dual$dX;
  list(X = A + B, dX = dA + dB)
}

`%times%` <- function(A_dual, B_dual) {
  A <- A_dual$X;  dA <- A_dual$dX;
  B <- B_dual$X;  dB <- B_dual$dX;
  list(
    X = A %*% B,
    dX = (t(B) %x% I(nrow(A))) %*% dA + (I(ncol(B)) %x% A) %*% dB
  )
}

I <- diag   # function to create diagonal matrices
```

2.1.3. A Layered Approach to Construction

In the previous section, we showed that formulating AD with dual matrices closely mirrors the underlying analytic derivation. Once the calculus rules for dual matrices are established, building an AD system becomes relatively straightforward. In this section, we present the full set of matrix calculus rules. The rules are grouped by type and presented in the order they would typically be implemented in practice. Illustrative statistical applications are provided in the Appendix.

In the following discussion, the derivative of any matrix is assumed to be w.r.t. some input z with d parameters. Hence, if A is a $(m \times n)$ matrix, then $\frac{dA}{dz}$ is a $(mn \times d)$ Jacobian matrix, and this shall be written as dA for the sake of convenience.

2.1.4. Notation

The symbols $I_\cdot, K_\cdot, E_\cdot, D_\cdot, 1_\cdot$ are reserved for the follow special matrices:

- I_n is the $n \times n$ identity matrix.
- I_{nq} is the $n \times q$ matrix where the entries on the diagonal are all ones, and the entries off the diagonal are all zeros.
- K_{nq} is the $nq \times nq$ commutation matrix. We also define $K_n := K_{nn}$.
- E_n is the $\frac{n(n+1)}{2} \times n^2$ elimination matrix.
- 1_{nq} is the $n \times q$ matrix of ones.

(Definitions of the commutation and elimination matrices can be found in Sections 2.2.4 and 2.2.5, respectively.)

Let A be a $m \times n$ matrix. We denote

- the (i,j)-entry of A is denoted by A_{ij} or $A_{i,j}$,
- the i-th row of A is denoted by A_{i*} or $A_{i,*}$,
- the j-th column of A is denoted by A_{*j} or $A_{*,j}$.

We define $v_A(i,j)$ and $v_A^{-1}(k)$ such that they satisfy the relations

$$\begin{aligned} (i,j)\text{-entry of } A \quad &\Leftrightarrow \quad v_A(i,j)\text{-th entry of vec}(A) \\ v_A^{-1}(k)\text{-entry of } A \quad &\Leftrightarrow \quad k\text{-th entry of vec}(A) \end{aligned}$$

and they are given by the formulas

$$v_A(i,j) = i + (j-1)m \quad \text{and} \quad v_A^{-1}(k) = \left([(k-1) \bmod m] + 1, \left\lceil \frac{k}{m} \right\rceil \right). \tag{7}$$

This indices-conversion function is needed because the derivative of the (i,j)-entry of A is stored in the $v_A(i,j)$-th row of dA, and vice versa.

2.1.5. Matrix Arithmetic

We present the matrix calculus rules associated with basic matrix arithmetic:

- Addition: Let A and B be $m \times n$ matrices, then $d(A+B) = dA + dB$.
- Subtraction: Let A and B be $m \times n$ matrices, then $d(A-B) = dA - dB$.
- Product: Let A and B be $m \times n$ and $n \times k$ matrices, then

$$d(AB) = (B^T \otimes I_m)dA + (I_k \otimes A)dB.$$

- Inverse: Let A be a $n \times n$ matrix, then $dA^{-1} = -(A^{-1} \otimes A^{-1})dA$.
- Kronecker-product: Let A and B be $m \times n$ and $p \times q$ matrices, then

$$d(A \otimes B) = (I_n \otimes K_{qm} \otimes I_p)\big[(I_{mn} \otimes \text{vec}(B))dA + \big(\text{vec}(A) \otimes I_{pq}\big)dB\big].$$

- Transpose: Let A be a $m \times n$ matrix, $dA^T = K_{mn}dA$.

We now examine the computational advantages of AD relative to FD through a complexity analysis of basic matrix operations. Our discussion is restricted to central (finite) differencing instead of forward differencing, since the former has superior accuracy ([20], pp. 378–379) and is generally preferred over the latter. Applying central differencing to a function f incurs a cost of evaluating f multiplied by twice the dimension of the inputs. (For ease of discussion, we assume that conditional branches, if they exist, have the same order of computational complexity.) Applying AD incurs a cost of evaluating f and the derivative df as given by the calculus rules.

For matrix arithmetic, suppose A and B are $n \times n$ matrices, so the dimension of the inputs is $d = 2n^2$ for matrix additions, subtractions, products, and Kronecker products, and $d = n^2$ for matrix inversions. The number of operations associated with applying the central differencing to $f(A,B) : \mathbb{R}^d \to \mathbb{R}^k$ is:

$$\begin{aligned} &2d \text{ perturbations (additions / subtractions) of the input} \\ &+ 2d \text{ evaluations of } f \text{ with the perturbed input} \\ &+ d \text{ times } 2k \text{ operations for finite-differencing the output (subtractions and divisions)} \\ &= 2d \cdot (1 + \text{cost}(f) + k). \end{aligned}$$

Assuming standard matrix multiplication, the computational costs of addition, subtraction, multiplication, and the Kronecker product are n^2, n^2, $2n^3 - n^2$, and n^4, respectively. Since we are only interested in the leading terms, these simplify to n^2, n^2, $2n^3$, and n^4. The number of operations for matrix inversion using the LU decomposition followed by the inversion of triangular matrices is around $2n^3$ [21]. It then follows that applying finite-differencing would require (in the leading term) $8n^4$ operations for addition and subtraction, $8n^5$ operations for product, $4n^5$ operations for matrix inversion, and $8n^6$ operations for the Kronecker product.

For AD, the number of operations works out to be $n^2 + 2n^4$ for addition and subtraction, $(2n^3 - n^2) + (8n^5 - 2n^4)$ for multiplication, $2n^3 + (8n^5 - 2n^4)$ for matrix inversion, and $n^4 + 6n^6$ for the Kronecker product. The results are summarised in Table 1. Note that when a Kronecker product is post-multiplied by a matrix, there is a shortcut to avoid the explicit computation of the Kronecker product. The details will be given in Section 2.2.7. From Table 1, we observe that finite differencing and AD have the same complexity for the operations listed in the table, but AD generally has equal or better leading coefficients, except for the matrix inversion.

Table 1. Comparison of the number of operations (in terms of the leading order) needed in finite differencing and AD.

Operations	Central Differencing	AD
Addition	$8n^4$	$2n^4$
Subtraction	$8n^4$	$2n^4$
Multiplication	$8n^5$	$8n^5$
Inversion	$4n^5$	$8n^5$
Kronecker product	$8n^6$	$6n^6$

2.1.6. Element-Wise Arithmetic

We now present matrix calculus rules for element-wise operations. These rules follow directly from applying scalar calculus to each entry independently. The cases of addition and subtraction are identical to those covered in the previous section on standard matrix arithmetic. Let A, B, and C be $m \times n$ matrices, and $\mathrm{diag}(\mathbf{v})$ be the square matrix in which the vector $\mathbf{v}$ is placed on the diagonal.

- Hadamard product:

$$\begin{aligned} & C_{ij} = A_{ij}B_{ij}, \quad i = 1, 2, \ldots, m, \ j = 1, 2, \ldots, n \\ \Rightarrow \quad & (dC)_{k,*} = B_{v_B^{-1}(k)}(dA)_{k,*} + A_{v_A^{-1}(k)}(dB)_{k,*}, \quad k = 1, \ldots, mn \end{aligned}$$

  Alternatively,

$$C = A \odot B \quad \Rightarrow dC = \mathrm{diag}(\mathrm{vec}(B))dA + \mathrm{diag}(\mathrm{vec}(A))dB.$$

- Hadamard division:

$$\begin{aligned} & C_{ij} = A_{ij}/B_{ij}, \quad i = 1, 2, \ldots, m, \ j = 1, 2, \ldots, n \\ \Rightarrow \quad & (dC)_{k,*} = B^{-1}_{v_B^{-1}(k)}(dA)_{k,*} - A_{v_B^{-1}(k)}B^{-2}_{v_B^{-1}(k)}(dB)_{k,*}, \quad i = 1, \ldots, mn \end{aligned}$$

  Alternatively,

$$C = A \oslash B \quad \Rightarrow dC = \mathrm{diag}(\mathrm{vec}(B^{\circ -1}))dA - \mathrm{diag}(\mathrm{vec}(A \odot B^{\circ -2}))dB.$$

- Univariate differentiable function f:

$$C_{ij} = f(A_{ij}), \quad i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, n$$
$$\Rightarrow \quad (dC)_{k,*} = f'(A_{v_A^{-1}(k)})(dA)_{k,*}, \quad i = 1, \ldots, mn$$

Alternatively,

$$C = \overset{\circ}{f}(A) \quad \Rightarrow dC = \operatorname{diag}\left(\operatorname{vec}\left(\overset{\circ}{f'}(A)\right)\right) dA,$$

where $\overset{\circ}{f}(A)$ denotes applying f element-wise to A. Note that f may also be a function that is differentiable almost everywhere, e.g., $f(x) = |x|$ is differentiable everywhere except at $x = 0$. When the derivative is evaluated at the non-differentiable locations, it is common to use a subgradient [8]—in this case, any value in the interval $[-1, 1]$—or simply to assume a value such as 0, effectively treating these points as having no impact on the result.

2.1.7. Scalar-Matrix Arithmetic

Let A be a $m \times n$ matrix and c be a scalar. Then the differentials $d(c \text{ Op } A)$ and $d(A \text{ Op } c)$, where Op $\in \{+, -, *, /\}$, can be computed by lifting the scalar c to a matrix of the same dimension as A, via multiplication with a matrix of ones, 1_{mn}. This allows the scalar-matrix operation to be treated as an element-wise operation. Importantly, this lifting is a conceptual construct, and the implementation need not construct a new matrix $k \cdot 1_{mn}$. Instead, the operation can be performed element-wise directly. For instance, the product rule $d(cA)$ becomes

$$\begin{aligned} B = cA \quad &\Rightarrow B_{ij} = cA_{ij}, && i = 1, 2, \ldots, m,\ j = 1, 2, \ldots, n; \\ &\Rightarrow (dB)_{k,*} = A_{v_A^{-1}(k)} dc + c(dA)_{k,*}, && k = 1, 2, \ldots, mn. \end{aligned}$$

2.1.8. Structural Transformation

We now present calculus rules for structural transformations—operations that extract, rearrange, or combine matrix entries without performing any arithmetic computations. Because these transformations are primarily operational rather than analytical, they seldom appear in formal derivations and are often left undocumented in standard references.

- Transpose: Let A be a $m \times n$ matrix, $dA^T = K_{mn} dA$.
- Row binding: Let A, B be $m \times n, k \times n$ matrices, $\text{rowBind}(A, B) := \begin{bmatrix} A \\ B \end{bmatrix}$, then

  $$C = \text{rowBind}(A, B) \quad \Rightarrow (dC)_{k,*} = 1_{r \le m} \cdot (dA)_{v_A(r,c),*} + 1_{r > m} \cdot (dB)_{v_B(r-m,c),*}$$

  where $(r, c) = v_C^{-1}(k), k = 1, 2, \ldots, (m + k)n$.
- Column binding: Let A, B be $m \times n, m \times k$ matrices, $\text{colBind}(A, B) := [A \ \ B]$, then

  $$C := \text{colBind}(A, B) \quad \Rightarrow dC = \text{rowBind}(dA, dB).$$

- Subsetting: Let A be a $m \times n$ matrix,
  1. Index extraction: A_{ij} for fixed $i, j \quad \Rightarrow dA_{ij} = (dA)_{v_A(i,j),*}$
  2. Row extraction: A_{i*} for fixed $i \quad \Rightarrow dA_{i*} = (dA)_{S,*}$, where $S = \{i, i + m, \ldots, i + (n - 1)m\}$
  3. Column extraction: A_{*j} for fixed $j \quad \Rightarrow dA_{*j} = (dA)_{S,}$, where $S = \{(j - 1)m + 1, (j - 1)m + 2, \ldots, (j - 1)m + n\}$
  4. Diagonal extraction: $[A_{ii}]_{i=1,2,\ldots,n}$ (column vector) $\quad \Rightarrow d[A_{ii}]_{i=1,2,\ldots,n} = (dA)_{S,}$, where $S = \{1, 1 + (m + 1), 1 + 2(m + 1) + \ldots, 1 + (\min(n, m) - 1)(m + 1)\}$.
- Vectorisation: Let A be a $n \times n$ matrix, $d \operatorname{vec}(A) = dA$
- Half-vectorisation: Let A be a $n \times n$ matrix, $d \operatorname{vech}(A) = (dA)_{S,}, S = \{v_A(i, j), i \ge j\}$. Note that S follows the order as the column-major order of A, i.e.,

$$S = \{v_A(1,1), v_A(2,1), v_A(3,1), \ldots, v_A(2,2), v_A(2,3), \ldots, v_A(n,n),\}.$$

- Diagonal expansion: Let $\mathbf{v}$ be a $n \times 1$ vector, then $diag(\mathbf{v})$ is defined to be the $n \times n$ matrix with $\mathbf{v}$ on the diagonal. If $B = \text{diag}(\mathbf{v})$, then for $k = 1, 2, \ldots, n^2$,

$$(dB)_{k,*} = (dA)_{k \bmod (n+1),*} \cdot 1_{k \in S} + \mathbf{0} \cdot 1_{k \notin S},$$

where $S = \{1, n+2, 2n+3+\ldots, n^2\}$.

2.1.9. Operations on Matrices

- Cholesky Decomposition: Let A be a $n \times n$ matrix, $A = LL^T$ be the Cholesky decomposition and $L = Chol(A)$, then

$$dL = D_n^*[E_n(I_{n^2} + K_n)(L \otimes I_n)D_n^*]^{-1}E_n\, dA$$

where $D_n^* = E_n^T$ is the duplication matrix for triangular matrices.

Let A be a $m \times n$ matrix.

- Column-sum:

$$\begin{aligned} &\text{colSum}(A) := \sum_i A_{i,*} \\ \Rightarrow \quad &(d\ \text{colSum}(A))_{k,*} = \sum_{i=(k-1)m+1,(k-1)m+2,\ldots,km} (dA)_{i,*}, \quad k = 1, \ldots, n \end{aligned}$$

- Row-sum:

$$\begin{aligned} &\text{rowSum}(A) := \sum_j A_{*,j} \\ \Rightarrow \quad &(d\ \text{rowSum}(A))_{k,*} = \sum_{i=k,k+m,\ldots,k+(n-1)m} (dA)_{i,*}, \quad k = 1, \ldots, m \end{aligned}$$

- Sum: $\text{sum}(A) := \sum_{i,j} A_{ij} \quad \Rightarrow d\ \text{sum}(A) = \text{colSum}(dA)$
- Cross-product:

$$\text{crossprod}(A) := A^T A \Rightarrow \quad d\ \text{crossprod}(A) = (I_{q^2} + K_{qq})(I_q \otimes A^T).$$

- Transpose of cross-product:

$$\text{tcrossprod}(A) := AA^T \Rightarrow \quad d\ \text{tcrossprod}(A) = (I_{n^2} + K_{nn})(A_n \otimes I_n).$$

Alternatively, both 'crossprod' and 'tcrossprod' can be implemented directly as is, since they are composed of the multiplication and transpose operations defined previously.

Let A be a $n \times n$ matrix.

- Determinant: $d\ \det(A) = \det(A) \cdot \text{vec}(A^{-T})^T \cdot dA$
- Trace: $d\ \text{tr}(A) = \text{vec}(I_n)^T dA$. Alternatively, it can be implemented by composing sum and diagonal extraction defined previously.

2.1.10. Random Variables

Given a probability space $(\Omega, \mathcal{F}, P)$, a random variable X is a $\mathcal{F}$-measurable function mapping Ω to $\mathbb{R}$. The formalism suggests that in the process of simulating a random variate, the randomness can always be isolated, and it is possible to differentiate (in the pathwise sense) the random variables $X \sim F_X(x; \boldsymbol{\alpha})$ w.r.t. the parameters $\boldsymbol{\alpha}$ when the derivative exists. In the simplest case of normal random variables, the parameters and the randomness can be separated as follows:

$$Z \sim N(\mu, \sigma) \quad \Rightarrow \quad Z = \mu + \sigma Z_0, \;\; Z_0 \sim N(0,1) \quad \Rightarrow \quad dZ = d\mu + d\sigma \cdot Z_0$$

As Z depends smoothly on the parameters μ and σ, the derivatives w.r.t. these parameters are well defined. This is commonly referred to as the reparametrisation trick [22].

When explicit separation cannot be done, we utilise the inverse transform method. Suppose $Z \sim F_Z(z;\theta)$ where $F_Z(z;\theta)$ is the cumulative density function of Z, assumed to be invertiable, and θ is the parameter. Then Z can be simulated using the inverse transform method $Z = F_Z^{-1}(U;\theta)$, $U \sim U[0,1]$. It then follows that if $F_Z^{-1}(\cdot;\theta)$ is differentiable in θ, then the derivative of a random sample is well-defined. This applies to, for instance, the Exponential, Weibull, Rayleigh, log-Cauchy, and log-Logistic distributions.

In the most general case where Z is high-dimensional and F_Z^{-1} may not be known, we rely on the class of isoprobabilistic transformations $T(x_1,\ldots,x_k;\alpha)$, which transforms an absolutely continuous k-variate distribution $F(x_1,\ldots,x_k;\alpha)$ into the uniform distribution on the k-dimensional hyper-cube [23]. This gives an explicit formula to find the derivative of a random vector:

$$\frac{\partial X}{\partial \alpha} = -\left(\frac{\partial T(X,\alpha)}{\partial X}\right)^{-1}\frac{\partial T(X,\alpha)}{\partial \alpha}$$

assuming $\det\left(\frac{\partial T(X,\boldsymbol{\alpha})}{\partial X}\right) \neq 0$ so that the inverse exists.

For clarity, let us consider a 1-dimensional example. Suppose we have a random variable $X \sim F_X(x;\boldsymbol{\alpha})$, where F_X is invertible, then an isoprobabilistic transformation $T(X,\boldsymbol{\alpha})$ would simply be $F_X(X,\boldsymbol{\alpha})$ as $F_X(X,\boldsymbol{\alpha})$ is distributed uniformly. Hence, it follows that

$$\frac{\partial X}{\partial \boldsymbol{\alpha}} = -f_X(X,\boldsymbol{\alpha})^{-1}\frac{\partial F_X(X,\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}}.$$

It is easy to check via elementary means that this is indeed correct. Starting with the identity $F_X(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha}) = U$ and applying implicit differentiation, we have

$$\begin{aligned}
&\frac{\partial F_X(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}} = \mathbf{0} \\
\Rightarrow\quad &\frac{\partial F_X}{\partial X}(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha})\cdot\frac{\partial F_X^{-1}(U,\boldsymbol{\alpha})}{d\boldsymbol{\alpha}} + \frac{\partial F_X}{\partial \boldsymbol{\alpha}}(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha}) = \mathbf{0} \\
\Rightarrow\quad &\frac{\partial F_X^{-1}(U,\boldsymbol{\alpha})}{d\boldsymbol{\alpha}} = -\frac{\frac{\partial F_X}{\partial \boldsymbol{\alpha}}(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha})}{f_X(F_X^{-1}(U,\boldsymbol{\alpha}),\boldsymbol{\alpha})} \\
\Rightarrow\quad &\frac{\partial X}{\partial \boldsymbol{\alpha}} = -f_X(X,\boldsymbol{\alpha})^{-1}\frac{\partial F_X(X,\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}}.
\end{aligned}$$

Some specific one-dimensional cases include the gamma, inverse-gamma, chi-squared, Dirichlet, Wishart, and inverse-Wishart distributions.

2.2. Optimising AD Implementation

In the previous section, we introduced the vectorised AD formulation along with the full set of matrix calculus rules to support the dual construction. In this section, we explore several implementation strategies aimed at optimising execution. Benchmarking is carried out in R using the `ADtools` package available on CRAN ([24]) and GitHub ([25]). While the exact performance gains presented here (and in Section 3) are environment-specific, the optimisation principles are broadly applicable, and improvements can be expected in other environments.

2.2.1. Memoisation

Memoisation (or tabulation) is a technique for non-invasively attaching a cache to a function, allowing it to store and reuse previously computed results for repeated inputs [26].

It can greatly accelerate tasks such as constructing large structured matrices and provides a convenient way to organise computations.

The technique works by checking whether a given input has already been evaluated. If so, the cached result is returned; if not, the computation is performed, and the result is stored for future use. Table 2 shows a speed comparison of the built-in R function `diag`, with and without memoisation. An illustrative 12-line implementation is included in the Appendix B.

Table 2. Speed comparison of the diagonal matrix function with and without memoisation. `mem_diag` is memorised. The best results in each column are highlighted in bold.

Function	First Time	Second Time	Average over 100 Executions
`diag(5000)`	**97.87 ms**	127.7 ms	117.3 ms
`mem_diag(5000)`	101.8 ms	**0.132 ms**	**0.959 ms**

2.2.2. Sparse Matrix Representation

For efficient implementation, all special matrices are constructed and stored using sparse representations, which improve both computational and memory efficiency. A sparse matrix is typically represented as a list of triples, where each triple (i, j, v) records the value v at position (i, j) in the matrix.

2.2.3. The Diagonal Matrix D_n

An $n \times n$ diagonal matrix D_n is represented as $\{(k, k, v_k), k = 1, 2, \ldots, n\}$, where v_k is the kth diagonal entry of D_n. This representation takes $O(n)$ storage space and incurs an $O(n^2)$ computation cost when multiplied by a $n \times n$ dense matrix. A speed comparison of the diagonal matrix function with dense and sparse representations is provided in Table 3. In the table, "Dense" uses the R function `diag`, and "Sparse" uses the R function `ADtools::diagonal`. Using sparse representation, a substantial increase in speed was observed for large matrices.

Table 3. Speed comparison of the diagonal matrix function with dense and sparse representations. Time is averaged over 100 executions. The best results in each column are highlighted in bold.

Tasks	n = 1000	n = 2000	n = 3000	n = 4000	n = 5000
Creation—Dense	2.436 ms	17.39 ms	50.23 ms	63.86 ms	303.7 ms
Creation—Sparse	**0.080 ms**	**0.091 ms**	**0.110 ms**	**0.077 ms**	**0.089 ms**
Multiplication—Dense	19.40 ms	111.6 ms	404.4 ms	1.041 s	1.576 s
Multiplication—Sparse	**17.42 ms**	**66.50 ms**	**236.6 ms**	**432.3 ms**	**668.2 ms**

2.2.4. The Commutation Matrix K_{nq}

The $nq \times nq$ matrix K_{nq} is a commutation matrix if $\text{vec}(A^T) = K_{nq}\text{vec}(A)$ for any $n \times q$ matrix A [27]. From (7), we have

$$\begin{aligned} k\text{th entry of vec}(A) &= v_A^{-1}(k)\text{th entry of } A = (a, b)\text{entry of } A \\ &= (b, a)\text{entry of } A^T = v_{A^T}(b, a)\text{th entry of vec}(A^T) \\ &= [b + (a-1)q]\text{th entry of vec}(A^T) \end{aligned}$$

where $a = [(k-1) \mod n] + 1, b = \left\lceil \frac{k}{n} \right\rceil$. As K_{nq} is the matrix that maps $\text{vec}(A)$ to $\text{vec}(A^T)$, and we have derived that the kth entry of $\text{vec}(A)$ needs to map to $\left\lceil \frac{k}{n} \right\rceil + [(k-1) \mod n]q$-th entry of $\text{vec}(A^T)$, it follows that K_{nq} is a matrix having value

1 at position $\left(\left\lceil \frac{k}{n} \right\rceil + [(k-1) \bmod n]q, k\right), k = 1, 2, \ldots, nq$. Therefore, in the sparse representation, we have

$$K_{nq} = \left\{ \left(\left\lceil \frac{k}{n} \right\rceil + [(k-1) \bmod n]q, k, 1 \right), k = 1, 2, \ldots, nq \right\}. \tag{8}$$

It is worth noting that since the commutation matrix simply reorders the entries of $\text{vec}(A)$, one can implement a function that directly remaps the indices as specified in (8), rather than explicitly constructing the matrix and performing the associated matrix multiplication. Table 4 compares the performance of commutation matrix functions implemented using dense and sparse representations. The "Dense" implementation relies on the R function `matrixcalc::elimination.matrix`, while the "Sparse" version uses `ADtools::elimination_matrix`. A substantial improvement in speed is observed with the sparse approach.

Table 4. Speed comparison of the commutation matrix function with dense and sparse representations. Time is averaged over 100 executions. The best results in each column are highlighted in bold.

Tasks	**n = q = 10**	**n = q = 20**	**n = q =30**	**n = q =40**
Creation—Dense	10.77 ms	708.4 ms	18.09 s	120.4 s
Creation—Sparse	**0.646 ms**	**0.667 ms**	**1.553 ms**	**1.595 ms**
Multiplication—Dense	0.727 ms	41.54 ms	477.3 ms	2.821 s
Multiplication—Sparse	**0.189 ms**	**2.388 ms**	**11.48 ms**	**71.69 ms**

2.2.5. The Elimination Matrix E_n

Let E (and E_n) denote the elimination matrix and D the duplication matrix. These matrices are defined by the identities they satisfy:

$$\text{vech}(A) = E \ \text{vec}(A) \tag{9}$$

$$D \ \text{vech}(A) = \text{vec}(A), \ \text{for symmetric matrix } A \tag{10}$$

where $\text{vech}(.)$ is the half-vectorisation operator (vectorising the lower-triangular part of a square matrix). The names of the special matrices come from the fact that D duplicates entries to turn a half vector into a full vector, and E eliminates entries to turn a full vector into a half vector. Note that if A is an $n \times n$ matrix, then $\text{vec}(A)$ has a length n^2 and $\text{vech}(A)$ has a length of $\frac{n(n+1)}{2}$. Hence, D has a dimension of $n^2 \times \frac{n(n+1)}{2}$, and E has a dimension of $\frac{n(n+1)}{2} \times n^2$.

Now we derive the sparse representation of the elimination matrix. First, for an $n \times n$ matrix A, we define $h_A(i,j)$ and $h_A^{-1}(k)$ such that they satisfy the relations.

$$\begin{aligned} (i,j)\text{-entry of } A \quad &\Leftrightarrow \quad h_A(i,j)\text{-th entry of } \text{vech}(A) \\ h_A^{-1}(k)\text{-entry of } A \quad &\Leftrightarrow \quad k\text{-th entry of } \text{vech}(A), \end{aligned}$$

where $i \geq j$, $h_A^{-1}(k)$ must be in the lower triangular part of A. The functions $h_A(.,.)$ and $h_A^{-1}(.)$ are given by the formulae:

$$h_A(i,j) = i + (j-1)n - \frac{j(j-1)}{2} \quad \text{and} \quad h_A^{-1}(k) = (a,b), \tag{11}$$

where $a = k + \frac{b(b-1)}{2} - (b-1)n$, $b = f(k,n,1)$ and

$$f(k,n,c) = \begin{cases} f(k-n, n-1, c+1) & \text{if } k > n, \\ c & \text{otherwise.} \end{cases}$$

$h_A(.,.)$ and $h_A^{-1}(.)$ are needed to convert back and forth between the matrix and the half-vector representations. It then follows that:

$$\begin{aligned} k\text{th entry of vech}(A) &= h_A^{-1}(k)\text{entry of } A = v_A(h_A^{-1}(k))\text{th entry of vec}(A) \\ &= v_A\left(k + \frac{b(b-1)}{2} - (b-1)n, b\right)\text{th entry of vec}(A) \\ &= k + \frac{b(b-1)}{2}\text{th entry of vec}(A), \end{aligned}$$

where $b = f(k,n,1)$. Since by definition E_n maps vec(A) to vech(A), and we have shown $k + \frac{b(b-1)}{2}$th entry of vec(A) is mapped to kth entry of vech(A), so E_n is a matrix having a value of 1 at the position $\left(k, k + \frac{b(b-1)}{2}\right)$. Hence, the sparse representation of the elimination matrix is given by:

$$E_n = \left\{ \left(k, k + \frac{b(b-1)}{2}, 1\right), k = 1, 2, \ldots, \frac{n(n+1)}{2} \right\}.$$

In the actual implementation, b does not need to be computed recursively; it can be obtained directly using the closed-form expression $b = \left\lceil (n+0.5) - \sqrt{(n+0.5)^2 - 2k} \right\rceil$. A performance comparison of the elimination matrix function using dense and sparse representations is shown in Table 5. In the table, "Dense" uses the R function `matrixcalc::commutation.matrix`, and "Sparse" uses the R function `ADtools::commutation_matrix`. The results demonstrate a clear speed advantage for the sparse implementation, with performance gains increasing as matrix size grows.

Table 5. Speed comparison of the elimination matrix function with dense and sparse representations. Time is averaged over 100 executions. The best results in each column are highlighted in bold.

Tasks	**n = 10**	**n = 20**	**n = 30**	**n = 40**	**n = 50**
Creation—Dense	4.981 ms	113.1 ms	939.9 ms	4.735 s	20.97 s
Creation—Sparse	**0.823 ms**	**0.816 ms**	**0.823 ms**	**0.816 ms**	**0.891 ms**
Multiplication—Dense	0.412 ms	21.98 ms	237.0 ms	1.490 s	5.715 s
Multiplication—Sparse	**0.115 ms**	**1.723 ms**	**6.720 ms**	**22.45 ms**	**49.16 ms**

We also define the half-duplication matrix D^* for lower-triangular matrix A to be the matrix that satisfies D^* vech$(A) =$ vec(A), where A is a lower-triangular matrix. For any lower-triangular matrix A, we have D^* vech$(A) = E^T$ vech(A).

2.2.6. Matrix Chain Multiplication

In the implementation of vectorised AD, the derivative computation frequently involves sequences of matrix multiplications, as shown in Equation (1). While mathematically straightforward, these operations can become computational bottlenecks in high-dimensional settings. A key yet often overlooked aspect is that matrix multiplication, although associative in output—i.e., $(AB)C = A(BC)$—is not associative in computational cost. To illustrate, consider $n \times n$ matrices A, B and a $n \times 1$ vector x. Evaluating $(AB)x$ requires explicitly forming the intermediate matrix AB, resulting in a complexity $O(n^3)$. In

contrast, computing $A(Bx)$ avoids this and reduces the cost to $O(n^2)$. This simple example highlights a crucial insight: although the result is invariant to the order of operations, the efficiency is not. In large-scale computations, suboptimal ordering—such as naive left-to-right evaluation—can be unnecessarily costly. In simple cases where the dimensions of the matrices are known in advance, an optimal order can be enforced manually. However, in many applications, matrix dimensions are unknown until runtime, making it impossible to prespecify the optimal multiplication order. This leads to the matrix chain multiplication problem.

Matrix chain multiplication is an optimisation problem concerned with multiplying a chain of matrices $A_1 \cdot A_2 \cdot \ldots \cdot A_m$ using the least number of arithmetic operations. This is traditionally solved using dynamic programming with the complexity $O(2^m)$, which can be reduced to $O(m^3)$ when the memoisation technique is employed. Ref. [28] provides an algorithm that solves the problem with a $O(m \log m)$ complexity. However, given that in many applications the length of the matrix chain rarely goes beyond $m = 5$, it is usually sufficient to consider the simpler dynamic programming solution as follows:

Let $m(i, j)$ be the minimal number of arithmetic operations needed to multiply out a chain of matrices $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$, and suppose for any i, A_i has dimension $d_i \times d_{i+1}$. Our goal is to find $m(1, n)$. The recursive formula is given by [29]:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + d_i \cdot d_{k+1} \cdot d_{j+1} \} & \text{if } i < j \end{cases}$$

The above only provides the optimal number of arithmetic operations. To obtain the order of multiplication, we define the split point of the matrix chain $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$ as:

$$s(i,j) = \arg\min_{i \le k < j} \{ m(i,k) + m(k+1,j) + d_i \cdot d_{k+1} \cdot d_{j+1} \}.$$

For example, if $s(1, 4) = 2$, then the matrix chain $A_1A_2A_3A_4$ should be split after index 2 in the order of $(A_1A_2)(A_3A_4)$, whereas if $s(1, 4) = 3$, then the matrix chain is ordered as $(A_1A_2A_3)(A_4)$, after which one inspects $s(1, 3)$ to decide the full order.

Table 6 shows the increase in speed gained by switching from a naive (left-to-right) order to an optimal order. The comparison was conducted using 1000 simulations. The length of the chain was sampled from the set $\{2, 3, 4, 5\}$, with probabilities proportional to $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}\}$, and matrix sizes were sampled from the discrete uniform distribution $U[10, 200]$. Note that there is no speed-up by multiplying two matrices because there is only one possible order (the extra 0.04 is merely statistical noise). For a matrix chain with a length of three to five, the average speed-up was about 1.5 times. The speed-up distributions conditioning on the length of the chain were all positively skewed and had positive excess kurtosis (i.e., fat tails).

Table 6. Comparing multiplications of a chain of matrices in the naive and optimal orders. Figures represent the mean and standard deviation (in bracket) of the speed-up over 1000 simulations.

Length of Chain	**2**	**3**	**4**	**5**
Speed-up, $t_{naive}/t_{optimal}$	1.04 (0.322)	1.47 (0.564)	1.64 (0.688)	1.56 (0.637)

2.2.7. Kronecker Products

Among the basic matrix operations, the Kronecker product is one of the most computationally expensive. In general, computing the Kronecker product of an $(m \times n)$ matrix and a $p \times q$ matrix has a complexity of $O(mnpq)$. For simplicity, assume $m = n = p = q$; then the complexity becomes $O(n^4)$. In the context of Jacobian matrix computations, it is

rare to compute a standalone Kronecker product. Instead, it typically appears as part of a larger expression—for example, in forms such as $X(B \otimes A)$ and $(B \otimes A)Z$, where A, B are of size $n \times n$ and X, Z are of size $m \times n^2, n^2 \times m$.

If one first computes the Kronecker product explicitly and then multiplies it by the remaining matrix, the total complexity is $O(n^4 + n^4 m) = O(n^4 m)$. However, by exploiting structural properties and avoiding explicit computation of the Kronecker product, the same result can be obtained in $O(n^3 m)$ operations. We now show that this reduced complexity holds in the general case as well:

Proposition 1. *Suppose $A_1, A_2, \ldots, A_p$ are $n \times n$ matrices ($p \geq 2$), and X, Z are $m \times n^p$ and $n^p \times m$ matrices, respectively. Then $X(A_1 \otimes A_2 \otimes \ldots \otimes A_p)$ and $(A_1 \otimes A_2 \otimes \ldots \otimes A_p)Z$ can be computed in $O(n^{p+1} m)$ operations instead of $O(n^{2p} m)$ operations, as observed in the naive order.*

The proposition suggests that unless the Kronecker product itself is of interest, one should never compute it explicitly when it comes to multiplication because the algorithm (presented after the proof) always performs $(p-1)$ orders faster. The proposition also holds for cases in which $A_1, \ldots, A_p$ have arbitrary sizes, but we do not state the proposition in that form because it obscures the complexity improvement and logic of the proof. Nevertheless, the algorithm provided later does support the most general case.

Proof. We begin with the base case $(B \otimes A)Z$. Let $b_{i,j}$ be the (i, j) element of B and Z_i be the ith block row of Z (which is of size $n \times m$), then

$$(B \otimes A)Z = \begin{bmatrix} b_{1,1}A & b_{1,2}A & \ldots & b_{1,n}A \\ b_{2,1}A & b_{2,2}A & \ldots & b_{2,n}A \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1}A & b_{n,2}A & \ldots & b_{n,n}A \end{bmatrix} \cdot \begin{bmatrix} \text{———} & Z_1 & \text{———} \\ \text{———} & Z_2 & \text{———} \\ \vdots & \vdots & \vdots \\ \text{———} & Z_n & \text{———} \end{bmatrix} \tag{12}$$

$$= \begin{bmatrix} \text{———} & \sum_{j=1}^{n} b_{1,j} A Z_j & \text{———} \\ \text{———} & \sum_{j=1}^{n} b_{2,j} A Z_j & \text{———} \\ \vdots & \vdots & \vdots \\ \text{———} & \sum_{j=1}^{n} b_{n,j} A Z_j & \text{———} \end{bmatrix} \overset{denote}{:=} \left[\sum_{j=1}^{n} b_{k,j} A Z_j \right]_{k=1\ldots n} \tag{13}$$

For the kth block-row, we have $\sum_{j=1}^{n} b_{k,j} A Z_j = A \sum_{j=1}^{n} b_{k,j} Z_j$. Together with the multiplication by A, the sum can be computed in $O(n^2 m)$ operations, and because there are n block rows, the overall complexity is $O(n^3 m)$.

We now proceed to the general case by abstracting the component that makes the above work and applying it recursively to the chain of Kronecker products. If we define two binary operations $\boxdot$ and $\circledast$ such that:

$$A \boxdot V = A \boxdot [V_k]_{k=1\ldots n} \overset{def}{:=} [AV_k]_{k=1\ldots n} \quad \text{and}$$

$$B \circledast Z = B \circledast [Z_k]_{k=1\ldots n} \overset{def}{:=} \left[\sum_{j=1}^{n} b_{k,j} Z_j \right]_{k=1\ldots n}, \tag{14}$$

then $(B \otimes A)Z = A \boxdot (B \circledast Z) = [A(B \circledast Z)_k]_{k=1\ldots n}$. The key idea behind the two new binary operations is that they define block-wise matrix multiplication. Suppose both V and Z can be split into n by 1 blocks. The first binary operation $A \boxdot V$ defines the block-wise (pre-)multiplication, where each block of V is pre-multiplied by A, with A having the number of columns matching the number of rows of a block. The second binary operation $\circledast$ defines the block-wise matrix multiplication. This operation produces a matrix of $n \times 1$ blocks, where the i-th block is given by $\sum_{j=1}^{n} b_{i,j} Z_j$, naturally extending the usual matrix

multiplication $\sum_{j=1}^{n} b_{i,j}c_j$, where c_j is the jth entry of a column vector c. The last term in (14) is also denoted as $[(B \circledast Z)_k]_{k=1\ldots n}$ such that $(B \circledast Z)_k$ corresponds to the kth block-row $\sum_{j=1}^{n} b_{k,j}Z_j$.

The new binary operations allow us to evaluate the expression in a different order and avoid the Kronecker product in the process. As a result, it takes fewer arithmetic operations to evaluate the expression, as we have seen in the base case. Next, it follows that

$$\begin{aligned}(C \otimes B \otimes A)Z &= (C \otimes (B \otimes A))Z = (B \otimes A) \boxdot (C \circledast Z) \\ &= \left[(B \otimes A)(C \circledast Z)_{k_C}\right]_{k_C=1\ldots n} \\ &= \left[A \boxdot (B \circledast (C \circledast Z)_{k_C})\right]_{k_C=1\ldots n} \\ &= \left[\left[A(B \circledast (C \circledast Z)_{k_C})_{k_B}\right]_{k_B=1\ldots n}\right]_{k_C=1\ldots n},\end{aligned}$$

and the general case is given by

$$(A_1 \otimes A_2 \otimes \ldots \otimes A_p)Z$$
$$= \underbrace{\left[\left[\cdots\left[\right.\right.\right.}_{p-1} A_p(A_{p-1} \circledast (A_{p-2} \circledast (\cdots \circledast (A_1 \circledast Z)_{k_1} \cdots))_{k_{p-2}})_{k_{p-1}} \underbrace{\left.\left.\left.\right]_{k_{p-1}=1\ldots n} \cdots\right]_{k_2=1\ldots n}\right]_{k_1=1\ldots n}}_{p-1}.$$

Intuitively, every time we use the new binary operations to avoid a Kronecker product, the complexity is reduced by one in order, and given that there are $(p-1)$ of them, we expect to see the total complexity to be $O(n^{2p-(p-1)}m) = O(n^{p+1}m)$. We now present the formal proof by induction.

Let $S(p)$ be the statement that $(A_p \otimes A_{p-1} \otimes \ldots \otimes A_1)Z^{(p)}$ has complexity $O(n^{p+1}m)$, where $A_1, \ldots, A_p$ are $n \times n$ matrices, and $Z^{(p)}$ denotes a $n^p \times m$ matrix. We have shown in (12) and (13) that $S(2)$ is true. Now suppose $S(k)$ is true and consider $S(k+1)$,

$$(A_{p+1} \otimes A_p \otimes \ldots \otimes A_1)Z \tag{15}$$
$$= (A_p \otimes \ldots \otimes A_1) \boxdot (A_{p+1} \circledast Z^{(p+1)}) \tag{16}$$
$$= \left[(A_p \otimes \ldots \otimes A_1)(A_{p+1} \circledast Z^{(p+1)})_k\right]_{k=1\ldots n} \tag{17}$$
$$= \left[(A_p \otimes \ldots \otimes A_1)Z_k^{(p)}\right]_{k=1\ldots n} \tag{18}$$

In (17), computing $(A_{p+1} \circledast Z^{(p+1)})_k$ requires $O(n^{p+1}m)$ operations, resulting in a matrix $Z_k^{(p)}$ of size $n^p \times m$. Next, in (18), the expression inside the square bracket, by hypothesis, has a complexity of $O(n^{p+1}m)$, and the complexity accumulated so far remains as $O(n^{p+1}m)$. Finally, given that there are n block-rows, the overall complexity is $O(n^{p+2}m)$. This proves the inductive step, and by induction, $S(n)$ is true for $n \geq 2$. This completes the proof for the case $(A_1 \otimes A_2 \otimes \ldots \otimes A_p)Z$ in Proposition 1.

For the other case, $X(A_1 \otimes A_2 \otimes \ldots \otimes A_p)$, because we are premultiplying the chain of Kronecker products, we work with blocks of columns instead of blocks of rows. Denote

$$X = \begin{bmatrix} | & | & \ldots & | \\ X_1 & X_2 & \ldots & X_n \\ | & | & \ldots & | \end{bmatrix}$$

by $[X_k]_{k=1\ldots n}^{c}$ (where c stands for columns). Then the two corresponding binary operators $\boxdot^c$ and $\circledast^c$ are defined as:

$$V \boxdot^c A = [V_k]_{k=1\ldots n}^c \boxdot^c A \overset{def}{:=} [V_k A]_{k=1\ldots n}^c, \quad \text{and}$$
$$Z \circledast^c B = [Z_k]_{k=1\ldots n}^c \circledast^c B \overset{def}{:=} \left[\sum_{i=1}^{n} b_{ik} Z_i\right]_{k=1\ldots n}.$$

Now, it follows that $X(B \otimes A) = (X \circledast^c B) \boxdot^c A$, and the remainder of the proof proceeds the same as in the other case. □

In Table 7, we compare the performance of evaluating $(B \otimes A)Z$ and $X(B \otimes A)$ with and without explicitly computing the Kronecker product. We conducted 1000 simulations, and in each simulation, the number of rows of X, A, B, and the number of columns of A, B, Z were sampled from ζ, where $\zeta \sim U[10, 50]$. Obviously, the number of columns of X needs to match the number of rows of $B \otimes A$ (= the number of rows of $B \times$ the number of rows of A), and likewise for the number of rows of Z. The speed-up was computed using $t_{explicit}/t_{implicit}$, which denotes the time needed to evaluate the full expression using explicit and implicit Kronecker products, respectively. We note that in the case of $(B \otimes A)Z$, there were two speed-up "outliers": one at 0.42 and, the other at 0.94. All the rest were above 1. The median speed-up was about 15X, and the mean speed-up was about 16X, favouring the implicit evaluation.

Table 7. Speed-up achieved by evaluating $(B \otimes A)Z$ and $X(B \otimes A)$ without explicitly calculating the Kronecker product. The speed-up is computed using $t_{explicit}/t_{implicit}$. The number of simulations is 1000.

Tasks/Percentiles	0%	25%	50%	75%	100%	Mean
$(B \otimes A)Z$	0.42	9.43	14.02	19.99	57.48	15.82
$X(B \otimes A)$	2.65	10.82	15.44	21.34	54.30	16.81

2.2.8. Kronecker Product: More Special Cases

We identified common special cases of the Kronecker product and represented those using the new binary operators to reduce the computation cost further. In particular, we examine the four cases: $A(B \otimes I)$, $A(I \otimes C)$, $(B \otimes I)D$, and $(I \otimes C)D$. These are chosen because they arise naturally in common operations such as the product rule $d(AB) = (B^T \otimes I_n)dA + (I_n \otimes A)dB$ and the transpose product rule $d(AA^T) = (I \otimes A + A \otimes I)$. Moreover, computing a Kronecker product (explicitly) with an identity matrix merely makes copies of the original matrix and arranging them in a particular way, hence yielding potential savings in time and memory use (despite that the complexity order remaining the same).

Note that $I \boxdot V = [IV_k]_{k=1\ldots n} = [V_k]_{k=1\ldots n} = V$ and $I \circledast Z = [Z_k]_{k=1\ldots n} = Z$, so

1. $(B \otimes I)D = I \boxdot (B \circledast D) = B \circledast D$
2. $(I \otimes C)D = C \boxdot (I \circledast D) = C \boxdot D$

Similarly, $V \boxdot^c I = [V_k I]_{k=1\ldots n} = [V_k]_{k=1\ldots n} = V$ and $Z \circledast^c I = [Z_k]_{k=1\ldots n} = Z$, so

3. $A(B \otimes I) = (A \circledast^c B) \boxdot^c I = A \circledast^c B$
4. $A(I \otimes C) = (A \circledast^c I) \boxdot^c C = A \boxdot^c C$

In Table 8, we present the speed-up using the optimised implementation over the naive implementation. We conducted 1000 simulations, and in each simulation, the number of rows of A, B, C, I and the number of columns of B, C, D, I were sampled from ζ, where $\zeta \sim U[10, 50]$. The number of columns of A and the number of rows of D were specified such that the multiplication makes sense (and the choice is unique). The speed-up was computed using $t_{optimised}/t_{naive}$, which denotes the evaluation time corresponding to the

optimised and the naive implementations, respectively. We observe that the median speed-up is about 8×, While the mean speed-up is about 10×, this aligns with our theoretical result that the Kronecker product should not be evaluated explicitly unless the product itself is of interest.

Table 8. Speed-up achieved evaluating $(B \otimes I)D$, $(I \otimes C)D$, $A(B \otimes I)$ and $A(I \otimes C)$ without explicitly computing the Kronecker product. The speed-up is computed using $t_{optimised}/t_{naive}$. The number of simulations is 1000.

Tasks/Percentiles	0%	25%	50%	75%	100%	Mean
$(B \otimes I)D$	1.51	5.02	7.79	12.23	51.33	9.48
$(I \otimes C)D$	1.95	5.06	7.83	11.62	79.33	9.48
$A(B \otimes I)$	2.3	5.88	8.54	12.9	68.36	10.25
$A(I \otimes C)$	2.22	5.13	8.25	12.71	61.62	9.98

3. Results

This section provides some computational examples to demonstrate the speed and efficacy of our proposed methods. We first benchmark our method against the traditional numerical derivative under basic matrix operations and the computation of a covariance matrix's log determinant. We then demonstrate our derivative computation's effectiveness within a large stochastic optimisation scheme, i.e., simulated maximum likelihood estimation (SMLE) of a stochastic factor model.

3.1. Basic Operations

We benchmarked the performance of AD against that of FD using the basic arithmetic operations addition: subtraction, multiplication, matrix inversion, and Kronecker product. The results are presented in Table 9. The time figures in the table represent the averages of 100 executions. Overall, AD performed much faster compared with the FD.

Table 9. Benchmarking of AD against central FD in terms of basic arithmetic operations. Faster times are in bold.

	Estimation Time (in ms) for Standard Tasks Under AD and FD by Matrix Size									
Tasks	**AD**	**FD**	**AD**	**FD**	**AD**	**FD**	**AD**	**FD**	**AD**	**FD**
	n = 10		n = 20		n = 30		n = 40		n = 50	
Addition	**4.61**	22.98	**5.64**	138.17	**16.15**	435.70	**73.76**	1136.07	**146.96**	2746.58
Subtraction	**3.92**	21.91	**5.42**	107.32	**19.33**	427.66	**77.41**	1236.07	**167.06**	2970.38
Multiplication	**13.60**	23.40	**20.48**	141.94	**38.36**	580.23	**139.87**	1732.45	**245.83**	4251.67
Inverse	**2.46**	20.94	**8.09**	93.98	**29.20**	334.35	**156.82**	945.50	**444.41**	1839.47
	n = 5		n = 10		n = 15		n = 20		n = 25	
Kronecker	15.1	**9.3**	**54.2**	181.7	**351.9**	1756.4	**1478.6**	7492.1	**5530.5**	31,950.7

3.2. Dynamic Factor Model Inference

Numerical assessments are often required in both classical and Bayesian statistical inference. Below, we illustrate the benefits of applying AD to derivative computation for the maximum likelihood estimation (MLE) of factor models when the analytical expression of the likelihood is not available and numerical assessment of derivatives is required. We show substantial computational gains using both simulated and real data. Readers interested in the use of AD in the context of Bayesian sensitivity analysis are referred to [30].

Factor models have been widely used in many areas, including psychology, bioinformatics, economics, and finance, to model the dependence structure of high-dimensional data. Different specifications of the factor model have been widely discussed in the liter-

ature ([31,32]). We follow [32] and consider a variation of the factor model in which the analytic expression of the derivative of the log-likelihood is intractable. Let $\mathbf{y}_t$ denote the $n \times 1$ vector of observations at time t where $t = 1, \ldots, T$, and let $\mathbf{f}_t$ represent a $k \times 1$ column vector of latent factors. Then, the k-factor model with student-t noise is specified as:

$$\mathbf{y}_t = \beta + \mathbb{A}\mathbf{f}_t + \epsilon_t, \tag{19}$$

where β is the $n \times 1$ column vector of intercepts and $\mathbb{A}$ is the $n \times k$ loading matrix. The factors are assumed to be normally distributed, $\mathbf{f}_t \sim N(\mathbf{0}, \Omega)$, where Ω is $k \times k$, and they are independent from innovations that are multivariate-t distributed $\epsilon_t \sim t_\nu(\mathbf{0}, \Sigma)$, where Σ is $n \times n$. For the purpose of identification, we require $n \geq 2k + 1$ and assume that $\mathbb{A}$ is lower triangular with diagonal entries all equal to 1 ([32]). This particular specification is commonly used in financial econometrics.

For maximum likelihood inference of such a model, the likelihood function can be maximised directly via Monte Carlo simulation. Let $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, ..\mathbf{y}_T)'$ be the observations and $\boldsymbol{\theta} = [\beta', \text{vech}(\mathbb{A})', \text{vech}(\Omega), \text{vech}(\Sigma)]'$ be the parameters. Under our setting, the likelihood function $g(y_t; \boldsymbol{\theta})$ does not have an analytical expression, and it needs to be evaluated using numerical methods. Specifically, we can write $\mathbf{f}_t = \Omega^{1/2}\mathbf{Z}_t$ where $\mathbf{Z}_t \sim N(\mathbf{0}, I_k)$. Therefore, it follows that:

$$g(y_t; \boldsymbol{\theta}) = \mathbb{E}_{\mathbf{Z}}[t_\nu(y_t | \beta + \mathbb{A}\Omega^{1/2}\mathbf{Z}_t, \Omega)],$$

where $t_\nu(\cdot|\mu, \Sigma)$ denotes the probability density function of the multivariate-t distribution. In the rest of this section, we focus on a case in which both Σ and Ω are diagonal matrices: $\Sigma = \text{diag}(\sigma_1^2, \ldots, \sigma_n^2)$ and $\Omega = \text{diag}(\omega_1^2, \ldots, \omega_k^2)$.

For the implementation of the simulated maximum likelihood approach, we use the Ada-Delta variation of the stochastic gradient descent algorithm [33] for the estimation (with hyperparameters $(\gamma, \eta, \epsilon) = (0.9, 0.01, 10^{-8})$). We first considered a simulated dataset of 1000 observations, each with 10 measurements, where the dimensions of the hidden factors were 3 and the entries in the factor loading matrix $\mathbb{A}$ were sampled from the standard normal distribution. The entries of the diagonal covariance matrices of the factors and the innovations were sampled from $U(1, 5)$ and $U(0.5, 1)$, respectively. The estimates converged at around 2000 iterations.

The first column of results in Table 10 reports the total run times with derivatives computed using AD and, for comparison, FD, for the simulated data example.

Table 10. Run-time comparison between SMLE analysis of the factor model using either AD or (central) FD in the stochastic gradient computations. The per-iteration summaries are based on 100 evaluations under the simulated data example. The best performance in each column is highlighted in bold.

Run-Time Comparison of Simulated MLE								
	Total Run-Time		**Per Iteration Run-Time (Simulated Data)**					
	Simulated Data	**Real Data**	**min**	**lq**	**Mean**	**Median**	**uq**	**max**
AD	**4.36 h**	**2.08 h**	**7.23 s**	**7.41 s**	**7.52 s**	**7.47 s**	**7.56 s**	**8.37 s**
FD	12.04 h	6.10 h	20.35 s	20.51 s	20.76 s	20.60 s	20.69 s	26.82 s

Under AD, the estimation took 4.36 h, compared with over 12 h using FD for the required derivative calculation. This reduction in run-time by almost 66% is confirmed by the remaining results in Table 10, which provide a more detailed run-time comparison between the two implementations, confirming the patterns from the overall run-time. The

improvement was consistent for both per-iteration run-time and total run-time in the case of simulated data and for the total run-time for simulated to real data.

The real data set contained data on currency exchange rates. The sample included 1045 observations of daily returns of nine international currency exchange rates relative to the United States dollar from January 2007 to December 2010. We applied a factor model to the log of the exchange rate returns (i.e., $y_{it} = 100 \log(p_{i,t}/p_{i,t-1})$, where p_{it} denotes the daily closing spot rate for currency i at time t. The nine selected currencies were the Australian dollar (AUD), Canadian dollar (CAD), Euro (EUR), Japanese yen (JPY), Swiss franc (CHF), British pound (GBP), South Korean won (KRW), New Zealand dollar (NZD), and New Taiwan dollar (TWD), representing the most heavily traded currencies over the period. The estimates converged at around 1000 iterations. As reported in the second column of results in Table 10, we again observed a substantial reduction in estimation time, with the run time under an AD-based derivative computation being a third of the time required by the FD-based computation.

4. Conclusions

This paper presents a vectorised formulation of AD grounded in matrix calculus, tailored for statistical applications that involve high-dimensional inputs and simulation-intensive computations. The proposed approach uses a compact set of matrix calculus rules to enable efficient and automatic derivative computation and introduces optimisation techniques, such as memoisation, sparse matrix representation, matrix chain multiplication, and implicit Kronecker product, to improve the efficiency of the AD implementation.

Compared to other AD approaches, our formulation aligns more naturally with the matrix-oriented notation commonly used in statistics and econometrics. It supports fully automatic workflows similar to source code transformation methods while also providing direct access to intermediate variables for inspection and analysis. Unlike imperative AD frameworks, it does not require users to adopt AD-specific programming idioms. In addition, the approach enables high-level optimisation by explicitly making use of matrix structure and the order of matrix multiplications, which are typically inaccessible in scalar-based or imperative implementations.

Despite its advantages, our formulation of AD introduces computational overhead, which can make it less efficient than FD for small-scale or low-dimensional problems. In such cases, the simplicity of FD often results in faster performance (but at the cost of lower accuracy). The benefits of our approach become more apparent in high-dimensional settings, where its scalability and accuracy outweigh the initial overhead, as we showed in the numerical study. A further limitation arises when computing second-order derivatives such as the Hessian. Under our vectorised approach, this requires dual numbers to carry second-order matrix derivatives, which can quickly exceed the memory capacity of typical personal computers. For a function mapping $\mathbb{R}^m$ to $\mathbb{R}^n$, the Jacobian has dimension $n \times m$, while the Hessian grows to $nm \times m$. In contrast, FD computes these matrices entry by entry by perturbing the function input one coordinate at a time; although slower, this method avoids out-of-memory issues. Similar memory constraints may also occur when handling extremely large Jacobian matrices.

In view of the modern trend toward increasingly large and high-dimensional datasets, the performance overhead of AD in small-scale settings is becoming less of a practical concern. As data and models continue to grow in complexity, the scalability advantages of AD are expected to outweigh its initial costs in a broader range of applications. For the memory demands associated with higher-order derivatives, potential solutions include distributed computing, on-disk array storage, and blocked algorithms that process derivative computations in smaller, memory-efficient segments. These strategies offer promising

avenues for extending the applicability of our vectorised AD approach to even larger and more demanding statistical problems.

Author Contributions: Conceptualisation, C.F.K. and D.Z.; methodology, C.F.K. and D.Z.; software, C.F.K.; validation, C.F.K. and D.Z.; formal analysis, C.F.K.; investigation, C.F.K.; resources, L.J. and D.Z.; data curation, C.F.K. and D.Z.; writing—original draft preparation, C.F.K.; writing—review and editing, C.F.K., L.J. and D.Z.; visualisation, C.F.K.; supervision, L.J. and D.Z.; project administration, L.J. and D.Z.; funding acquisition, L.J. and D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: The authors acknowledge support from the Australian Research Council through funding from DP180102538 and FT170100124.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The exchange rate data used in this study is publicly available from the Federal Reserve Economic Data (FRED) at https://fred.stlouisfed.org and is freely accessible without restrictions.

Acknowledgments: We thank the anonymous reviewers for their valuable comments and constructive suggestions.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Appendix A. Illustrative Examples

In this section, we present some statistical applications that use the matrix operations that could benefit significantly from the use of AD.

Example A1. *Local sensitivity of the Seemingly Unrelated Regression (SUR) model [34]. Consider the SUR model,*

$$y_\mu = X_\mu \beta_\mu + u_\mu, \quad \mu = 1, \ldots, M,$$

- *where* $y_\mu, u_\mu \in \mathbb{R}^T, X_\mu \in \mathbb{R}^{T \times l_\mu}, \beta_\mu \in \mathbb{R}^{l_\mu}$, *and*
- $E(u_\mu) = 0, V(u_\mu) = \sigma_{\mu\mu} I$ *and* $E(u_i u_j) = \sigma_{ij} I$, I *is the identity matrix.*

In a more compact form,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} = \begin{bmatrix} X_1 & 0 & \cdots & 0 \\ 0 & X_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & X_M \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_M \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_M \end{bmatrix}$$

and we write it as $y = X\beta + u$, *where* $V(u) = \Sigma_c \otimes I, \Sigma_c = [\sigma_{ij}]_{i,j=1,\ldots,M}$. *The Generalised-Least-Squares (GLS) estimator is given by* $\hat{\beta} = (X^T(\Sigma_c \otimes I)^{-1}X)^{-1}X^T(\Sigma_c \otimes I)^{-1}y$. *Given the matrix multiplications, inversions and Kronecker products involved, it is tedious to find the analytical expression of the local sensitivity* β *with respect to all the noise parameters* σ_{ij}, *i.e.,* $\frac{d\hat{\beta}}{d\Sigma_c}$. *In contrast, AD only requires implementing the original expression and then the derivative will be available "for free".*

Example A2. *Local sensitivity of the Bayesian normal regression model. Consider the model* $y \sim N(X\beta, V)$ *with the normal prior on the parameter* $\beta \sim N(b^*, H^{*-1})$ *and with* b^*, H^* *being the hyperparameters. The posterior mean* b *of* β *is given by*

$$b = H^{-1}(H^* b^* + X^T V^{-1} y), \quad \text{where } H = H^* + X^T V^{-1} X. \tag{A1}$$

The local sensitivity of the posterior mean is concerned with the effect of a small change of the hyperparameters V^{-1}, b^*, H^{*-1} *and the data* X *on the posterior mean* b.

Even in such simple case, it is clear that applying AD directly to (A1) is less error-prone than deriving and implementing the analytic derivative by hand:

$$\begin{aligned}\mathbf{d}b = &\left[(b-b^*)' H^* \otimes H^{-1} H^*\right] D_k \mathbf{d}v\left(H^{*-1}\right) + H^{-1} H^* \mathbf{d}b^* + \\ &\left[H^{-1} \otimes e' V^{-1} - b' \otimes H^{-1} X' V^{-1}\right] \mathbf{d}\,\mathrm{vec}\, X + \left[e' \otimes H^{-1} X'\right] D_n \mathbf{d}v\left(V^{-1}\right),\end{aligned}$$

where $\mathbf{d}$ *is the standard differential operators,* $e = y - Xb$, *and* D_n, D_k *are duplication matrices of appropriate dimensions.*

Example A3. *(Mixed-regressive) Spatial-Auto-Regressive (SAR) model [35] (pp. 8, 16).*

$$y = X\beta + \sum_{j=1}^{p} \lambda_j W_j y + \epsilon, \quad \epsilon \sim N(0, \sigma^2 I_m)$$

where y, ϵ *are* $m \times 1$ *vector,* X *is a* $m \times k$ *matrix,* β *is a* $k \times 1$ *vector,* $\{\lambda_j, j = 1, \ldots, p\}$ *are scalars, and* $\{W_j, j = 1, \ldots, p\}$ *are* $m \times m$ *spatial weight matrices. The loglikelihood function* $l(\lambda_1, \ldots, \lambda_p, \beta, \sigma^2)$ *is given by:*

$$-\frac{m}{2}\ln 2\pi\sigma^2 + \ln\left|I_m - \sum_{j=1}^{p}\lambda_j W_j\right| - \frac{1}{2\sigma^2}\left(\left[I_m - \sum_{j=1}^{p}\lambda_j W_j\right] y - X\beta\right)^T \left(\left[I_m - \sum_{j=1}^{p}\lambda_j W_j\right] y - X\beta\right)$$

and the derivative is needed to perform MLE using gradient-based methods. In addition to the convenience of not needing to implement the derivative manually, AD often has the advantage of having less duplicate computation compared with when the loglikelihood function and its derivative are implemented separately.

Example A4. *MLE with Simultaneous equations [19] (p. 371). Simultaneous equation model is a generalisation of the multivariate linear regression model*

$$y_i^T, = x_i^T B_0 + u_i^T, i = 1, \ldots, n,$$

where $y_i, u_i \in \mathbb{R}^m, x_i \in \mathbb{R}^k$ *are column vectors, and* B_0 *is a* $k \times m$ *matrix. and it has the form of*

$$y_i^T \Gamma_0 + x_i^T B_0 + u_i^T, i = 1, \ldots, n,$$

where Γ_0 *is a* $m \times m$ *matrix. Assuming that* $u_i \sim N(0, \Sigma_0), i = 1, \ldots, n$ *and the* $n \times k$ *data matrix has full rank* k, *the loglikelihood consists of the cross-product, determinant and trace operations as follows:*

$$l(\theta) = -\frac{mn}{2}\log 2\pi + \frac{n}{2}|\Gamma_0^T \Gamma_0| - \frac{n}{2}\log|\Sigma_0| - \frac{1}{2}\,\mathrm{tr}\,\Sigma_0^{-1}(Y\Gamma_0 + XB_0)^T(Y\Gamma_0 + XB_0).$$

In the above, the parameters Γ_0, B_0 *are collected into* θ, *and* X, Y *are* $\{x_i^T, y_i^T\}_{i=1,\ldots,n}$ *stacked by rows.*

For the multilevel generalisation [36] where the observations are clustered into l independent groups (of the same size $n_l = n/l$), the loglikelihood is given by:

$$l(\theta) = -\frac{nm}{2}\ln(2\pi) - \frac{ml}{2}\ln|V_0| - \frac{n}{2}\ln\left|\left(\Gamma_0^{-1}\right)^T\Sigma_0\Gamma_0^{-1}\right| - \frac{1}{2}\sum_{j=1}^{l}\operatorname{tr}\left(V_0^{-1}\left(Y_j - X_jB_0\Gamma_0^{-1}\right)\Gamma_0\Sigma_0^{-1}\Gamma_0^T\left(Y_j - X_jB_0\Gamma_0^{-1}\right)^T\right).$$

Note that U_j is $\{u_i^T\}_{i=1,\ldots,n_l}$ stacked by rows, and it follows a matrix normal distribution $N_{n_l,m}(0, V_0, \Sigma_0)$.

In the example above, AD offers an easy way to extend existing model to incorporate structural assumptions, which often leads a more complicated derivative expression. It enables researchers to readily experiment with different ways of generalising the working model.

Example A5. *Infinite Gaussian Mixture model. Consider the model $y \sim N(x\beta, r^{-1}), r^{-1} \sim \Gamma(k,\theta)$, where x, y are the data and r, k, θ are the parameters. Let $f(y;\mu,\sigma^2)$ be the normal density and $g(r;k,\theta)$ be the gamma density, the log-likelihood is given by:*

$$\begin{aligned} l(\beta,k,\theta) &= \log\int_0^\infty f(y;x\beta,r^{-1})g(r^{-1};k,\theta)dr^{-1} \\ &\approx \log\frac{1}{N}\sum_{i=1}^{N} f(y;x\beta,r_i^{-1}), \quad r_i^{-1} \sim g(r^{-1};k,\theta), \end{aligned}$$

where the second line is the Monte Carlo approximation. As the simulated log-likelihood depends on parameters through the random sample, it is more convenient to use AD to compute the derivative, especially if one wants to explore different choices of the mixture distribution g. Moreover, it is also practical to use AD when the mixture models are multi-level, or when the marginalised parameters are high-dimensional.

Appendix B. Code Listings

Listing A1. Implementation of the subtraction and inverse matrix calculus rules in R.

```
`%minus%` <- function(A_dual, B_dual) {
  A <- A_dual$X;  dA <- A_dual$dX;
  B <- B_dual$X;  dB <- B_dual$dX;
  list(X = A - B, dX = dA - dB)
}

`%divide%` <- function(A_dual, B_dual) {
  A <- A_dual$X;  dA <- A_dual$dX;
  B <- B_dual$X;  dB <- B_dual$dX;

  B_inv <- solve(B)
  dB_inv <- -(t(B_inv) %x% B_inv) %*% dB

  B_inv_dual <- list(X = B_inv, dX = dB_inv)
  A_dual %times% B_inv_dual
}
```

Listing A2. An example using the simple AD system in Section 2.1.2.

```
f <- function(A, B) {
    A %*% (A %*% B + B %*% B) + B
}

# Derivative by Auto-Differentiation
```

```
df_AD <- function(A, B) {  # A and B are dual matrices
    A %times% ((A %times% B) %plus% (B %times% B)) %plus% B
}

# Derivative by Analytic Formula
df_AF <- function(A, B, dA, dB) {
    # optimisation by hand to avoid repeated computation
    I_n <- I(nrow(A))
    I_n2 <- I(nrow(A)^2)
    In_x_A <- I_n %x% A
    In_x_B <- I_n %x% B
    tB_x_In <- t(B) %x% I_n
    # the analytic formula
    (t(A %*% B + B %*% B) %x% I_n + (In_x_A) %*% tB_x_In) %*% dA +
        (In_x_A %*% In_x_A + In_x_A %*% (tB_x_In + In_x_B) + I_n2) %*% dB
}

## ------------------------------------------------------------
# Helper functions
zeros <- function(nr, nc) matrix(0, nrow = nr, ncol = nc)
dual <- function(X, dX) list(X = X, dX = dX)

# Main code
n <- 10
set.seed(123)
A <- matrix(rnorm(n^2), nrow = n, ncol = n)
B <- matrix(rnorm(n^2), nrow = n, ncol = n)
res <- f(A, B)

dA <- cbind(I(n^2), zeros(n^2, n^2))
dB <- cbind(zeros(n^2, n^2), I(n^2))
res_DF <- df_AF(A, B, dA, dB)              # Analytic approach
res_AD <- df_AD(dual(A, dA), dual(B, dB))  # AD approach

# Compare accuracy
sum(abs(res_AD$X - res))        # 0
sum(abs(res_AD$dX - res_DF))    # 5.016126e-13
```

Listing A3. An illustrative implementation of one-argument memoisation in R.

```
memoise <- function(f) {      # takes a function ‘f’ as input
  record <- list()            # attach a table to ‘f’ (using lexical scoping)
  hash <- as.character
  return(function(x) {        # returns a memoised ‘f’ as output
    result <- record[[hash(x)]]     # retrieves result
    if (is.null(result)) {          # if the result does not exist
      result <- f(x)                # then evaluate it and
      record[[hash(x)]] <<- result  # save it for future
    }
    return(result)
  })
}
```

Listing A4. R code to compare the speed and accuracy of AD and FD.

```
# remotes::install_github("kcf-jackson/ADtools")
library(ADtools)

# 1. Setup
set.seed(123)    # for reproducibilit
X <- matrix(rnorm(10,000), 100, 100)
Y <- matrix(rnorm(10,000), 100, 100)
B <- matrix(rnorm(10,000), 100, 100)
```

```
f <- function(B) { sum((Y - X %*% B)^2) }
# Deriving analytic derivative by hand
df <- function(B) { -2 * t(X) %*% (Y - X %*% B) }


# 2. Speed comparison
system.time({
  AD_res <- auto_diff(f, at = list(B = B))
})
# user  system elapsed
# 0.387   0.054   0.445

system.time({
  FD_res <- finite_diff(f, at = list(B = B))
})
# user  system elapsed
# 10.660   1.918  12.591

system.time({
  truth <- df(B)      # runs fastest when available
})
# user  system elapsed
# 0.001   0.000   0.001


# 3. Accuracy comparison
AD_res <- as.vector(deriv_of(AD_res))
FD_res <- as.vector(FD_res)
truth  <- as.vector(truth)

max(abs(AD_res - truth))
# [1] 0
max(abs(FD_res - truth))
# [1] 0.006982282
```

Listing A5. R code to illustrate that our vectorised formulation can produce derivative automatically and seamlessly.

```
# Example 1: Seemingly Unrelated Regression
set.seed(123)
T0 <- 10
M <- 5
l <- 6

# Regression coefficients
beta <- do.call(c, lapply(1:M, \(id) rnorm(l, mean = 0, sd = 2)))

# Predictors
Xs <- lapply(1:M, \(id) matrix(rnorm(T0 * l), nrow = T0, ncol = l))
X <- diag(1, nrow = M * T0, ncol = M * l)
for (i in seq_along(Xs)) {
  X[1:T0 + (i-1) * T0, 1:l + (i-1) * l] <- Xs[[i]]
}
X

# Noise
Sigma_c <- crossprod(matrix(rnorm(M^2), nrow = M))
I <- diag(T0)
u <- mvtnorm::rmvnorm(1, mean = rep(0, T0 * M),
                      sigma = kronecker(Sigma_c, I))

# Observation
```

```
y <- X %*% beta + t(u)

# Estimator
estimator <- function(Sigma_c, I, X, y) {
  inv_mat <- solve(kronecker(Sigma_c, I))
  beta_est <- solve(t(X) %*% inv_mat %*% X, t(X) %*% inv_mat %*% y)
}

# remotes::install_github("kcf-jackson/ADtools")
library(ADtools)
auto_diff(estimator,
          wrt = c("Sigma_c"),
          at = list(Sigma_c = Sigma_c, I = I, X = X, y = y))
```

References

1. Gardner, J.; Pleiss, G.; Weinberger, K.Q.; Bindel, D.; Wilson, A.G. Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 7587–7597.
2. Abril-Pla, O.; Andreani, V.; Carroll, C.; Dong, L.; Fonnesbeck, C.J.; Kochurov, M.; Kumar, R.; Lao, J.; Luhmann, C.C.; Martin, O.A.; et al. PyMC: A modern, and comprehensive probabilistic programming framework in Python. *PeerJ Comput. Sci.* **2023**, *9*, e1516. [CrossRef] [PubMed]
3. Joshi, M.; Yang, C. Algorithmic Hessians and the fast computation of cross-gamma risk. *IIE Trans.* **2011**, *43*, 878–892. [CrossRef]
4. Allen, G.I.; Grosenick, L.; Taylor, J. A generalized least-square matrix decomposition. *J. Am. Stat. Assoc.* **2014**, *109*, 145–159. [CrossRef]
5. Jacobi, L.; Joshi, M.S.; Zhu, D. Automated sensitivity analysis for Bayesian inference via Markov chain Monte Carlo: Applications to Gibbs sampling. *SSRN* **2018**. [CrossRef]
6. Ranganath, R.; Gerrish, S.; Blei, D. Black box variational inference. In Proceedings of the Artificial Intelligence and Statistics. PMLR, Reykjavik, Iceland, 22–25 April 2014; pp. 814–822.
7. Revels, J.; Lubin, M.; Papamarkou, T. Forward-mode automatic differentiation in Julia. *arXiv* **2016**, arXiv:1607.07892.
8. Baydin, A.G.; Pearlmutter, B.A.; Radul, A.A.; Siskind, J.M. Automatic differentiation in machine learning: A survey. *J. Mach. Learn. Res.* **2018**, *18*, 1–43.
9. Kucukelbir, A.; Tran, D.; Ranganath, R.; Gelman, A.; Blei, D.M. Automatic differentiation variational inference. *J. Mach. Learn. Res.* **2017**, *18*, 1–45.
10. Chaudhuri, S.; Mondal, D.; Yin, T. Hamiltonian Monte Carlo sampling in Bayesian empirical likelihood computation. *J. R. Stat. Soc. Ser. B Stat. Methodol.* **2017**, *79*, 293–320. [CrossRef]
11. Chan, J.C.; Jacobi, L.; Zhu, D. Efficient selection of hyperparameters in large Bayesian VARs using automatic differentiation. *J. Forecast.* **2020**, *39*, 934–943. [CrossRef]
12. Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; Lerer, A. Automatic differentiation in PyTorch. In Proceedings of the NIPS 2017 Autodiff Workshop, Long Beach, CA, USA, 9 December 2017.
13. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. {TensorFlow}: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
14. Kucukelbir, A.; Ranganath, R.; Gelman, A.; Blei, D. Automatic variational inference in Stan. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 7–12 December 2015; pp. 568–576.
15. Klein, W.; Griewank, A.; Walther, A. Differentiation methods for industrial strength problems. In *Automatic Differentiation of Algorithms*; Springer: New York, NY, USA, 2002; pp. 3–23.
16. Griewank, A.; Walther, A. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*; Siam: Philadelphia, PA, USA, 2008; Volume 105.
17. Griewank, A.; Juedes, D.; Utke, J. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. (TOMS)* **1996**, *22*, 131–167. [CrossRef]
18. Bischof, C.H.; Roh, L.; Mauer-Oats, A.J. ADIC: An extensible automatic differentiation tool for ANSI-C. *Softw. Pract. Exp.* **1997**, *27*, 1427–1456. [CrossRef]
19. Magnus, J.R.; Neudecker, H. *Matrix Differential Calculus with Applications in Statistics and Econometrics*; Wiley: Hoboken, NJ, USA, 1999.
20. Glasserman, P. *Monte Carlo Methods in Financial Engineering*; Springer Science & Business Media: New York, NY, USA 2003; Volume 53.

21. Intel. Matrix Inversion: LAPACK Computational Routines. 2020. Available online: https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2023-0/matrix-inversion-lapack-computational-routines.html (accessed on 15 March 2025).
22. Kingma, D.P.; Welling, M. Auto-Encoding Variational Bayes. In Proceedings of the International Conference on Learning Representations, Banff, AB, Canada, 14–16 April 2014.
23. Rosenblatt, M. Remarks on a multivariate transformation. *Ann. Math. Stat.* **1952**, *23*, 470–472. [CrossRef]
24. Kwok, C.F.; Zhu, D.; Jacobi, L. ADtools: Automatic Differentiation Toolbox, R Package Version 0.5.4, CRAN Repository. 2020. Available online: https://cran.r-project.org/src/contrib/Archive/ADtools/ (accessed on 15 March 2025).
25. Kwok, C.F.; Zhu, D.; Jacobi, L. ADtools: Automatic Differentiation Toolbox. GitHub Repository. 2020. Available online: https://github.com/kcf-jackson/ADtools (accessed on 15 March 2025).
26. Abelson, H.; Sussman, G.J.; Sussman, J. *Structure and Interpretation of Computer Programs*; MIT Press: Cambridge, MA, USA, 1996.
27. Lütkepohl, H. *Handbook of Matrices*; Wiley Chichester: Chichester, UK, 1996; Volume 1.
28. Hu, T.; Shing, M. Computation of matrix chain products. Part II. *SIAM J. Comput.* **1984**, *13*, 228–251. [CrossRef]
29. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. *Introduction to Algorithms*; MIT Press: Cambridge, MA, USA, 2009.
30. Chan, J.C.; Jacobi, L.; Zhu, D. An automated prior robustness analysis in Bayesian model comparison. *J. Appl. Econom.* **2019**, *37*, 583–602. [CrossRef]
31. Brennan, M.J.; Chordia, T.; Subrahmanyam, A. Alternative factor specifications, security characteristics, and the cross-section of expected stock returns. *J. Financ. Econ.* **1998**, *49*, 345–373. [CrossRef]
32. Geweke, J.; Zhou, G. Measuring the pricing error of the arbitrage pricing theory. *Rev. Financ. Stud.* **1996**, *9*, 557–587. [CrossRef]
33. Zeiler, M.D. Adadelta: An adaptive learning rate method. *arXiv* **2012**, arXiv:1212.5701.
34. Zellner, A. An efficient method of estimating seemingly unrelated regressions and tests for aggregation bias. *J. Am. Stat. Assoc.* **1962**, *57*, 348–368. [CrossRef]
35. LeSage, J.; Pace, R.K. *Introduction to Spatial Econometrics*; CRC Press: Boca Raton, FL, USA, 2009.
36. Hernández-Sanjaime, R.; González, M.; López-Espín, J.J. Multilevel simultaneous equation model: A novel specification and estimation approach. *J. Comput. Appl. Math.* **2020**, *366*, 112378. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.